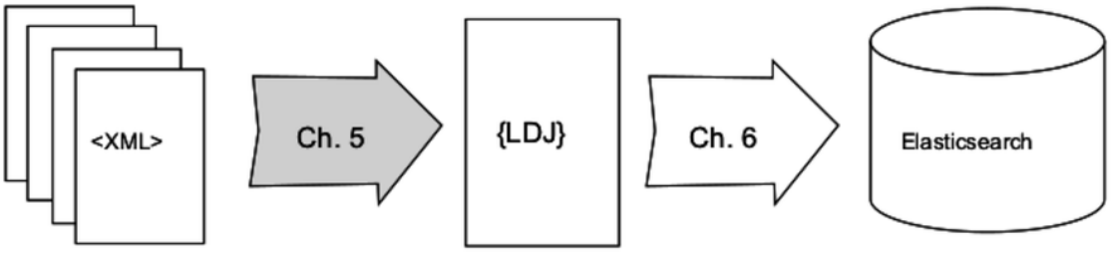
Chapter 5

# Transforming Data and Testing Continuously

Broadly speaking, there are two kinds of data: the kind that your own apps produce and the kind that comes from somewhere else. It would be nice if you only ever had to deal with data that you created. But the reality is that you'll almost certainly have to work with outside data sources during your career, perhaps frequently!

Between this chapter and the next, you'll use Node.js to take real data from the wild and put it into your own local datastore. This work can be neatly approached in two phases: transforming the raw data into an intermediate format, and importing that intermediate data into the datastore.




In this chapter, you'll learn how to use Node.js to transform XML data into the lingua franca of modern data formats, JSON and its close cousin line-delimited JSON (LDJ). Then, in the following chapter, you'll create a command-line tool to bring this LDJ content into Elasticsearch, a NoSQL database that indexes JSON objects.

While writing, testing, and debugging tools to transform raw XML data into LDJ, we'll investigate the following aspects of Node.js:

*Node.js Core*
: Using Chrome DevTools, it's possible to inspect a running Node.js application. You'll learn how to set breakpoints, step through your running Node.js code, and interrogate scoped variables.

*Patterns*
: Much of this chapter involves extracting data from XML files and transforming it into JSON for insertion into a document database. We'll use Cheerio for this, a DOM-based XML parser with a jQuery-like API. To use it effectively, you'll learn the basics of CSS selectors.

*JavaScriptisms*
: In the Node.js ecosystem, it's fairly common to have modules that export a single stateless function rather than a collection of objects, classes, or methods. In this chapter, you'll develop such a module iteratively using behavior-driven development (BDD) techniques.

*Supporting Code*
: We're going to double down on npm in this chapter, adding scripts to launch Mocha tests in standalone mode, continuous testing mode, and debug mode. You'll also learn to use Chai, an assertion library that pairs well with Mocha to write expressive, behavioral tests.

To kick off the chapter, we have to procure the data that we're going to be working with. Then we'll pick through it to get an understanding of the data format, as well as our desired output.

For processing data, it's quite useful to develop unit tests. For this reason, prior to developing the data-processing code we'll set up the infrastructure for continuously running Mocha tests. Moreover, we'll approach the problem through BDD techniques by using Chai, a popular assertion library.

Getting into the nitty-gritty details of querying the raw XML data, we'll use Cheerio, a module that lets you dive into HTML and XML data by using CSS selectors to find interesting elements. Don't worry if you're not yet familiar with writing CSS selectors—it's a useful skill, and we'll build up gradually.

In the final part of this chapter, you'll use the parsing code to roll through the raw data and produce new data that's ready for insertion into a database. You'll learn to walk down a directory tree sequentially, and how to step through your code using Chrome DevTools.

It's a lot to cover, but you can do it. Let's get started!

## Procuring External Data

Before we can start manipulating data with Node.js, we have to get it. The data we'll be using comes from Project Gutenberg, which is dedicated to making public-domain works available as free ebooks.[34]

Project Gutenberg produces catalog download bundles that contain Resource Description Framework (RDF) files for each of its 53,000-plus books. (RDF is an XML-based format.) The `bz2` compressed version of the catalog file is about 40 MB. Fully extracted, it contains a little over 1 GB of RDF files.

To begin, create two sibling directories on your machine, called `databases` and `data`.

```
$ mkdir databases
$ mkdir data
```

The databases project directory will hold all of the programs and configuration files you'll be developing in this chapter. Unless otherwise specified, commands you run will be from a terminal out of this directory.

The data directory will hold the raw data files that we'll be working with. If you want to put this directory somewhere else for storage reasons, that's fine, but the examples in this chapter will assume that it's a sibling of your databases project directory, so modify any paths accordingly.

With that out of the way, open a terminal to your data directory and run the following commands:

```
$ cd data
$ curl -O http://www.gutenberg.org/cache/epub/feeds/rdf-files.tar.bz2
$ tar -xvjf rdf-files.tar.bz2
x cache/epub/0/pg0.rdf
x cache/epub/1/pg1.rdf
x cache/epub/10/pg10.rdf
...
x cache/epub/9998/pg9998.rdf
x cache/epub/9999/pg9999.rdf
x cache/epub/999999/pg999999.rdf
```

This will create a cache directory that contains all the RDF files. Each RDF file is named after its Project Gutenberg ID and contains the metadata about one book. For example, book number 132 is Lionel Giles's 1910 translation of *The Art of War*, by Sunzi.

Here's a very stripped-down excerpt from cache/epub/132/pg132.rdf that shows only the fields that we care about and some surrounding detail:

```
<rdf:RDF>
<pgterms:ebook rdf:about="ebooks/132">
<dcterms:title>The Art of War</dcterms:title>
<pgterms:agent rdf:about="2009/agents/4349">
<pgterms:name>Sunzi, active 6th century B.C.</pgterms:name>
</pgterms:agent>
<pgterms:agent rdf:about="2009/agents/5101">
<pgterms:name>Giles, Lionel</pgterms:name>
</pgterms:agent>
<dcterms:subject>
<rdf:Description rdf:nodeID="N26bb21da0c924e5abcd5809a47f231e7">
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/LCSH"/>
<rdf:value>Military art and science -- Early works to 1800</rdf:value>
</rdf:Description>
</dcterms:subject>
<dcterms:subject>
<rdf:Description rdf:nodeID="N269948d6ecf64b6caf1c15139afd375b">
<rdf:value>War -- Early works to 1800</rdf:value>
```

```
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/LCSH"/>
</rdf:Description>
</dcterms:subject>
</pgterms:ebook>
</rdf:RDF>
```

The important pieces of information that we'd like to extract are as follows:

- The Gutenberg ID (132)
- The book's title
- The list of authors (agents)
- The list of subjects

Ideally, we'd like to have all of this information formatted as a JSON document suitable for passing in to a document database. For this particular book, our desired JSON would be this:

```
{
"id": 132,
"title": "The Art of War",
"authors": [
"Sunzi, active 6th century B.C.",
"Giles, Lionel"
],
"subjects": [
"Military art and science -- Early works to 1800",
"War -- Early works to 1800"
]
}
```

To get this nice JSON representation, we'll have to parse the RDF file. On the way there, this provides a great opportunity to explore the BDD pattern.

