Contributing to Node.js Core

Jarrod Connolly

February 13, 2021

Abstract

My first contribution to Node.js core started from a missing piece in N-API, and this article is about finding a place to start in a large project. I cover the working groups, the review process, and what I learned from a welcoming community. The technical change matters less than understanding how a project of this size actually ships work.

Introduction

I had always wanted to contribute to a large open-source project like Node.js but found it daunting to find a place to start. The codebase is enormous. The governance model has working groups, steering committees, and a formal review process. I did not know where to begin.

One day while building an N-API native addon, I hit a gap. My addon needed to create and inspect JavaScript Date objects from C, but N-API had no Date support. The functionality simply was not there. I had found my opening.

This article is about what happened next: navigating the Node.js contribution process, writing the C code, working through review with some of the people I had admired from a distance, and watching a small change ship to millions of users. If you have ever wanted to contribute to a large project but felt unsure how to start, this is for you.

What is N-API?

N-API is a C API that lets developers write native addons for Node.js. It abstracts away the underlying JavaScript engine (V8), so addons compiled for one Node.js version continue to work on later versions without recompilation.

The Node.js documentation puts it well:

> N-API (pronounced N as in the letter, followed by API) is an API for building native Addons. It is independent from the underlying JavaScript runtime (for example, V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js.

There is also a C++ wrapper called `node-addon-api` that makes the API more ergonomic. The node-addon-examples repository is a great place to see N-API in action.

At the time, N-API supported strings, numbers, objects, arrays, buffers, promises, and typed arrays. But it had no way to work with JavaScript Dates. If you needed to pass a timestamp from C into JavaScript, you were out of luck.

Preparation

Node.js is a large project with a formal contribution process, and I expected friction. What I found surprised me: the project's documentation for contributors is thorough, clear, and genuinely welcoming.

The pull-requests guide walked me through setting up a local development environment, building Node.js from source, and running the test suite. It also explained the commit message format, a convention I had seen in the git log but never understood until I read the rationale behind it. I returned to this document repeatedly throughout the process.

The C++ style guide and writing tests guide gave me confidence that I was writing code that would pass review. The collaborator guide was particularly helpful: it explains how reviewers approach a pull request, what labels mean, and how the CI system works. Reading it from the reviewer's perspective changed how I thought about my own submission.

I forked the repository, got a local build compiling, ran the test suite to establish a baseline, and started coding. The documentation made what could have been an intimidating setup feel methodical.

Coding

My goal was straightforward: add Date support to N-API. I needed three functions to round out the feature.

`napi_create_date` allocates a JavaScript Date object from a C `double` representing milliseconds since the Unix epoch. This was the main entry point. Here is the core of the implementation:

```
napi_status napi_create_date(napi_env env,
                             double time,
                             napi_value* result) {
  NAPI_PREAMBLE(env);
  CHECK_ARG(env, result);

  v8::Local<v8::Value> value =
      v8::Date::New(env->context(), time).ToLocalChecked();
  *result = v8impl::JsValueFromV8LocalValue(value);

  return napi_clear_last_error(env);
}
```

`napi_is_date` checks whether a given `napi_value` is a Date object. This lets addon authors validate arguments before operating on them.

`napi_get_date_value` retrieves the underlying time value from a Date, allowing C code to read timestamps passed from JavaScript.

I also had to add a new error code (`napi_date_expected`) to the status enum, update the N-API version to 4, and write documentation for all three functions. The documentation followed the same pattern as every other N-API function: a YAML metadata block with the version it was added, the function signature, parameter descriptions, and a one-line summary.

The test suite covered both C and JavaScript. The C test called each function directly through the N-API bindings. The JavaScript test created a Date, passed it through the native addon, and asserted the round-trip was correct:

```
const test_date = require(`./build/${common.buildType}/test_date`);

const dateTypeTestDate = test_date.createDate(1549183350);
assert.strictEqual(test_date.isDate(dateTypeTestDate), true);

const dateValue = test_date.getValue(dateTypeTestDate);
assert.strictEqual(dateValue, 1549183350);
```

I learned a lot by reading existing N-API implementations. The code for `napi_create_promise` and `napi_is_promise` showed me the pattern: a `NAPI_PREAMBLE` macro for setup, a `CHECK_ARG` for validation, the V8 API call, and a `napi_clear_last_error` to wrap up. Following existing conventions meant fewer surprises during review.

In total, the change touched 7 files and added 227 lines.

Pull Request

I opened pull request #25917 on February 4, 2019. Four people reviewed it: Anna Henningsen, Michael Dawson, Colin Ihrig, and James M Snell. These were names I recognized from Node.js release notes and conference talks. Having them review my code was intimidating and exciting in equal measure.

The review was thorough but never harsh. Each comment was specific, actionable, and framed as a suggestion rather than a demand. One exchange stuck with me. Anna noticed that my test function names used lowercase while other files in the codebase used PascalCase. I replied with examples showing the inconsistency existed across the codebase: some functions used `createError`, others used `CreateTypedArray`. Michael chimed in that N-API had no firm convention either way. Anna agreed it was minor and not blocking. I updated the names to match the dominant style anyway.

That exchange taught me something about how healthy open-source projects operate. Reviewers surface observations to spark discussion, not to issue rulings. Anna raised a point, I offered context, Michael added his perspective, and we reached a quick consensus. Nobody dug in. Nobody had to be right. The conversation itself improved the code, and the tone made me want to participate more, not less.

The CI system caught an unrelated timeout on a SmartOS test runner. A Python bug in the test harness (`'list' object has no attribute 'splitlines'`) crashed the runner, and the build hung until the 15-minute timeout. I read the log, identified that it was not my code, and noted it in the thread. Diagnosing CI failures is a skill I did not expect to practice on my first PR.

After the review rounds were complete and CI was green, there was a pause. On February 25, I wrote:

> Just pinging to see if it would be possible to land this PR. My first and I am looking to complete the cycle once before moving on to future PRs.

Michael replied that a security release had locked the CI. He asked me to remind them after the release went out. A few days later, on February 28, he ran a lite CI to confirm nothing had changed and landed the commit as `13b1aaf`. Twenty-four days from PR to merge.

The change shipped in Node.js v11.11.0 on March 5, 2019, and became part of N-API version 4. Months later, Gabriel Schulhof backported it to the v10.17.0 LTS release. I did not have to do anything for the backport. Someone else saw the value and did the work. That is the quiet multiplier of contributing to a well-maintained project.

Experience

I started this process intimidated by the size of Node.js and the reputation of its maintainers. I finished it with code running in every Node.js installation shipped after March 2019. The gap between those two states was filled by documentation that told me exactly what to do and reviewers who treated my code with respect.

The people I had looked up to (Anna, Michael, Colin, James) were not gatekeepers. They were guides. Later that year at the Node+JS Interactive conference, I had the chance to thank several of them in person. The warmth online extended to the conference hallway.

If you have been hesitating to contribute to an open-source project, start by looking for a gap. Not a large feature. Not a refactor. A small, concrete piece of missing functionality that you genuinely need. Read the project's contribution documentation carefully. It exists because the maintainers want you to succeed. Write tests. Be gracious in review. And when your PR sits idle for a few days, a polite ping goes a long way.

The barrier to contributing to large projects is lower than it looks. The people on the other side of the pull request want your code to land as much as you do.