Extracted from:

Node.js 8 the Right Way

Practical, Server-Side JavaScript That Scales

The Pragmatic Bookshelf

Raleigh, North Carolina

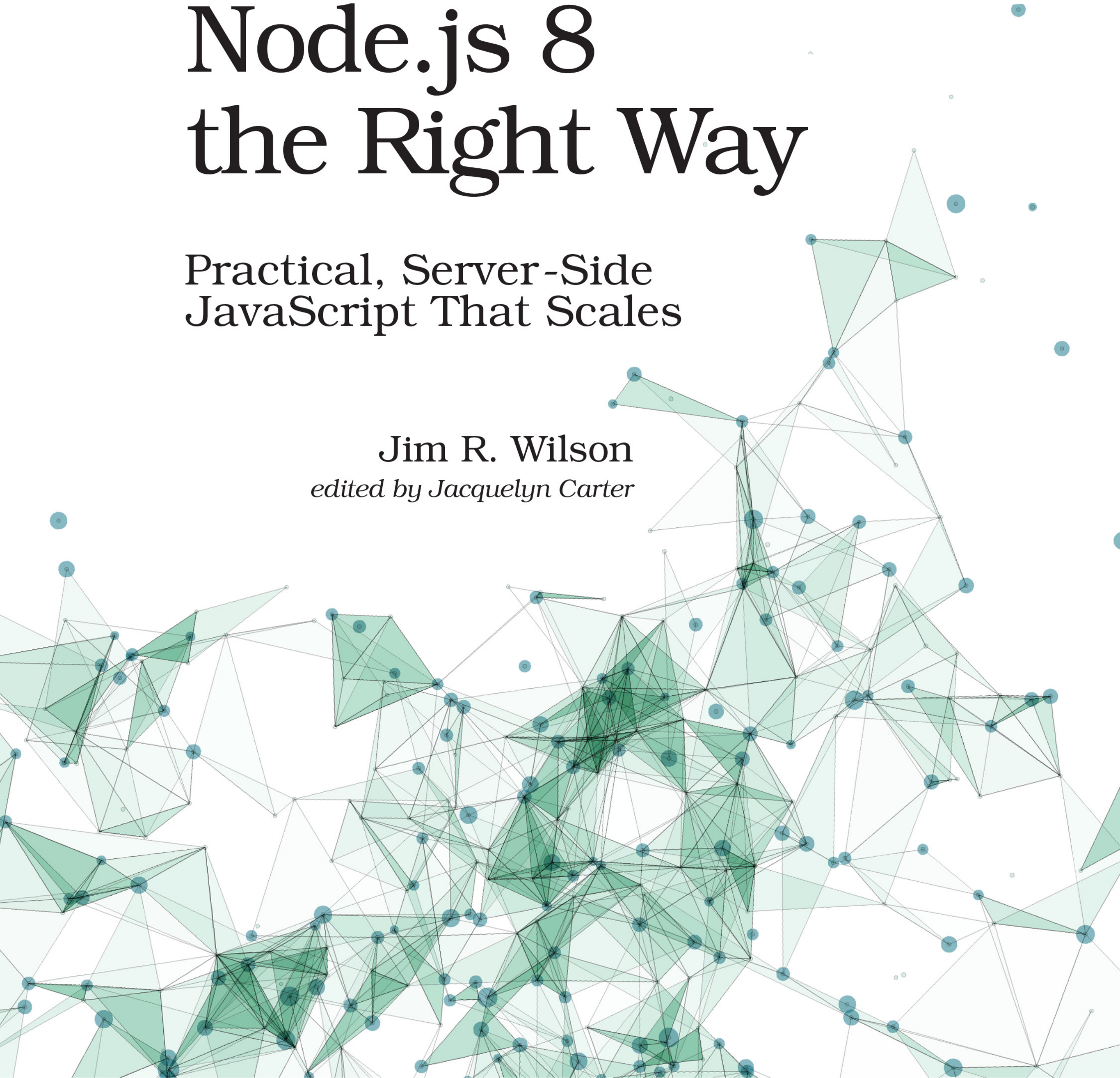
The Pragmatic Programmers

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

Pragmatic Bookshelf

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

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

Emulating Synchronous Style with async and await

One of the most powerful new features in Node.js 8 is the introduction of *async functions*. Part of the 2017 ECMAScript draft specification, async functions allow you reap the benefits of Promises for simplifying code flow while structuring your code in a more natural way.

The key is that unlike a regular function, which always runs to completion, an async function can be intentionally suspended midexecution to wait on the resolution of a Promise. Note that this does not violate the central maxim that JavaScript is single-threaded. It's not that some other code will preempt your async function, but rather that you choose to unblock the event loop to *await* a Promise.