## 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*, 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 describecallback 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.[35]

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

## Extracting Data from XML with Cheerio

At this point, you should have two successfully passing tests in test/parse-rdf-test.js that are powered by your module in lib/parse-rdf.js. In this section, we'll expand the tests to cover all of our requirements for parsing Project Gutenberg RDF files, and implement the library code to make them pass.

To extract the data attributes we desire, we'll need to parse the RDF (XML) file. As with everything in the Node.js ecosystem, there are multiple valid approaches to parsing, navigating, and querying XML files.

Let's discuss some of the options, then move on to installing and using Cheerio.

### Considering XML Data Extraction Options

In this chapter, we will be treating the RDF files like regular, undifferentiated XML files for parsing and for data extraction. The benefit to you (as opposed to addressing them specifically as RDF/XML) is that the skills and techniques you learn will transfer to parsing other kinds of XML and HTML documents.

For situations like this, where the documents are relatively small, I prefer to use Cheerio, a fast Node.js module that provides a jQuery-like API for working with HTML and XML documents.[36] Cheerio's big advantage is that it offers a convenient way to use CSS selectors to dig into the document, without the overhead of setting up a browser-like environment.

Cheerio isn't the only DOM-like XML parser for Node.js—far from it. Other popular options include xmldom and jsdom,[37] [38] both of which are based on the W3C's DOM specification.

In your own projects, if the XML files that you're working with are quite large, then you're probably going to want a streaming SAX parser instead. SAX, which stands for *Simple API for XML*, treats XML as a

stream of tokens that your program digests in sequence. Unlike a DOM parser, which considers the document as a whole, a SAX parser operates on only a small piece at a time.

Compared to DOM parsers, SAX parsers can be quite fast and memory-efficient. But the downside of using a SAX parser is that your program will have to keep track of the document structure in flight. I've had good experiences using the sax Node.js module for parsing large XML files.[39]

Speaking of RDF/XML in particular, it's a rich data format for which custom tooling is available. If you find yourself working with linked data in the wild, you may find it more convenient to convert it to JSON for Linked Data (JSON-LD) and then perform additional operations from there.

JSON-LD is to JSON as RDF is to XML.[40] With JSON-LD, you can express relationships between entities, not just a hierarchical structure like JSON allows. The jsonld module would be a good place to start for this.[41]

Which of these approaches is best for you really comes down to your use case and personal taste. If your documents are large, then you'll probably want a SAX parser. If you need to preserve the structured relationships in the data, then JSON-LD may be best. Do you need to fetch remote documents? Some modules have this capability built in (Cheerio does not).

Our task at hand is to extract a small amount of data from relatively small files that are readily available locally. I find Cheerio to be an excellent fit for this particular kind of task, and I hope you will too!

## Getting Started with Cheerio

To get started with Cheerio, install it with npm and save the dependency.

```
$ npm install --save --save-exact cheerio@0.22.0
```

Please be careful with the version number here. Cheerio has not historically followed the semantic versioning convention, introducing breaking changes in minor releases. If you install any version other than 0.22.0, the examples in this book may not work.

Before we start using Cheerio, let's create some BDD tests that we can make pass by doing so. If Mocha is not already running continuously, open a terminal to your databases project directory and run the following:

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

It should clear the screen and report two passing tests:

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

Great; now let's require that the book object returned by parseRDF has the correct numeric ID for *The Art of War*. Open your parse-rdf-test.js file and

expand the second test by adding a check that the `book` object has an `id`property containing the number `132`.

databases/test/parse-rdf-test.js

```
it('should parse RDF content', () => {
const book = parseRDF(rdf);
expect(book).to.be.an('object');
expect(book).to.have.a.property('id', 132);
});
```

This code takes advantage of Chai's sentence-like BDD API, which we'll use in increasing doses as we add more tests.

Since we have not yet implemented the code to include the `id` in the returned `book` object, as soon as you save the file, your Mocha terminal should report this:

```
1 passing (4ms)
1 failing

1) parseRDF should parse RDF content:
AssertionError: expected {} to have a property 'id'
at Context.it (test/parse-rdf-test.js:32:28)
```

Good! The test is failing exactly as we expect it should.

Now it's time to use Cheerio to pull out the four fields we want: the book's ID, the title, the authors, and the subjects.

## Reading Data from an Attribute

The first piece of information we hope to extract using Cheerio is the book's ID. Recall that we're trying to grab the number `132` out of this XML tag:

```
<pgterms:ebook rdf:about="ebooks/132">
```

Open your `lib/parse-rdf.js` file and make it look like the following:

databases/lib/parse-rdf.js

```
'use strict';
const cheerio = require('cheerio');

module.exports = rdf => {
const $ = cheerio.load(rdf);

const book = {};

book.id = +$('pgterms\\:ebook').attr('rdf:about').replace('ebooks/', '');

return book;
};
```

This code adds three things to the version listed in *Enabling Continuous Testing with Mocha*:

- At the top, we now require Cheerio.

- Inside the exported function, we use Cheerio's `load` method to parse the `rdf` content. The `$` function that's returned is very much like jQuery's `$` function.
- Using Cheerio's APIs, we extract the book's ID and, finally, format it.

