accusoft®

Next Generation Native Node: N-API in C++

Featured

Steve Brooks
Sr. Software Engineer

Intro to Node JS

Some time ago, way back in the Node JS 0.10.x days, we needed to expose some native libraries previously written in C++ for use by an existing Node JS based service.

There were two reasons to reuse this existing C++ code base:

1. The code was already written and debugged so wrapping it made a lot of sense from a cost perspective.
2. The intellectual property was somewhat easier to protect in a binary than it was in a JavaScript module.

Fortunately, support for native modules was already available at the time, and we took advantage of them. Over the lifetime of that project, we ended up with a few breaking changes due to the instability of the Node.js Addon API through it's early releases. Now, 5 years later, the planned backward compatibility support for the newer Node Native API sounds pretty exciting.

Below is a brief history of the Native Node APIs and a look at the current new standards for native node development as of Node.js v10.

History

Node.js introduced Node.js Addons prior to v0.1.14 and were simply native node modules that could be required and used by any other JavaScript node module. The Addon API and build chain changed several times through the early years of Node.js which resulted in number of breaking changes between releases.

Addons

A simple “hello world” example (hello.cc) from the Node.js 0.7.4 release is shown below:

```
// hello.cc
#include <node.h>
#include <v8.h>

v8::Handle<v8::Value> Method(const v8::Arguments& args) {
  v8::HandleScope scope;
  return scope.Close(v8::String::New(“hello, world”));
}

void init(v8::Handle<v8::Object> target) {
  target->Set(v8::String::NewSymbol(“hello”),
      v8::FunctionTemplate::New(Method)->GetFunction());
}
NODE_MODULE(hello, init)
```

The code above is essentially a native C++ implementation of the JavaScript module below

```
function Method() { return ‘hello, world’; };

exports.hello = Method;
```

The hello.cc module merely defines the behavior for the module. The build process for the module was a separate concern.

Early versions of the build chain used python-based WAF which required the definition of a wscript file to define the build for the Addon.

```
srcdir = ‘.’
blddir = ‘build’
VERSION = ‘0.0.1’

def set_options(opt):
  opt.tool_options(‘compiler_cxx’)

def configure(conf):
```

```
def build(bld):
  obj = bld.new_task_gen('cxx', 'shlib', 'node_addon')
  obj.target = 'hello'
  obj.source = 'hello.cc'
```

The module could then be built from the command line with

```
node-waf configure build
```

which would create the native addon module under ./build/Release/hello and could be required for use with:

```
var addon = require('./build/Release/hello');
console.log(addon.hello()); // 'hello, world'
```

Fortunately soon after, the WAF build process was replaced with Google GYP in Node.js 0.8.16 so that the build definition became much more succinct with the binding-gyp file:

