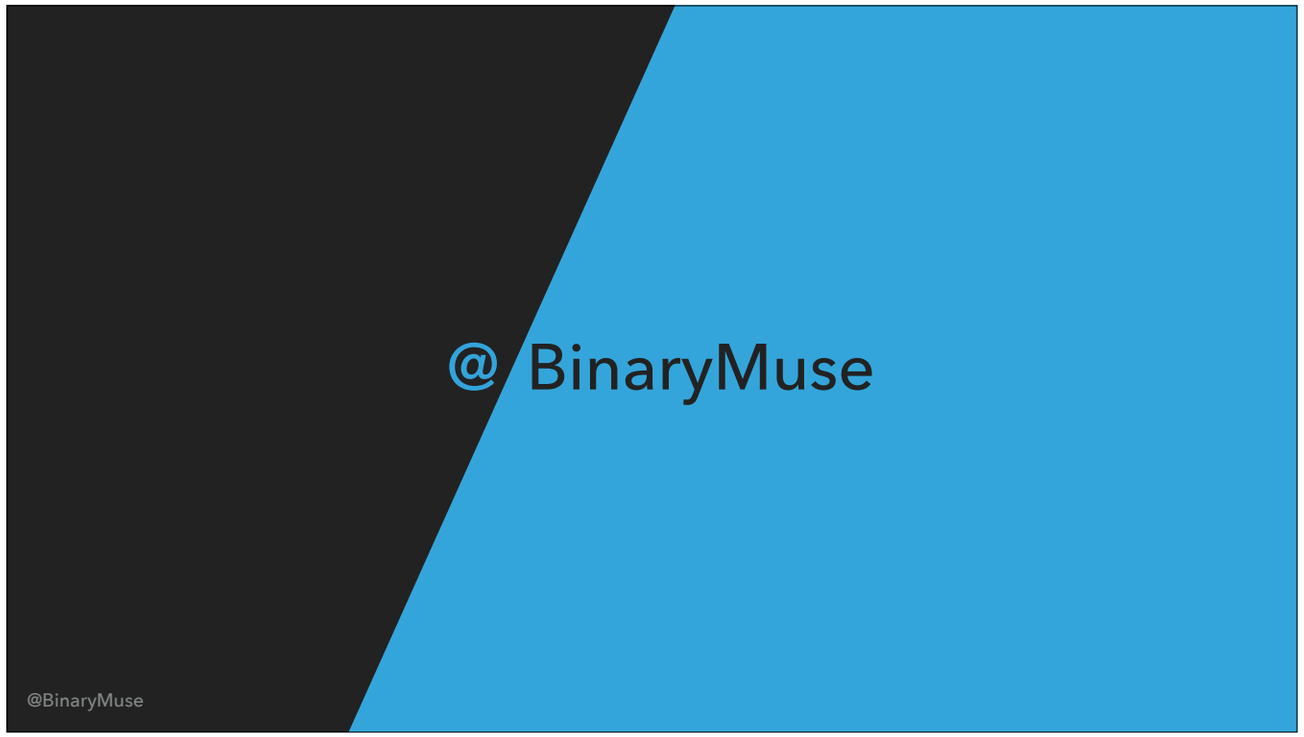
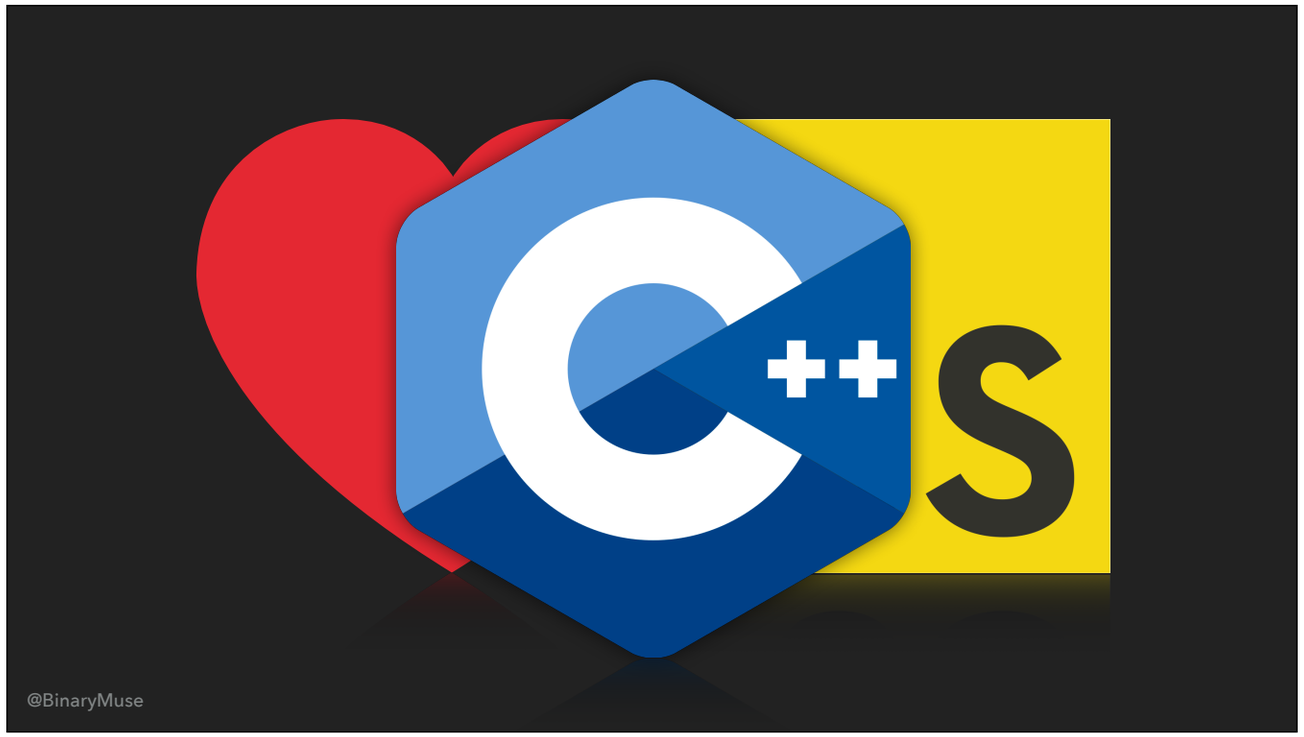
Hello! I’m Michelle Tilley.