The line where we set `book.id` is fairly dense, so let's break it down. Here's the same line, but split out and commented so we can dissect it:

```
book.id = // Set the book's id.
+ // Unary plus casts the result as a number.
$('pgterms\\:ebook') // Query for the <pgterms:ebook> tag.
.attr('rdf:about') // Get the value of the rdf:about attribute.
.replace('ebooks/', ''); // Strip off the leading 'ebooks/' substring.
```

In CSS, the colon character (`:`) has special meaning—it is used to introduce *pseudo selectors* like `:hover` for links that are hovered over. In our case, we need a literal colon character for the `<pgterms:ebook>` tag name, so we have to escape it with a backslash. But since the backslash is a special character in JavaScript string literals, that too needs to be escaped-. Thus, our query selector for finding the tag is `pgterms\\:ebook`.

Once we have selected the `pgterms:ebook` tag, we pull out the `rdf:about` attribute value and strip off the leading `ebooks/` substring, leaving only the string `"132"`. The leading unary plus (`+`) at the start of the line ensures that this gets cast as a number.

If all has gone well so far, your terminal running Mocha's continuous testing should again read `2 passing`.

## Reading the Text of a Node

Next, let's add a test for the title of the book. Insert the following code right after the test for the book's ID.

databases/test/parse-rdf-test.js

```
expect(book).to.have.a.property('title', 'The Art of War');
```

Your continuous testing terminal should read as follows:

```
1 passing (3ms)
1 failing

1) parseRDF should parse RDF content:
AssertionError: expected { id: 132 } to have a property 'title'
at Context.it (test/parse-rdf-test.js:35:28)
```

Now let's grab the title and add it to the returned `book` object. Recall that the XML containing the title looks like this:

```
<dcterms:title>The Art of War</dcterms:title>
```

Getting this content is even easier than extracting the ID. Add the following to your `parse-rdf.js` file, after the line where we set `book.id`:

databases/lib/parse-rdf.js

```
book.title = $('dcterms\\:title').text();
```

Using Cheerio, we select the tag named dcterms:title and save its text content to the book.text property. Once you save this file, your tests should pass again.

## Collecting an Array of Values

Moving on, let's add tests for the array of book authors. Open your parse-rdf-test.js file and add these lines:

databases/test/parse-rdf-test.js

```
expect(book).to.have.a.property('authors')
.that.is.an('array').with.lengthOf(2)
.and.contains('Sunzi, active 6th century B.C.')
.and.contains('Giles, Lionel');
```

Here we really start to see the expressive power of Chai assertions. This line of code reads almost like an English sentence.

Expect book to have a property called authors that is an array of length two and contains "Sunzi, active 6th century B.C." and "Giles, Lionel".

In Chai's language-chaining model, words like and, that, and which are largely interchangeable. This lets you write clauses like .and.contains('X') or .that.contains('X'), depending on which version reads better in your test case.

Once you save this change, your continuous testing terminal should again report a test failure:

```
1 passing (11ms)
1 failing

1) parseRDF should parse RDF content:
AssertionError: expected { id: 132, title: 'The Art of War' } to have a
property 'authors'
at Context.it (test/parse-rdf-test.js:39:28)
```

To make the test pass, recall that we will need to pull out the content from these tags:

```
<pgterms:agent rdf:about="2009/agents/4349">
<pgterms:name>Sunzi, active 6th century B.C.</pgterms:name>
</pgterms:agent>
<pgterms:agent rdf:about="2009/agents/5101">
<pgterms:name>Giles, Lionel</pgterms:name>
</pgterms:agent>
```

We're looking to extract the text of each <pgterms:name> tag that's a child of a <pgterms:agent>. The CSS selector *pgterms:agent pgterms:name* finds the elements we need, so we can start with this:

```
$('pgterms\\:agent pgterms\\:name')
```

You might be tempted to grab the text straight away like this:

```
book.authors = $('pgterms\\:agent pgterms\\:name').text();
```

But unfortunately, this won't give us what we want, because Cheerio's text method returns a single string and we need an array of strings. Instead, add the following code to your parse-rdf.js file, after the book.title piece, to correctly extract the authors:

databases/lib/parse-rdf.js

```
book.authors = $('pgterms\\:agent pgterms\\:name')
.toArray().map(elem => $(elem).text());
```

Calling Cheerio's .toArray method converts the collection object into a true JavaScript Array. This allows us to use the native map method to create a new array by calling the provided function on each element and grabbing the returned value.

Unfortunately, the collection of objects that comes out of toArray doesn't consist of Cheerio-wrapped objects, but rather document nodes. To extract the text using Cheerio's text, we need to wrap each node with the $ function, then call text on it. The resulting mapping function is elem => $(elem).text().

## Traversing the Document

Finally, we're down to just one more piece of information we wanted to pull from the RDF file—the list of subjects.

```
<dcterms:subject>
<rdf:Description rdf:nodeID="N26bb21da0c924e5abcd5809a47f231e7">
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/LCSH"/>
<rdf:value>Military art and science -- Early works to 1800</rdf:value>
</rdf:Description>
</dcterms:subject>
<dcterms:subject>
<rdf:Description rdf:nodeID="N269948d6ecf64b6caf1c15139afd375b">
<rdf:value>War -- Early works to 1800</rdf:value>
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/LCSH"/>
</rdf:Description>
</dcterms:subject>
```

As with previous examples, let's start by adding a test. Insert the following code into your parse-rdf-test.js after the other tests.

databases/test/parse-rdf-test.js

```
expect(book).to.have.a.property('subjects')
.that.is.an('array').with.lengthOf(2)
.and.contains('Military art and science -- Early works to 1800')
.and.contains('War -- Early works to 1800');
```

Unfortunately, these subjects are a little trickier to pull out than the authors were. It would be nice if we could use the tag structure to craft a simple CSS selector like this:

```
$('dcterms\\:subject rdf\\:value')
```

However, this selector would match another tag in the document, which we don't want.

```
<dcterms:subject>
<rdf:Description rdf:nodeID="Nfb797557d91f44c9b0cb80a0d207eaa5">
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/LCC"/>
<rdf:value>U</rdf:value>
</rdf:Description>
</dcterms:subject>
```

To spot the difference, look at the `<dcam:memberOf>` tags' `rdf:resource` URLs. The ones we want end in `LCSH`, which stands for Library of Congress Subject Headings.[42] These headings are a collection of rich indexing terms used in bibliographic records.

Contrast that with the tag we don't want to match, which ends in `LCC`. This stands for Library of Congress Classification.[43] These are codes that divide all knowledge into 21 top-level classes (like `U` for Military Science) with many subclasses. These could be interesting in the future, but right now we only want the Subject Headings.

