Node.js medium account (list out the authors from the team)

Title: Building Modern Native Add-ons for Node.js in 2020

Authors: Name/Twitter

Chengzhong Wu (@legendecas)
Gabriel Schulhof (@gabrielschulhof)
Jim Schlight (@jimschlight)
Kevin Eady
Michael Dawson (@mhdawson1)
Nicola Del Gobbo (@NickNaso)

Introduction

N-API provides an ABI-stable API that can be used to develop native add-ons for Node.js, simplifying the task of building and supporting such add-ons across Node.js versions.

With downloads of [node-addon-api](https://www.npmjs.com/package/node-addon-api) surpassing 2.5 million per week, all LTS versions of Node.js supporting N-API version 3 or [higher](https://nodejs.org/dist/latest-v15.x/docs/api/n-api.html#n_api_n_api_version_matrix) and node.js 15.x being released with support for N-API 7, it is a good time to take a look at the progress on simplifying native add-on development for Node.js.

When we started working on N-API back in 2016 (original [proposal](https://github.com/nodejs/node-eps/blob/master/005-ABI-Stable-Module-API.md) is 12 Dec 2016) we knew it was going to be a long journey. There are many native packages in the ecosystem and we understood the transition would take quite some time.

The good news is that we have come a long way since the initial proposal. There has been a lot of work by the Node.js collaborators and the team focussed on N-API as well as package authors who have moved over. In that time, N-API has become the [default](https://nodejs.org/api/addons.html#addons_c_addons) recommendation for how to build native add-ons.

While the basic design has remained consistent (as planned), we've added incremental features in each new N-API version in order to address feedback from package authors as they adopted N-API and node-addon-api.

It's also been great to see the positive feedback from package authors along the way. For example: https://twitter.com/mafintosh/status/1256180505210433541

Having said that, let's dive into some of the new features/functions that have been added over the last few years.

New features/functions

As people have been using N-API and node-addon-api we've been adding the key features that have been needed, including generally improving the add-on experience.

N-API 4	Thread-safe Functions
N-API 5	Date objects Attach finalizer to object
N-API 6	BigInts Set Instance Data Get All Property Names
N-API 7	Detached ArrayBuffers

The changes fall into 3 main categories which are covered in the sections which follow.

Multi-Threaded and Asynchronous Programming

As Node.js becomes more prominent in the computing world, the need to interact with native OS-level asynchronous activities has grown. Node.js is a single-threaded implementation of the JavaScript language, where only the main thread may interact with JavaScript values.

Performing computationally-intensive tasks on the main thread will block program execution, queuing events and callbacks in the event loop. As we gained experience with real-world use, in order to facilitate program integrity across multiple threads, both N-API and its

wrapper node-addon-api were updated to provide several mechanisms to call into the Node.js thread from outside the main event loop, depending on use-case:

- **AsyncWorker**: provides a mechanism to perform a one-shot action, and notify Node.js of its eventual completion or failure.

- **AsyncProgressWorker**: similar to the above, adding the ability to provide progress updates for the asynchronous action.

- **Thread-safe functions**: provides a mechanism to call into Node.js at any time from any number of threads.

Context-sensitivity

Another recent Node.js development is the arrival of [workers](https://nodejs.org/docs/latest/api/worker_threads.html). These are full-fledged Node.js environments running in threads parallel to the Node.js main thread. This means that native add-ons can now be loaded and unloaded multiple times as the main process creates and destroys worker threads.