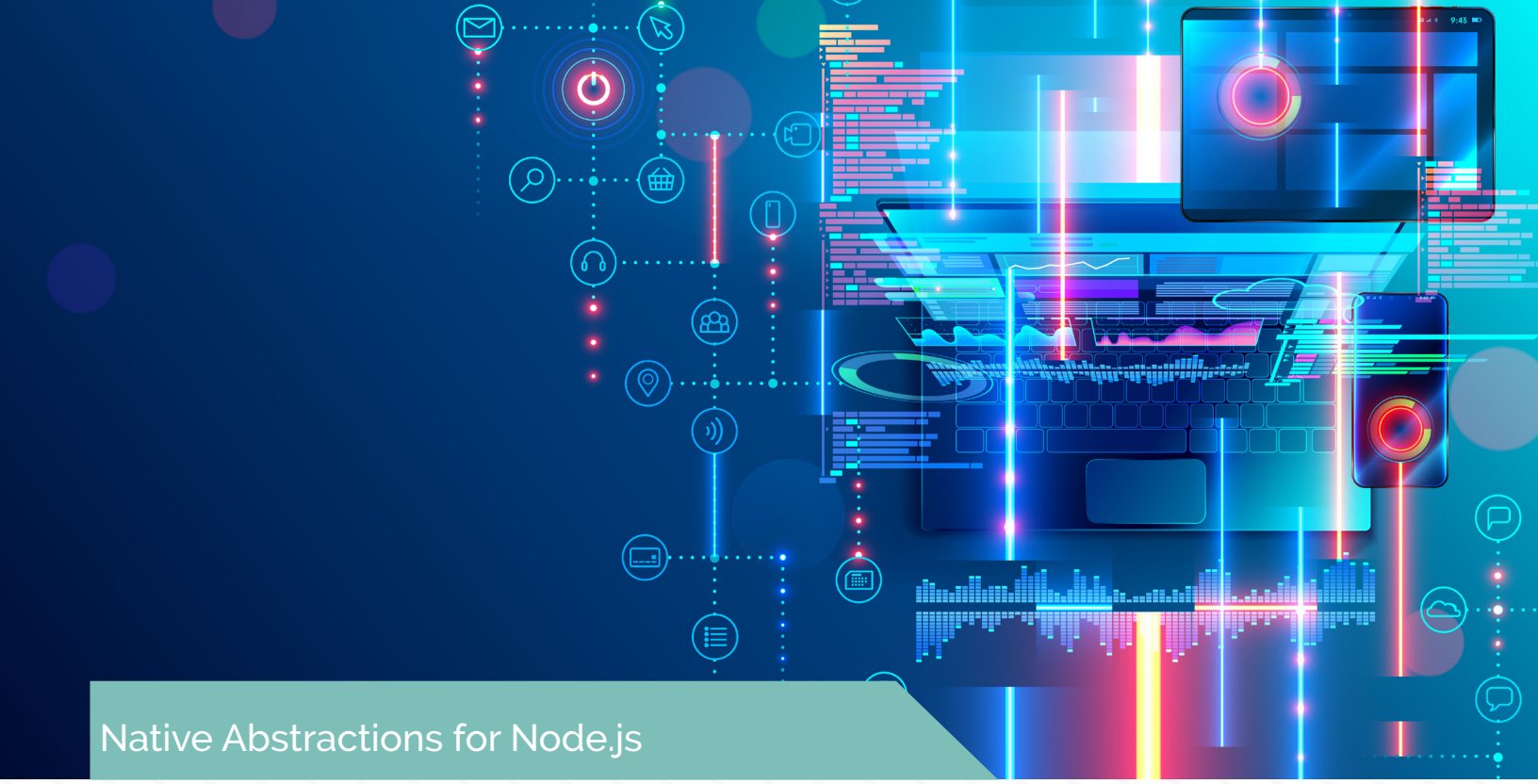
```
{
  "targets": [{
    "target_name": "hello",
    "sources": [ "hello.cc" ]
  }]
}
```

This could then be built with:

```
node-gyp build
```

It would result in the same release build as above.

Sadly, Io.js 1.0 introduced the first of several breaking changes to node Addons which gave developers the impetus to move to an abstraction layer: Native Abstractions for Node.js (NAN).



Native Abstractions for Node.js

Early on, the need for an abstraction to protect developers against breaking changes became apparent. Rod Vagg created the Native Abstractions for Node.js to address that need.

By Io.js 1.0, the Addon Hello World example had morphed to the example below.

```
// hello.cc
#include <node.h>

void Method(const v8::FunctionCallbackInfo<v8::Value>& args) {
  v8::Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(v8::String::NewFromUtf8(isolate, “hello,
world”));
}

void init(v8::Local<Object> exports) {
  NODE_SET_METHOD(exports, “hello”, Method);
}
NODE_MODULE(addon, init)
```

Because of these changes, Addons created prior to this exhibited in errors during compilation. The Native Abstractions for Node.js (NAN) wrapped the underlying Addon API so that small changes to the underlying Addon API would not break user code for future releases of Node.