With your continuous test still failing, here's the code you can add to your `parse-rdf.js` to make it pass:

databases/lib/parse-rdf.js

```
book.subjects = $('[rdf\\:resource$="/LCSH"]')
.parent().find('rdf\\:value')
.toArray().map(elem => $(elem).text());
```

Let's break this down. First, we select the `<dcam:memberOf>` tags of interest with the CSS selector `[rdf\:resource$="/LCSH"]`. The brackets introduce a CSS *attribute selector*, and the `$=` indicates that we want elements whose `rdf:resource` attribute ends with `/LCSH`.

Next, we use Cheerio's `.parent` method to traverse up to our currently selected elements' parents. In this case, those are the `<rdf:Description>` tags. Then we traverse back down using `.find` to locate all of their `<rdf:value>` tags.

Lastly, just like with the book authors, we convert the Cheerio selection object into a true `Array` and use `.map` to get each element's text. And that's it! At this point your tests should be passing, meaning your `parseRDF` function is correctly extracting the data we want.

## Anticipating Format Changes

One quick note before we move on—an older version of the Project Gutenberg RDF format had its subjects listed like this:

```
<dcterms:subject>
<rdf:Description>
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/LCSH"/>
<rdf:value>Military art and science -- Early works to 1800</rdf:value>
<rdf:value>War -- Early works to 1800</rdf:value>
</rdf:Description>
</dcterms:subject>
```

Instead of finding each subject's <rdf:value> living in its own <dcterms:subject> tag, we find them bunched together under a single one. Now consider the traversal code we just wrote. By finding the /LCSH tag, going up to its parent <rdf:Description>, and then searching down for <rdf:value> tags, our code would work with both this earlier data format and the current one (at the time of this writing, anyway).

Whenever you work with third-party data, there's a chance that it could change over time. When it does, your code may or may not continue to work as expected. There's no hard and fast rule to tell you when to be more or less specific with your data-processing code, but I encourage you to stay vigilant to these kinds of issues in your work.

The beauty of testing in these scenarios is that when a data format changes, you can add more tests. This gives you confidence that you're meeting the new demands of the updated data format while still honoring past data.

## Recapping Data Extraction with Cheerio

After all of the incremental additions of the last several sections, here's what your final parse-rdf-test.js should look like:

databases/test/parse-rdf-test.js

```
'use strict';

const fs = require('fs');
const expect = require('chai').expect;
const parseRDF = require('../lib/parse-rdf.js');

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

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

it('should parse RDF content', () => {
const book = parseRDF(rdf);

expect(book).to.be.an('object');
expect(book).to.have.a.property('id', 132);
expect(book).to.have.a.property('title', 'The Art of War');

expect(book).to.have.a.property('authors')
.that.is.an('array').with.lengthOf(2)
.and.contains('Sunzi, active 6th century B.C.')
.and.contains('Giles, Lionel');

expect(book).to.have.a.property('subjects')
.that.is.an('array').with.lengthOf(2)
.and.contains('Military art and science -- Early works to 1800')
.and.contains('War -- Early works to 1800');
```

```
});
});
```

And here's the parse-rdf.js itself:

databases/lib/parse-rdf.js

```
'use strict';
const cheerio = require('cheerio');

module.exports = rdf => {
const $ = cheerio.load(rdf);

const book = {};

book.id = +$('pgterms\\:ebook').attr('rdf:about').replace('ebooks/', '');

book.title = $('dcterms\\:title').text();

book.authors = $('pgterms\\:agent pgterms\\:name')
.toArray().map(elem => $(elem).text());
book.subjects = $('[rdf\\:resource$="/LCSH"]')
.parent().find('rdf\\:value')
.toArray().map(elem => $(elem).text());

return book;
};
```

Using this, we can now quickly put together a command-line program to explore some of the other RDF files. Open your editor and enter this:

databases/rdf-to-json.js

```
#!/usr/bin/env node
const fs = require('fs');
const parseRDF = require('./lib/parse-rdf.js');
const rdf = fs.readFileSync(process.argv[2]);
const book = parseRDF(rdf);
console.log(JSON.stringify(book, null, ' '));
```

Save this file as rdf-to-json.js in your databases project directory. This program simply takes the name of an RDF file, reads its contents, parses them, and then prints the resulting JSON to standard output.

Previously when calling JSON.stringify, we passed only one argument, the object to be serialized. Here we're passing three arguments to get a prettier output. The second argument (null) is an optional replacer function that can be used for filtering (this is almost never used in practice). The last argument (' ') is used to indent nested objects, making the output more human-readable.

Let's try it! Open a terminal to your databases project directory and run this:

```
$ node rdf-to-json.js ../data/cache/epub/11/pg11.rdf
{
    "id": 11,
    "title": "Alice's Adventures in Wonderland",
```

```
    "authors": [
        "Carroll, Lewis"
    ],
    "subjects": [
        "Fantasy"
    ]
}
```

If you see this, great! It's time to start performing these conversions in bulk.

## Processing Data Files Sequentially

By now your lib/parse-rdf.js is a robust module that can reliably convert RDF content into JSON documents. All that remains is to walk through the Project Gutenberg catalog directory and collect all the JSON documents.

More concretely, we need to do the following:

1. Traverse down the data/cache/epub directory looking for files ending in rdf.
2. Read each RDF file.
3. Run the RDF content through parseRDF.
4. Collect the JSON serialized objects into a single, bulk file for insertion.

The NoSQL database we'll be using is Elasticsearch, a document datastore that indexes JSON objects. Soon, in Chapter 6, *Commanding Databases*, we'll dive deep into Elasticsearch and how to effectively use it with Node.js. You'll learn how to install it, configure it, and make the most of its HTTP-based APIs.

For now, though, our focus is just on transforming the Gutenberg data into an intermediate form for bulk import.

Conveniently, Elasticsearch has a bulk-import API that lets you pull in many records at once. Although we could insert them one at a time, it is significantly faster to use the bulk-insert API.

The format of the file we need to create is described on Elasticsearch's Bulk API page.[44] It's an LDJ file consisting of actions and the source objects on which to perform each action.