An example should clarify. Consider this contrived function that returns a Promise.

```
const delay = (timeout = 0, success = true) => {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      if (success) {
        resolve(`RESOLVED after ${timeout} ms.`);
      } else {
        reject(`REJECTED after ${timeout} ms.`);
      }
    }, timeout);
  });
  return promise;
};
```

The delay() function takes two arguments, a timeout in milliseconds and a success Boolean value that indicates whether the returned Promise should be resolved (true) or rejected (false) after the specified amount of time. Using the delay() function is pretty straightforward—you call its .then() and .catch() methods to assign callback handlers. Here's an example:

```
const useDelay = () => {
  delay(500, true)
    .then(msg => console.log(msg))    // Logs "RESOLVED after 500 ms."
    .catch(err => console.log(err));  // Never called.
};
```

The useDelay() function invokes the delay() to get a Promise that's scheduled to be resolved after 500 milliseconds. Whether the Promise is resolved or rejected, the result is logged to the console.

Now, let's see what useDelay() would look like as an async function.

```
const useDelay = async () => {
  try {
    const msg = await delay(500, true);
    console.log(msg);  // Logs "RESOLVED after 500 ms."
  } catch (err) {
    console.log(err);  // Never called.
  }
};
```

First, notice the addition of the async keyword in the function declaration. This signals that the function can use the await keyword to suspend while resolving a Promise.

Next, check out the await keyword inside the try{} block, right before the call to delay(). Inside of an async function, await suspends execution until the Promise has been settled. If the Promise is resolved, then the await expression evaluates to the resolved value and the async function picks up where it left off.

On the other hand, if the Promise is rejected, then the rejection value gets thrown as an exception. In this case, we use the catch{} block to handle it.

Using async functions together with Promises presents a consistent, synchronous coding style for both synchronous and asynchronous operations. To practice, we'll use async functions for the remaining bundle APIs that we'll be adding.

Providing an Async Handler Function to Express

Open your lib/bundle.js and add the following inside the module.exports function, after the bundle-creation API you added previously.

web-services/b4/lib/bundle.js
```
/**
 * Retrieve a given bundle.
 * curl http://<host>:<port>/api/bundle/<id>
 */
app.get('/api/bundle/:id', async (req, res) => {
  const options = {
    url: `${url}/${req.params.id}`,
    json: true,
  };
  try {
    const esResBody = await rp(options);
    res.status(200).json(esResBody);
  } catch (esResErr) {
    res.status(esResErr.statusCode || 502).json(esResErr.error);
  }
});
```

This code block sets up a handler for the /bundle/:id route, which will allow us to retrieve a book bundle by its ID. Note the use of async before the function parameters in the opening line to indicate that we're using an async function. Inside the async function, like with other route handlers, the code proceeds in two parts: the setup of options and the request to Elasticsearch.

After setting up the options, we use a try/catch block to handle the success and failure modes of the Elasticsearch request. We issue the Elasticsearch request itself with the expression await rp(options). This causes the async function to suspend while waiting for the Promise to be settled.

If the Promise is resolved, then the return value of the await expression will be the Elasticsearch response body. In this case, we send it onward with a 200 OK status via the Express response object, res.

If the Promise is rejected, then the await expression throws the rejection value as an exception, which we catch and process. In this case, the rejection value is an object with rich information about the nature of the failure. We use that object's .statusCode and .error properties to close out the Express response.

Let's try this out using curl and jq. Open the terminal where you saved the BUNDLE_ID earlier and run the following command:

```
$ curl -s localhost:60702/api/bundle/$BUNDLE_ID | jq '.'
{
  "_index": "b4",
  "_type": "bundle",
  "_id": "AVuFkyXcpWVRyMBC8pgr",
  "_version": 1,
  "found": true,
  "_source": {
    "name": "light reading",
    "books": []
  }
}
```

The bundle object itself is in the _source property of this object. You can also try getting a bundle for a nonexistent ID to see what that returns.

```
$ curl -s localhost:60702/api/bundle/no-such-bundle | jq '.'
{
  "_index": "b4",
  "_type": "bundle",
  "_id": "no-such-bundle",
  "found": false
}
```

Back in your terminal that's running Node.js, you should see lines like the following:

```
GET /api/bundle/AVuFkyXcpWVRyMBC8pgr 200 60.512 ms - 133
GET /api/bundle/no-such-bundle 404 40.986 ms - 69
```

You should know one quick thing about the try/catch block before we move on. Consider this bad implementation that omits the try/catch block:

web-services/b4/lib/bundle.js
```
// BAD IMPLEMENTATION! async Express handler without a try/catch block.
app.get('/api/bundle/:id', async (req, res) => {
  const options = {
    url: `${url}/${req.params.id}`,
    json: true,
  };

  const esResBody = await rp(options);
  res.status(200).json(esResBody);
});
```

What would happen if the Promise returned by the rp() call was rejected instead of resolved? Do you have a guess?