An early example of the NAN API version of the same hello world app is shown below.

```
// hello.cc
#include <node.h>
#include <nan.h>

NAN_METHOD(Method) {
  NanScope();
  NanReturnValue(String::New("hello, world"));
}

void InitAll(Handle<Object> exports) {
  exports->Set(NanSymbol("hello"),
    FunctionTemplate::New(Method)->GetFunction());
}
NODE_MODULE(addon, InitAll)
```

The associated binding-gyp file shows little change from the Node Addon version.

```
{
  "targets": [{
    "target_name": "hello",
    "sources": [ "hello.cc" ]
  }],
  "include_dirs": ["<!(node -e \"require('nan')\")"]
}
```

The only difference is the addition of the include_dirs property that includes the dependent NAN header files.

While NAN does provide a less verbose API and abstraction from the Node Addon API, it still presented breaking changes when upgrading from early versions of Node.js. In its defence, NAN has become much more stable as a result of the Addon API's increased stability in later versions. NAN is still very much a viable alternative if you are planning to support Node.js v4 and later. For more modern versions of Node.js (e.g. v10 and later) you should consider using the newer N-API.

N-API

Like the original Node.js Addon, the new N-API is delivered as a part of the Node.js distribution. Starting with Node.js v8, it was delivered as an experimental feature and has become stable in the Node.js v10 release. The real benefit of N-API is that it is a true abstraction and is being developed to be forward compatible. For example, modules created with N-API v3 in Node.js v10 are guaranteed to be forward compatible with future versions of Node.js which should result in no rework as projects upgrade Node.js versions. The N-API / Node.js version matrix is shown below.

	N-API Version				
Node.js	**1**	**2**	**3**	**4**	**5**
v8.x	v8.0.0	v8.10.0	v8.11.2	v8.16.0	
v9.x	v9.0.0	v9.3.0	v9.11.0		
v10.x			v10.0.0	v10.16.0	
v11.x			v.11.0.0	v.11.8.0	
v12.x				v12.0.0	v12.11.0
v13.x					v13.0.0

C versus C++

Strictly speaking, N-API is implemented in C. While there is nothing preventing us from the C implementation, there is a C++ header only implementation that wraps the C implementation and is much easier to use. Additionally, if N-API adds new features and the C++ wrapper does not immediately wrap those features, but you still want to use them, there is nothing to prevent you from using the C implementation along with the C++ one in the same module. For the remainder of this article, we'll be using the C++ wrapper.

So, what can I do with it?

Let's look at a couple of simple examples and then move on to something more complex and useful.

N-API can still use the node-gyp build system but additionally, it can now use CMake which is likely more familiar to C/C++ developers. If you are interested in using CMake, see the cmake-js documentation.

First, let's look at the N-API version of the hello, world example. The module itself is not a big departure from the other versions:

```
// hello.cc
 #include <napi.h>

Napi::String Method(const Napi::CallbackInfo& info) {
  Napi::Env env = info.Env();
  return Napi::String::New(env, “hello, world”);
}

Napi::Object Init(Napi::Env env, Napi::Object exports) {
  exports.Set(Napi::String::New(env, “hello”), Napi::Function::New(env,
Method));
  return exports;
}
NODE_API_MODULE(hello, Init)
```

There is still a function definition (Method) which provides the implementation, and an Init function which defines the module exports. There is really nothing else of interest here.

The binding-gyp is slightly more complex than the original implementation:

```
{
  “targets”: [{
    “target_name”: “hello”,
    “sources”: [ “hello.cc” ],
    “include_dirs”: [“<!@(node -p \”require(‘node-addon-api’).
include\”)”],
    “cflags!”: [ “-fno-exceptions” ],
    “cflags_cc!”: [ “-fno-exceptions” ],
    “defines”: [ ‘NAPI_DISABLE_CPP_EXCEPTIONS’ ]
  }]
}
```

Like the previous versions, it requires the target name, source, and include_dirs, but it also adds compiler definitions to disable the C++ exception mechanism.