In our case, we're performing *index* operations—that is, inserting new documents into an index. Each source object is the book object returned by parseRDF. Here's an example of an action followed by its source object:

```
{"index":{"_id":"pg11"}}
{"id":11,"title":"Alice's Adventures in Wonderland","authors":...}
```

And here's another one:

```
{"index":{"_id":"pg132"}}
{"id":132,"title":"The Art of War","authors":...}
```

In each case, an action is a JSON object on a line by itself, and the source object is another JSON object on the next line. Elasticsearch's bulk API allows you to chain any number of these together like so:

```
{"index":{"_id":"pg11"}}
{"id":11,"title":"Alice's Adventures in Wonderland","authors":...}
{"index":{"_id":"pg132"}}
{"id":132,"title":"The Art of War","authors":...}
```

The `_id` field of each index operation is the unique identifier that Elasticsearch will use for the document. Here I've chosen to use the string `pg` followed by the Project Gutenberg ID. This way, if we ever wanted to store documents from another source in the same index, they shouldn't collide with the Project Gutenberg book data.

To find and open each of the RDF files under the `data/cache/epub` directory, we will use a module called `node-dir`. Install and save it as usual. Then we will begin like this:

```
$ npm install --save --save-exact node-dir@0.1.16
```

This module comes with a handful of useful methods for walking a directory tree. The method we'll use is `readFiles`, which sequentially operates on files as it encounters them while walking a directory tree.

Let's use this method to find all the RDF files and send them through our RDF parser. Open a text editor and enter this:

databases/rdf-to-bulk.js

```
'use strict';

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

const dirname = process.argv[2];

const options = {
match: /\.rdf$/, // Match file names that in '.rdf'.
exclude: ['pg0.rdf'], // Ignore the template RDF file (ID = 0).
};

dir.readFiles(dirname, options, (err, content, next) => {
if (err) throw err;
const doc = parseRDF(content);
console.log(JSON.stringify({ index: { _id: `pg${doc.id}` } }));
console.log(JSON.stringify(doc));
next();
});
```

Save the file as rdf-to-bulk.js in your databases project directory. This short program walks down the provided directory looking for files that end in rdf, but excluding the template RDF file called pg0.rdf.

As the program reads each file's content, it runs it through the RDF parser. For output, it produces JSON serialized actions suitable for Elasticsearch's bulk API.

Run the program, and let's see what it produces.

```
$ node rdf-to-bulk.js ../data/cache/epub/ | head
```

If all went well, you should see 10 lines consisting of interleaved actions and documents—like the following, which has been truncated to fit on the page.

```
{"index":{"_id":"pg1"}}
{"id":1,"title":"The Declaration of Independence of the United States of Ame...
{"index":{"_id":"pg10"}}
{"id":10,"title":"The King James Version of the Bible","authors":[],"subject...
{"index":{"_id":"pg100"}}
{"id":100,"title":"The Complete Works of William Shakespeare","authors":["Sh...
{"index":{"_id":"pg1000"}}
{"id":1000,"title":"La Divina Commedia di Dante: Complete","authors":["Dante...
{"index":{"_id":"pg10000"}}
{"id":10000,"title":"The Magna Carta","authors":["Anonymous"],"subjects":["M...
```

Because the head command closes the pipe after echoing the beginning lines, this can sometimes cause Node.js to throw an exception, sending the following to the standard error stream:

```
events.js:160
throw er; // Unhandled 'error' event
^

Error: write EPIPE
at exports._errnoException (util.js:1022:11)
at WriteWrap.afterWrite [as oncomplete] (net.js:804:14)
```

To mitigate this error, you can capture error events on the process.stdout stream. Try adding the following line to rdf-to-bulk.js and rerunning it.

```
process.stdout.on('error', err => process.exit());
```

Now, when head closes the pipe, the next attempt to use console.log will trigger the error event listener and the process will exit silently. If you're worried about output errors other than EPIPE, you can check the err object's code property and take action as appropriate.

```
process.stdout.on('error', err => {
if (err.code === 'EPIPE') {
process.exit();
}
throw err; // Or take any other appropriate action.
});
```

At this point we're ready to let rdf-to-bulk.js run for real. Use the following command to capture this LDJ output in a new file called bulk_pg.ldj.

```
$ node rdf-to-bulk.js ../data/cache/epub/ > ../data/bulk_pg.ldj
```

This will run for quite a while, as rdf-to-bulk.js traverses the epub directory, parses each file, and tacks on the Elasticsearch action for it. When it's finished, the bulk_pg.ldj file should be about 11 MB.

## Debugging Tests with Chrome DevTools

The examples so far in this chapter may have given you an overly harmonious view of what it's like to develop a data-transformation program in Node.js. The reality is that as you learn the APIs and explore the data, you'll frequently make mistakes and want to track down where you went wrong.

Fortunately, it's possible to attach Chrome's DevTools to Node.js, bringing the full power of Chrome's debugging features with it. If you've done any serious web programming, then you're probably already familiar with Chrome's DevTools—if so, this will be a refresher on how to use them.

In this section, you'll learn how to start up your continuous test suite with Mocha in such a way that you can attach Chrome DevTools and step through the code at your own pace. You'll also be able to execute commands interactively through the console, set breakpoints, and expect variables.

### Running Mocha in Debug Mode with npm

So far, to run Mocha tests we've used npm test and npm run test:watch, both of which trigger scripts defined in the project's package.json. Now we'll add a new script called test:debug that runs Mocha in a way that allows the Chrome DevTools to become attached.

Unfortunately, the mocha command we've been using doesn't make it easy, because it spawns a child Node.js process to carry out the tests. So we need to go one level deeper.

When you use npm to install Mocha, it puts two command-line programs into node_modules/mocha/bin: mocha(which we've been using) and _mocha (note the leading underscore). The former invokes the latter in a newly spawned child Node.js process when you use mocha from the command line or through npm.