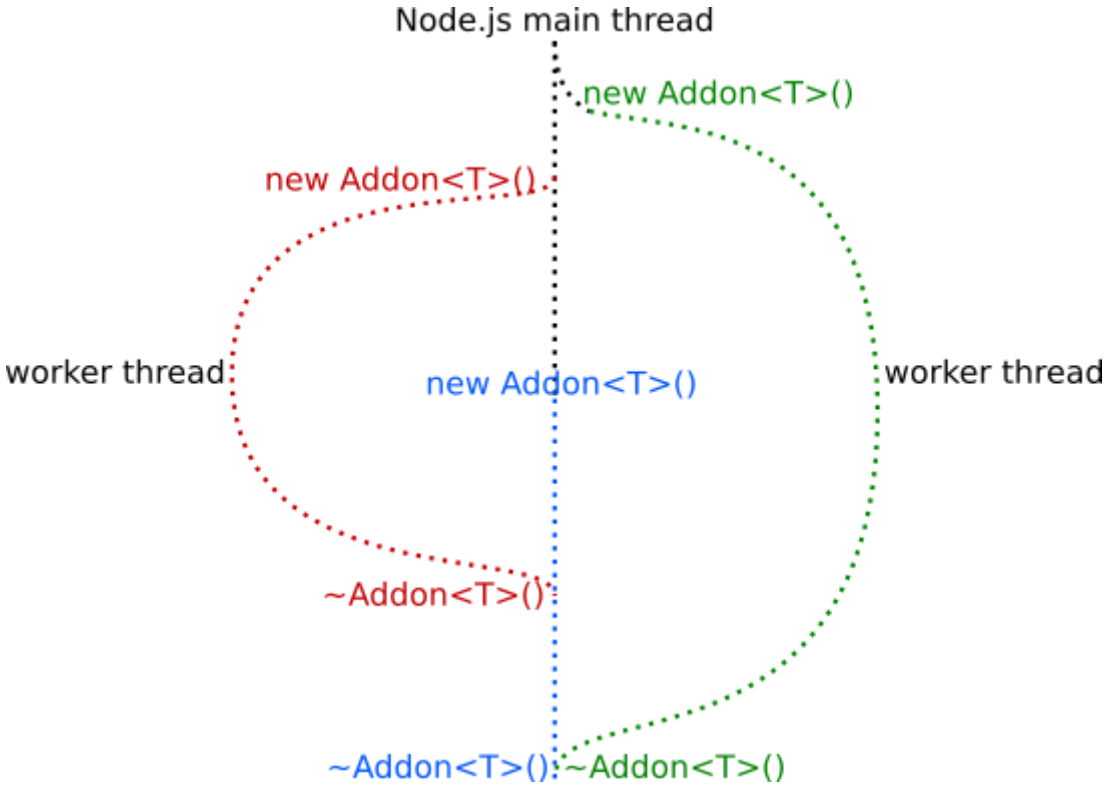
Since threads share the same memory space as the main process, multiple copies of a native add-on must now be able to co-exist in a single process. On the other hand, the library containing a native add-on is only loaded once per process. Thus, global data stored by a native add-on, which was so far stored in global variables, must no longer be stored in such a way, because global storage is not thread-safe.

Static data members of C++ classes are also stored in a thread-unsafe manner, so those must also be avoided. It's also important to remember that the thread is not necessarily that which makes an add-on instance unique. Thus, thread-local storage of global variables should also be avoided.

In N-API version 6 we started providing a space for storing per-instance global data by introducing the concept of add-on instances, multiple of which can co-exist in a process, and by providing some tools for creating self-contained add-ons, such as

- the [`NAPI_MODULE_INIT()`](https://nodejs.org/api/all.html) macro, which will initialize an add-on in such a way that it can then be loaded multiple times during the life cycle of the Node.js process.
- [`napi_get_instance_data()`](https://nodejs.org/api/n-api.html#n_api_environment_life_cycle_apis) and [`napi_set_instance_data()`](https://nodejs.org/api/n-api.html#n_api_environment_life_cycle_apis) in order to provide a place for safely storing global data associated with a single instance of an add-on.
- The node-addon-api [`Addon<T>`](https://github.com/nodejs/node-addon-api/blob/3.0.2/doc/addon.md#example) class, which neatly combines the above tools to create a class whose instances represent instances of an add-on present in the various worker threads

created by Node.js. Thus, add-on maintainers can store per-add-on-instance data as variables in an instance of the `Addon<T>` class and Node.js will create an instance of the `Addon<T>` class whenever it is needed on a new thread:



Additional helper methods

As package maintainers used N-API we discovered a few additional APIs that were commonly needed. These included:

- Date objects
- BigInts
- Retrieving property names from objects
- Detaching ArrayBuffers

Building

One of the other main areas where the N-API team worked to fill in gaps and make it easier for maintainers to consume N-API was the build workflow, including additions to CMake.js, node-pre-gyp and prebuild.