Node Addons, NAN, and N-API all support the features described below, but since this is a post about N-API, I'll only be describing the N-API ones.

Function Arguments

Non-trivial functions require data on which to operate. In Node.js, as in other interoperability environments, data must be marshalled from the managed JavaScript environment to the native C++ module for use there. N-API provides helpers to do that. Consider a slight modification to our hello world example in which a name string is passed for the target of the hello.

```
// hello.cc - With a parameter
#include <sstream>
#include <napi.h>

Napi::Value Method(const Napi::CallbackInfo& info) {
  Napi::Env env = info.Env();

  std::string arg0 = info[0].As<Napi::String>().Utf8Value();

  std::stringstream str;
  str << "hello, " << arg0;

  return Napi::String::New(env, str.str());
}

Napi::Object Init(Napi::Env env, Napi::Object exports) {
  exports.Set(Napi::String::New(env, "hello"), Napi::Function::New(env,
Method));
  return exports;
}

NODE_API_MODULE(addon, Init)
```

In this example, we get the first argument passed to the function and attempt to convert it to a Napi::String. Once that is available, we convert it to a UTF8 string and then use std::stringstream to format the output in order to return "hello, n-api" when the function is called with the string parameter "n-api".

Callbacks

Most functions in node are not simple functions that just return a result, so in this section we'll implement a callback that will return results through a function call.

```
// hello.cc - With callback
#include <napi.h>

void RunCallback(const Napi::CallbackInfo& info) {
  Napi::Env env = info.Env();

  Napi::Function cb = info[0].As<Napi::Function>();
  cb.Call(env.Global(), {Napi::String::New(env, “hello world”)});
}

Napi::Object Init(Napi::Env env, Napi::Object exports) {
  return Napi::Function::New(env, RunCallback);
}
NODE_API_MODULE(addon, Init)
```

The RunCallback function in this case retrieves the callback function pointer from the arguments passed to it by the caller in the same way that the string argument was retrieved in the last example. Before the function completes, it executes the callback function passing the “hello world” string as the first parameter. The client code looks as you might expect:

```
const { hello } = require(‘bindings’)(‘addon’);

hello((msg) => {
  console.log(msg); // “hello world”
});
```

Unfortunately, this implementation does not provide actual asynchronous execution. In this case, the function call blocks until after it completes the callback. For true asynchronous execution, we require a somewhat more complex implementation.

Actual Asynchrony

The AsyncWorker virtual class provides a nice wrapper to do asynchronous work and then hook the completion of that work in order to make a callback to the caller. In this example, we’ll take it one step further and return a Promise to the caller which will later be resolved or rejected depending on the outcome of the asynchronous work.

```
// hello.cc
#include <sstream>
#include <napi.h>

class AsyncWorker : public Napi::AsyncWorker {
public:
  static Napi::Value Create(const Napi::CallbackInfo& info) {

    if (info.Length() != 1) {
      return Reject(info.Env(), “MissingArgument”);
    } else if (!info[0].IsString()) {
      return Reject(info.Env(), “InvalidArgument”);
    }

    std::string input = info[0].As<Napi::String>().Utf8Value();
    AsyncWorker* worker = new AsyncWorker(info.Env(), input);
    worker->Queue();
    return worker->deferredPromise.Promise();
  }

protected:
  static Napi::Value Reject(Napi::Env env, const char* msg) {
    Napi::Promise::Deferred failed = Napi::Promise::Deferred::New(env);
    failed.Reject(Napi::Error::New(env, msg).Value());
    return failed.Promise();
  }

  void Execute() override {
    if(input.size() < 1) {
      SetError(“EmptyName”);
      return;
    }

     std::stringstream str;
    str << “hello, “ << input;

    result = str.str();
  }

  virtual void OnOK() override {
      deferredPromise.Resolve(Napi::String::New(Env(), result));
  }
```

```
  virtual void OnError(const Napi::Error& e) override {
      deferredPromise.Reject(e.Value());
  }

private:
  AsyncWorker(napi_env env, std::string& name) :
    Napi::AsyncWorker(env),
    input(name),
    result(),
    deferredPromise(Napi::Promise::Deferred::New(env)) { }

  std::string input;
  std::string result;

  Napi::Promise::Deferred deferredPromise;
};

Napi::Object Init(Napi::Env env, Napi::Object exports) {
  return Napi::Function::New(env, AsyncWorker::Create);
}
NODE_API_MODULE(addon, Init)
```