To attach the Node.js debugger, we have to cut out the middleman and invoke _mocha directly. Open your package.json, and add the following test:debug script to the scripts section.

```
  "scripts": {
  "test": "mocha",
  "test:watch": "mocha --watch --reporter min",
» "test:debug":
» "node --inspect node_modules/mocha/bin/_mocha --watch --no-timeouts"
  },
```

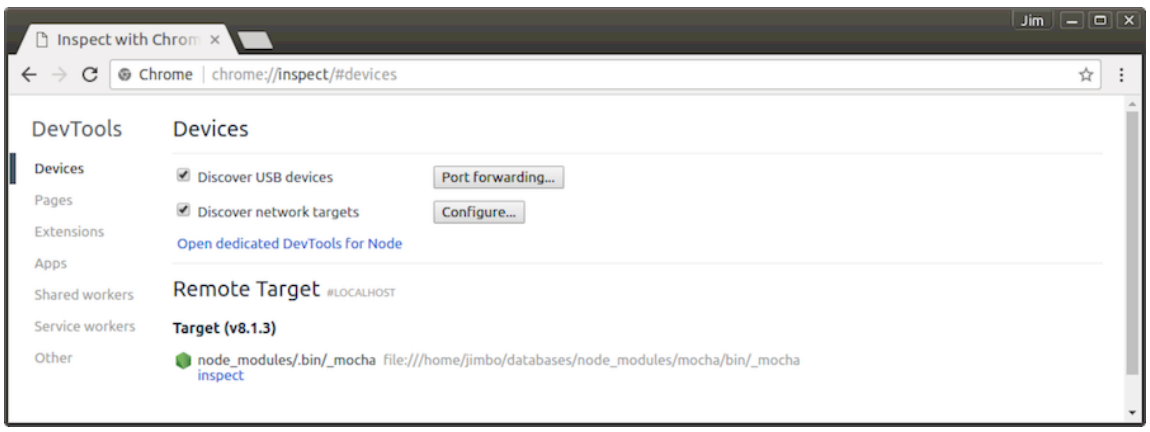
The `--inspect` flag tells Node.js that we intend to run in debug mode, which will output a special URL you can open in Chrome to attach DevTools to the process. The `--watch` flag you're already familiar with—it tells Mocha to watch files for changes and rerun the tests when they happen.

Finally, the `--no-timeouts` flag tells Mocha that we don't care how long tests take to complete. By default, Mocha will time out asynchronous tests and call them failing after two seconds. But if you're engaged in step-through debugging, it may take significantly longer.

After you save the file, try out `npm run test:debug` to see what happens.

```
$ npm run test:debug

> databases@1.0.0 test:debug ./code/databases
> node --inspect node_modules/mocha/bin/_mocha --watch --no-timeouts

Debugger listening on ws://127.0.0.1:9229/06a172b5-2bee-475d-b069-0da65d1ea2af
For help see https://nodejs.org/en/docs/inspector



parseRDF
✓ should be a function
✓ should parse RDF content


2 passing (35ms)
```

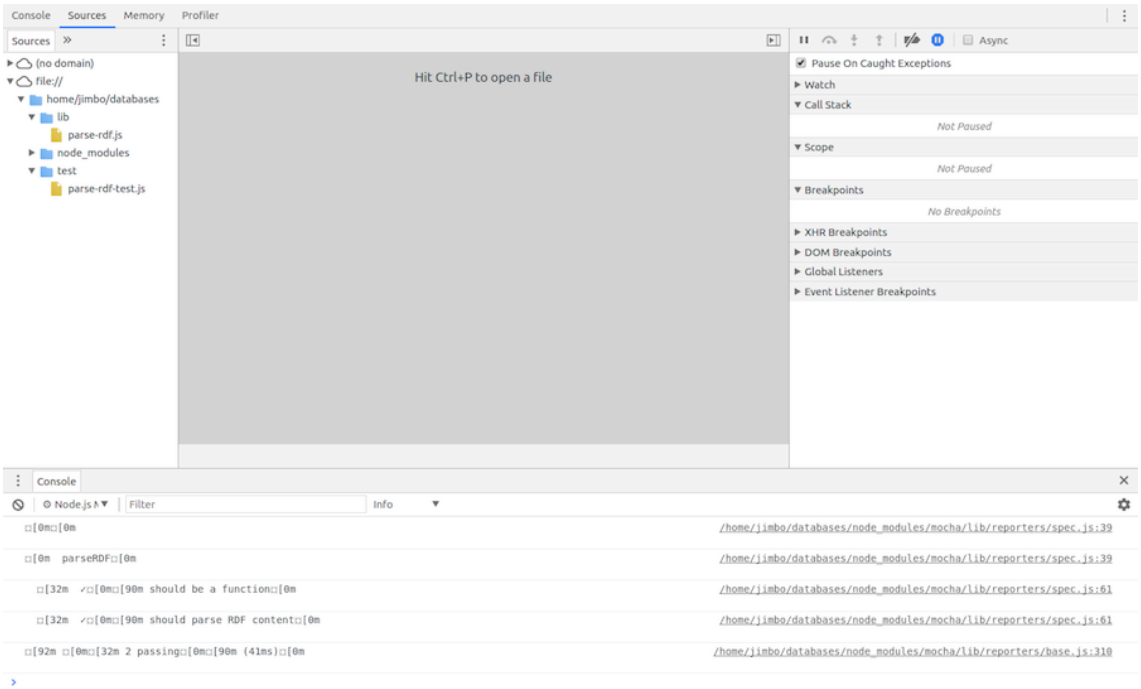
The special URL beginning with `ws://` is a WebSocket that Chrome can connect to for debugging. Open a Chrome browser and navigate to `chrome://inspect`. This will take you to the Devices page of Chrome DevTools.

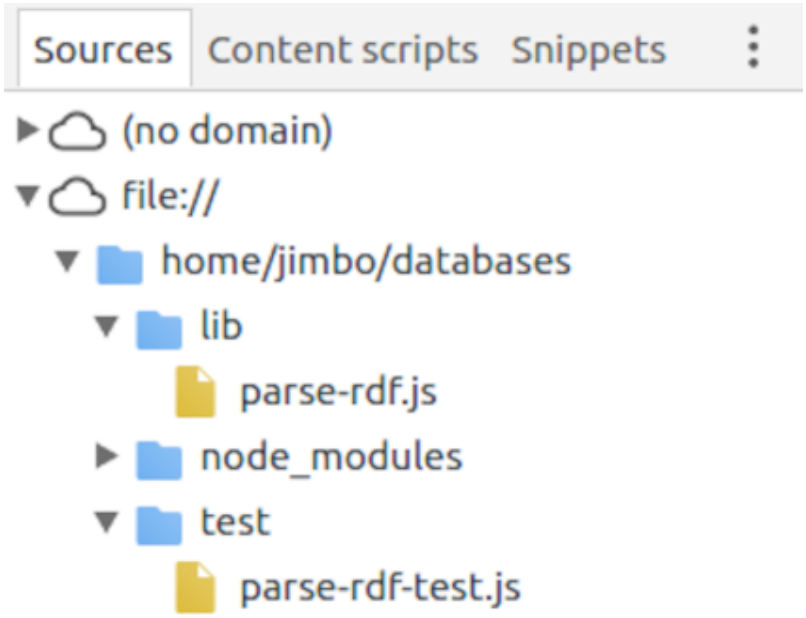


Under the heading Remote Target #LOCALHOST, you should see an entry for your Node.js process running Mocha. Click the blue *inspect* link to launch the debugger.