Let's try it out. Comment out the try/catch lines from your async function, then save the file. Then try again to access a nonexistent bundle with curl using the verbose flag.

```
$ curl -v localhost:60702/api/bundle/no-such-bundle
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 60702 (#0)
> GET /api/bundle/no-such-bundle HTTP/1.1
> Host: localhost:60702
> User-Agent: curl/7.47.0
> Accept: */*
>
```

You should notice two things. First, the curl call never seems to terminate. It just hangs there after sending the request but receiving no response.

The second thing to notice is the warning messages in the Node.js terminal.

```
(node:16075) UnhandledPromiseRejectionWarning: Unhandled promise rejection
   (rejection id: 1): StatusCodeError: 404 - {"_index":"b4","_type":"bundle",
   "_id":"no-such-bundle","found":false}
(node:16075) DeprecationWarning: Unhandled promise rejections are deprecated.
    In the future, Promise rejections that are not handled will terminate the
    Node.js process with a nonzero exit code.
```

It turns out that indeed the await clause triggers the rejection object to be thrown, but since it's not caught inside the async function, it bubbles up to a Promise returned by the async function itself. That Promise is rejected, but since its rejection wasn't handled, we get warnings.

The moral of the story is always provide a try/catch block when using an async function as an Express route handler. More generally, it'll depend on the purpose of your async function, but as a rule of thumb you should consider the consequence of rejected Promises and take action accordingly.

Now let's move on to adding a few more APIs.

Setting the Bundle Name with PUT

Now we'll use an async function to implement an API endpoint that allows setting the name property of a book bundle.

Open your lib/bundle.js and add the following, after the GET bundle API.

web-services/b4/lib/bundle.js

```
/**
 * Set the specified bundle's name with the specified name.
 * curl -X PUT http://<host>:<port>/api/bundle/<id>/name/<name>
 */
app.put('/api/bundle/:id/name/:name', async (req, res) => {
  const bundleUrl = `${url}/${req.params.id}`;

  try {
    const bundle = (await rp({url: bundleUrl, json: true}))._source;

    bundle.name = req.params.name;

    const esResBody =
      await rp.put({url: bundleUrl, body: bundle, json: true});
    res.status(200).json(esResBody);

  } catch (esResErr) {
    res.status(esResErr.statusCode || 502).json(esResErr.error);
  }
});
```

Inside the async function, first we build out the bundleUrl based on the provided ID. Next, we begin the try/catch block in which we'll perform all of the Elasticsearch requests and response handling.

Take a look at the first line inside the try{} block. Here, we're using await with rp() to suspend as before, but it's a parenthesized expression. Outside of the expression, we use ._source to pull out just the bundle object from the broader Elasticsearch response. This demonstrates that the results of an awaited Promise can be used in more complex expressions.

Once we have the bundle object, we overwrite its name field with the provided name parameter value. Then it's time to PUT that object back into Elasticsearch with rp.put(). The resulting Elasticsearch response body should contain information about the successful operation, which we send back through the Express response.

As usual, if anything went wrong we catch the Elasticsearch response error and report back through the Express response. One you save the file, you can try it out.

In the same terminal where you have the BUNDLE_ID still saved as an environment variable, run the following to set the bundle name to *foo*:

```
$ curl -s -X PUT localhost:60702/api/bundle/$BUNDLE_ID/name/foo | jq '.'
{
  "_index": "b4",
  "_type": "bundle",
  "_id": "AVuFkyXcpWVRyMBC8pgr",
  "_version": 2,
  "result": "updated",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "created": false
}
```

You can confirm that it was indeed saved by retrieving the bundle using the GET bundle API.

```
$ curl -s localhost:60702/api/bundle/$BUNDLE_ID | jq '._source'
{
  "name": "foo",
  "books": []
}
```

Note that Express routes treat forward slashes as delimiters, so if you wanted to set the name of a bundle to *foo/bar*, you'd need to URI-encode the slash as *%2F*. The same goes for other special characters such as question marks and hash symbols.

Now let's move on to more complex route handlers. Next you'll learn how to manage concurrent unsettled Promises to make simultaneous asynchronous requests.