This may be a lot to take in, so let's break it down into pieces.

We've created an AsyncWorker class that inherits from Napi::AsyncWorker so that it gets all the base asynchronous implementation. It also provides an abstract method, Execute, that we are required to implement which will be doing the asynchronous work. We'll revisit that soon.

```
class AsyncWorker : public Napi::AsyncWorker {
...
protected:
  void Execute() override {
  ...
  }
...
};
```

We've provided a static factory method, Create, which is responsible for validating and unpacking the arguments, queuing the asynchronous worker, and returning a promise that will resolve or reject when the work is completed.

```
static Napi::Value Create(const Napi::CallbackInfo& info) {

  if (info.Length() != 1) {
    return Reject(info.Env(), “MissingArgument”);
  } else if (!info[0].IsString()) {
    return Reject(info.Env(), “InvalidArgument”);
  }

  std::string input = info[0].As<Napi::String>().Utf8Value();
  AsyncWorker* worker = new AsyncWorker(info.Env(), input);
  worker->Queue();
  return worker->deferredPromise.Promise();
}
```

Note that here, we validate the arguments and return a rejected Promise if the argument is missing or is not a string. We use the protected static helper method, Reject, to do this.

```
static Napi::Value Reject(Napi::Env env, const char* msg) {
  Napi::Promise::Deferred failed = Napi::Promise::Deferred::New(env);
  failed.Reject(Napi::Error::New(env, msg).Value());
  return failed.Promise();
}
```

Here we construct a new Deferred and then reject it with a new Error value before finally returning the rejected Promise.

Once the argument is validated, we marshall it from a Napi::String to a std::string so we can operate on it using the C++ standard libraries.

We then create an instance of our AsyncWorker class and pass the environment and the input string to the constructor to initialize them. We then call the inherited Queue method to place our work into the event queue for eventual execution and return a promise to be resolved later.

The Execute method is actually trivial:

```
void Execute() override {
  if(input.size() < 1) {
    SetError(“EmptyName”);
    return;
  }

  std::stringstream str;
  str << “hello, “ << input;

  result = str.str();
}
```

Here we get the input via the class's input member which we set at construction. We could have validated that the name was not empty in the static Create method, but doing this here allows us to illustrate the SetError method which can be used to indicate that an error occurred during the Execute function.

If the string is not empty, then we use the standard library stringstream to format a new string and assign it to the class's result member.

Once the Execute method succeeds or fails, the Napi::AsyncWorker base class will call either the OnOK or OnError virtual methods:

```
  virtual void OnOK() override {
      deferredPromise.Resolve(Napi::String::New(Env(), result));
  }

  virtual void OnError(const Napi::Error& e) override {
      deferredPromise.Reject(e.Value());
  }
```

In the success case, we simply resolve the deferred Promise with the result member. Note that we marshal the std::string to a new Napi::String to be returned in the resolve arguments.