## Using Chrome DevTools to Step Through Your Code

At this point, you should have your Chrome browser running, with a DevTools window open and connected to your Node.js debugging session. When you press Enter, Chrome should bring up Chrome DevTools attached to your process. Then make sure you have the Sources tab selected.



In the left pane, under the file:// heading, you will find the hierarchy of directories and files we have been working on. Under the lib directory you should see parse-rdf.js, and under test there should be the parse-rdf-test.jsfile.

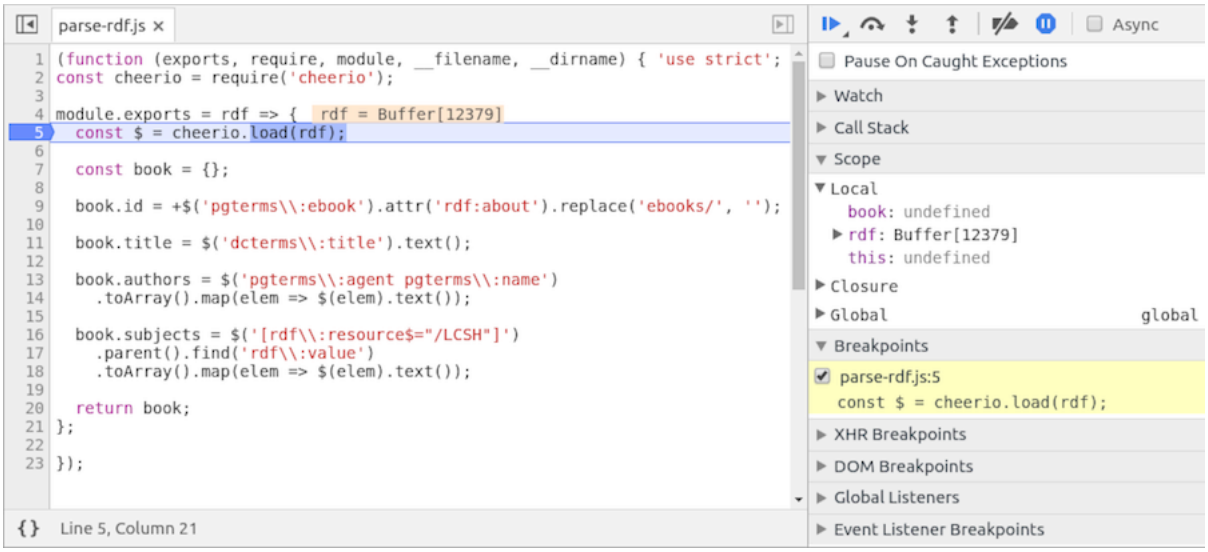
Select the parse-rdf.js file to bring up its contents in the center panel. You can set breakpoints by clicking on the line numbers. Set one now, inside but near the top of the module's exported function (as shown in the first figure).

```
1  (function (exports, require, module, __filename, __dirname) { 'use strict';
2  const cheerio = require('cheerio');
3
4  module.exports = rdf => {
5    const $ = cheerio.load(rdf);
6
7    const book = {};
8
9    book.id = +$('pgterms\\:ebook').attr('rdf:about').replace('ebooks/', '');
10
11   book.title = $('dcterms\\:title').text();
12
13   book.authors = $('pgterms\\:agent pgterms\\:name')
14     .toArray().map(elem => $(elem).text());
15
16   book.subjects = $('[rdf\\:resource$="/LCSH"]')
17     .parent().find('rdf\\:value')
18     .toArray().map(elem => $(elem).text());
19
20   return book;
21 };
22
23 });
```

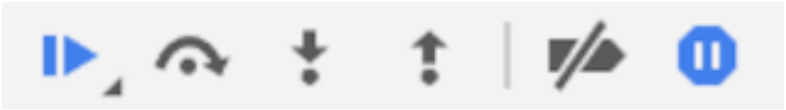
Since Mocha is running in watch mode, any time a file changes it will rerun the tests, hitting the breakpoint. So to trigger a test run, open a terminal to your databases project directory and touch either file.

```
$ touch test/parse-rdf-test.js
```

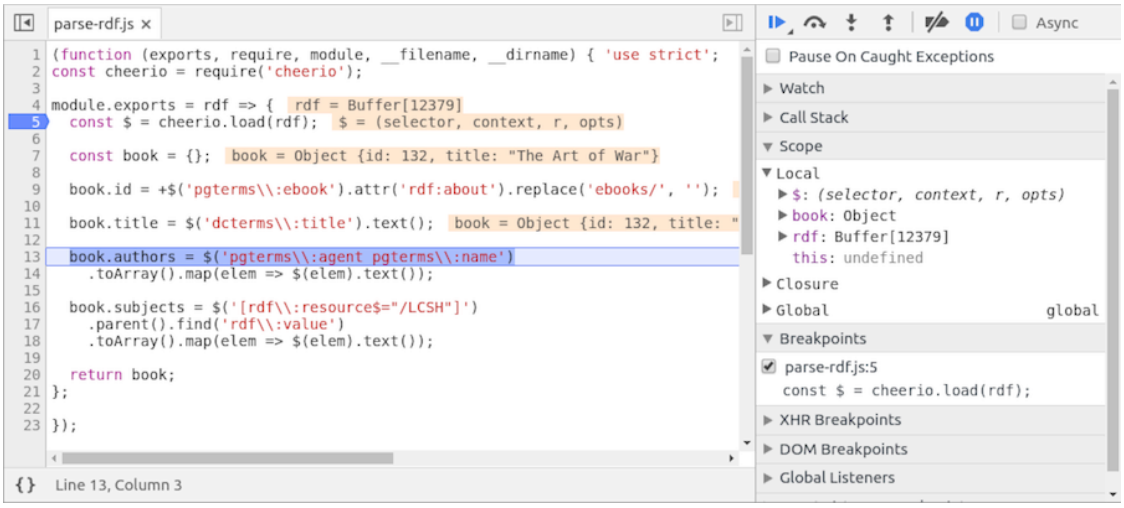
Back in Chrome DevTools, the test run should now be paused at your breakpoint (as shown in the second figure).