Historically, Node.js native addons have been built using node-gyp. For source code libraries that are already being built using CMake, the CMake.js build tool is an attractive alternative

for building Node.js native add-ons. We have recently added an example of an add-on built using CMake.

Detailed information about using CMake.js with N-API add-ons can be found on the N-API Resource.

One of the realities of developing Node.js native add-ons is the fact that as part of installing the package using `npm install` the C or C++ code must be compiled and linked. This compilation step requires that a viable C/C++ toolchain be installed on the system doing the compilation. This can present a barrier to the adoption of native add-ons as the user of the add-on may not have the necessary tools installed. This can be addressed by creating prebuilt binaries that can be downloaded by the user of the native add-on.

A number of build tools can be used to create prebuilt binaries. node-pre-gyp builds binaries that are typically uploaded to AWS S3. prebuild is similar to node-pre-gyp but uploads the binaries to a GitHub release.

prebuildify is another option similar to the above that enables the native add-on developer to bundle the prebuilt binaries into the module uploaded to npm. The advantage of this approach is that the binaries are immediately available to the user when the package is downloaded. Although the downloaded npm package is larger in size, in practice the entire download process is faster for the user because secondary download requests to AWS S3 or a GitHub release are unnecessary.

Resources for getting started

One resource available to help get started is the node-addon-examples GitHub repository, containing samples of various Node.js native add-ons. The root of the repository contains folders for different functional aspects, from a simple Hello World add-on to a more complex multi-threaded add-on. Each example folder contains up to three subfolders: one for each Node.js add-on implementation (legacy NAN, N-API, and node-addon-api). To get started with the Hello World example using the node-addon-api implementation, simply run:

```
git clone https://github.com/nodejs/node-addon-examples.git
cd node-addon-examples/1_hello_world/node-addon-api/
npm i
node .
```

Another resource available is the The N-API Resource. This website contains information and additional in-depth walkthroughs regarding building Node.js add-ons and other advanced topics, such as:

- tools needed to get started
- migration guide from NAN
- differences between build systems (node-gyp, cmake, ...)
- context-sensitivity and thread-safety

Closing out and call to action

Since the earliest days Node.js supported the ability to add features written in native code (C / C++) and to expose them through a JavaScript interface. Over time we recognized that there were challenges in implementing, maintaining, and distributing the resulting addons. N-API was identified as one of the core areas for improvements requested by module owners in order to address those challenges. The whole team and the community began to contribute to the creation of this new API in the core.

The resulting C API is now a part of every Node.js distribution and a C++ convenience wrapper called node-addon-api is distributed as an external package through npm. N-API was launched with the promise to guarantee the API and the ABI compatibility across different major versions of Node.js and this has introduced a series of benefits:

- It has removed the need to recompile modules when migrating to newer major versions of Node.js
- It allows JavaScript engines other than V8 to implement N-API which, in turn, allows add-on maintainers to target different runtimes (such as Babylon Native or IoT.js, and Electron) with the same code they use for supporting Node.js.
- Since N-API is a C API it is possible to implement native add-ons using languages other than C / C++ (such as Go or Rust).

When N-API has been released as an experimental API in Node.js v8.0.0 its adoption started to grow slowly, but many developers started to send feedback and contributions and this led us to add new features and to create new tools to better support all the native add-ons ecosystem.

Today N-API is widely used for the development of native add-ons. Some of the most used native add-ons have been ported to N-API:

- sharp (~900k per week)
- bcrypt (~500k per week)
- sqlite3 (~300k per week)

In the last few years many improvements happened for N-API and for native add-ons in general that bring the users' and maintainers' experience with native add-ons almost up to par with JavaScript modules.

Get Involved

We are constantly making progress on N-API and in general on the native add-ons ecosystem, but we always need more help. You could help us and the whole community to continue improving N-API in many ways:

- Porting your own native module to use N-API
- Porting a native module that your app depends on to N-API
- Adding new features to N-API
- Adding new features from N-API to node-addon-api
- Fixing or adding test cases for node-addon-api

- Fixing or adding examples to node-addon-examples

If you are interested in joining us, see details in https://github.com/nodejs/abi-stable-node#meeting on how to join our weekly meeting.