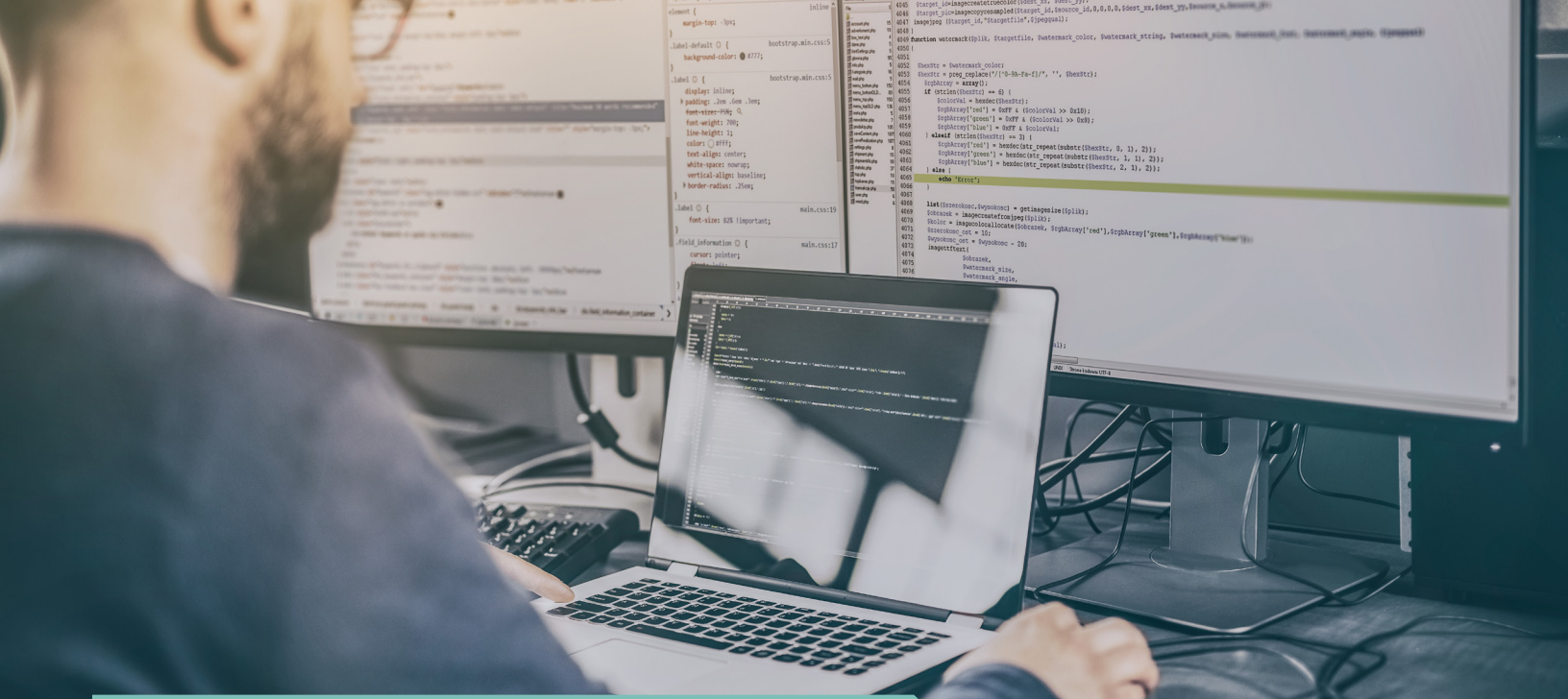
In the failure case, we reject the deferred promise with any error we set in the Execute method. In this case specifically the EmptyName error.

Lastly, the Init is just like all of the other examples, except in this case we pass the static AsyncWorker::Create method as the sole exported function.

The client code for this looks exactly like you might expect:

```
const helloWorker = require('bindings')('addon');

(async ()=> {
  try {
    const result = await helloWorker('world');
    console.log(result); // 'hello, world'
  } catch(err) {
    console.error(err, 'Failure in helloWorker');
    throw err;
  }
})();
```



Wrapping Up

While the C-based Node.js N-API is the future of native node development, the node-addon-api C++ wrapper provides a fairly succinct and intuitive interface developers can use to create native modules for a variety of applications. Best of all, the code written for Node.js v10 is expected to work (without recompiling) for v11, v12, v13, and beyond with no code changes.