You can use the clickable icons at the top of the right-hand sidebar to step through your code.



As you step forward through the code, DevTools will decorate the source view with information about the current state. You can also explore the available variables and their contents in the Scope section of the right-hand sidebar (as shown in the figure).




At the time of this writing, a few important features are missing from the Node.js DevTools experience. Most notably, although DevTools appears to allow you to make changes to the source files locally in the browser, it doesn't give you a way to save those changes to disk. And without the

ability to save, your Node.js process (Mocha) won't be able to see the changes and run the tests.

## Wrapping Up

In this first chapter of Part II of the book, we started working with data from external sources. Acquiring, transforming, storing, and querying data are crucial skills in modern software development with Node.js.

Using Project Gutenberg's catalog data, you iteratively developed the code and tests to parse and make sense of RDF (XML) files. This allowed us to use Mocha and to harness the expressive power of Chai, an assertion library that facilitates for BDD.

For the nuts and bolts of parsing and querying the XML documents, you used Cheerio, a Node.js module that provides a jQuery-like API. Although we didn't use a lot of CSS, we used some sophisticated selectors to pick out specific elements, then we walked the DOM using Cheerio's methods to extract data.

Once this robust parsing library was complete, we used it in combination with the `node-dir` module to create `rdf-to-bulk.js`. This program walks down a directory tree looking for RDF files, parses each one, and collects the resulting output objects. You'll use this intermediate, bulk data file in the following chapter to populate an Elasticsearch index.

Finally, you learned how to launch a Node.js program in debug mode and attach Chrome DevTools for interactive, step-through debugging. While there are certainly some kinks that need to be worked out, it sure beats debugging by gratuitous `console.log`!

Whereas this chapter was all about manipulating input data and transforming it into a usable form, the next chapter is about storing this data and querying it from a database. In particular, we're going to use Elasticsearch, a full-text indexing, JSON-based document datastore. With its RESTful, HTTP-based API, working with Elasticsearch will let us use Node.js in new and interesting ways.

In case you'd like to have more practice with the techniques we used in this chapter, the following tasks ask you to think about how you would pull out even more data from the RDF files we've been looking at. Good luck!

### Extracting Classification Codes

When extracting fields from the Project Gutenberg RDF (XML) files, in *Traversing the Document*, we specifically selected the Library of Congress Subject Headings (LCSH) and stored them in an array called `subjects`. At that time, we carefully avoided the Library of Congress

Classification (LCC) single-letter codes. Recall that the LCC portion of an RDF file looks like this:

```
<dcterms:subject>
<rdf:Description rdf:nodeID="Nfb797557d91f44c9b0cb80a0d207eaa5">
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/LCC"/>
<rdf:value>U</rdf:value>
</rdf:Description>
</dcterms:subject>
```

Using your BDD infrastructure built on Mocha and Chai, implement the following:

- Add a new assertion to `parse-rdf-test.js` that checks for `book.lcc`. It should be of type `string` and it should be at least one character long. It should start with an uppercase letter of the English alphabet, but not `I`, `O`, `W`, `X`, or `Y`.
- Run the tests to see that they fail.
- Add code to your exported module function in `parse-rdf.js` to make the tests pass.

Hint: When working on the code, use Cheerio to find the `<dcam:memberOf>` element with an `rdf:resource` attribute that ends with `/LCC`. Then traverse up to its parent `<rdf:Description>`, and read the text of the first descendent `<rdf:value>` tag. You may want to refer to Chai's documentation when crafting your new assertions.[45]

## Extracting Sources

Most of the metadata in the Project Gutenberg RDF files describes where each book can be downloaded in various formats. For example, here's the part that shows where to download the plain text of *The Art of War*:

```
<dcterms:hasFormat>
<pgterms:file rdf:about="http://www.gutenberg.org/ebooks/132.txt.utf-8">
<dcterms:isFormatOf rdf:resource="ebooks/132"/>
<dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
2016-09-01T01:20:00.437616</dcterms:modified>
<dcterms:format>
<rdf:Description rdf:nodeID="N2293d0caa918475e922a48041b06a3bd">
<dcam:memberOf rdf:resource="http://purl.org/dc/terms/IMT"/>
<rdf:value
rdf:datatype="http://purl.org/dc/terms/IMT">text/plain</rdf:value>
</rdf:Description>
</dcterms:format>
<dcterms:extent rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
343691</dcterms:extent>
</pgterms:file>
</dcterms:hasFormat>
```

Suppose we wanted to include a list of download sources in each JSON object we create from an RDF file. To get an idea of what data you might want, take a look at the Project Gutenberg page for *The Art of War*.[46]

Consider these questions:

- Which fields in the raw data would we want to capture, and which could we discard?
- What structure would make the most sense for this data?
- What information would you need to be able to produce a table that looked like the one on the Project Gutenberg site?

Once you have an idea of what data you'll want to extract, try creating a JSON object by hand for this one download source. When you're happy with your data representation, use your existing continuous testing infrastructure and add a test that checks for this new information.

Finally, extend the `book` object produced in `parse-rdf.js` to include this data to make the test pass. You can do it!

## Footnotes

[34] http://www.gutenberg.org
[35] https://mochajs.org/#reporters
[36] https://www.npmjs.com/package/cheerio
[37] https://www.npmjs.com/package/xmldom
[38] https://www.npmjs.com/package/jsdom
[39] https://www.npmjs.com/package/sax
[40] https://json-ld.org/spec/latest/json-ld/
[41] https://www.npmjs.com/package/jsonld
[42] https://en-wikipedia-org.accedys2.bbtk.ull.es/wiki/Library_of_Congress_Subject_Headings
[43] https://en-wikipedia-org.accedys2.bbtk.ull.es/wiki/Library_of_Congress_Classification
[44] https://www.elastic.co/guide/en/elasticsearch/reference/5.2/docs-bulk.html
[45] http://chaijs.com/api/bdd/
[46] http://www.gutenberg.org/ebooks/132