I work at a small startup within Microsoft; you might have heard of it.

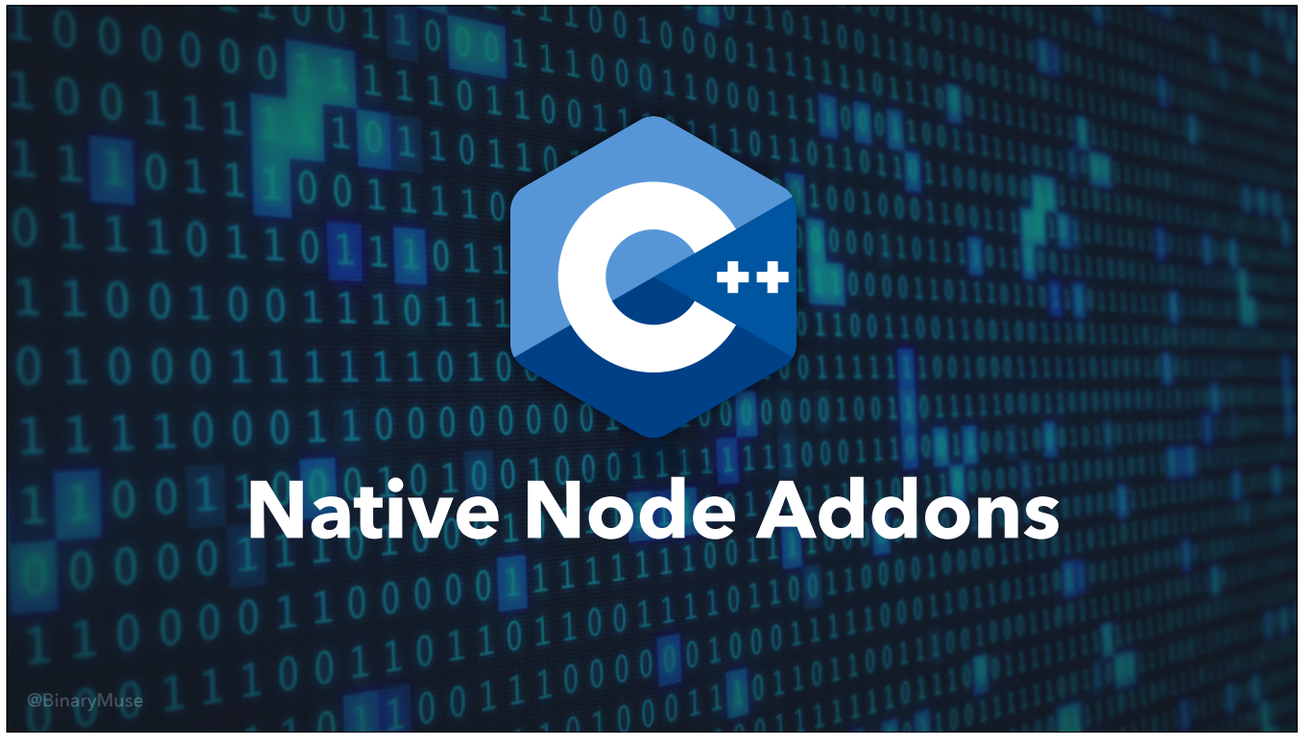


You can find me online as BinaryMuse pretty much everywhere…

…where you can see me talk about my biggest passions, like motorcycles, being extremely gay, and capitalizing the “H” in “GitHub.” But seriously, capitalize that “H.”

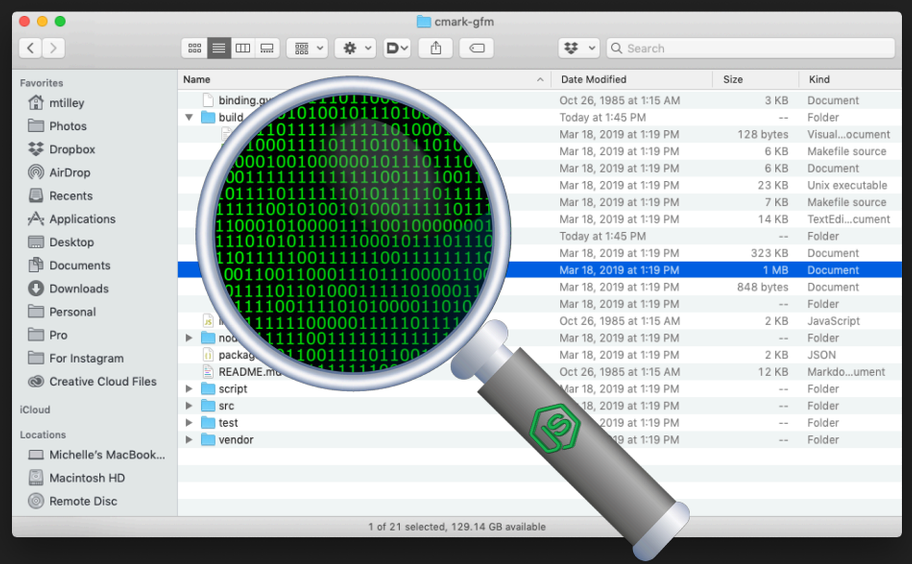


Know what else I’m passionate about? JavaScript! Which is good, cause this is Waffle.js. But we’re not gonna talk about JavaScript today, at least not directly, because I’m here to tell you about C++, namely…



…native Node.js addons.

What is a Native Addon?



@BinaryMuse

So what is a native addon? A native addon is a bit of code [*] usually written in C or C++, that's been compiled down to a file that Node.js is able to load and execute at runtime. This allows you to call C++ code from JS and get JS values back from C++.

How to Make a Native Node Addon

@BinaryMuse

Let’s start by taking a look at how to create a simple native addon.

Now, Node.js uses the V8 JavaScript engine, so to write JavaScript using C++, you need to use the V8 API. The V8 API can be a little difficult to learn and can change fairly significantly across versions, meaning that it’s hard to write a native Node addon that works across many versions of Node (since those versions of Node have different versions of V8). The solution to this problem was…

NAN

Native Abstractions for Node.js

@BinaryMuse

NAN! NAN stands for Native Abstractions for Node.js.

NAN provides logic to make developing native addons across Node versions easier, and also has some help utilities that make addon development in general easier.

For a long time, NAN was the de facto standard for writing native addons. But now, there’s a new recommendation: N-API.

N-API is an API for building native addons that is completely independent from the underlying runtime (i.e., V8) and is maintained as part of Node.js itself. It's also ABI stable across Node.js versions; this means if you write and compile an addon for Node 8, it will work in Node 10 and Node 12 as well without needing to be re-compiled.

N-API is a C-based API, but there's a great C++ wrapper called node-addon-api that we'll use here.

Step 1: add one `binding.gyp`

@BinaryMuse

So once we have `node-addon-api` installed with npm, the first thing we need to do is create a binding.gyp file. binding.gyp is a special file that signals to Node that it needs to build a native addon when installing this package.

Step 1: add one `binding.gyp`

```
{
  "targets": [
    {
      "target_name": "my_addon",
      "sources": [ "addon_src.cc" ],
      "include_dirs": [
        "<!@(node -p \"require('node-addon-api').include\")"
      ],
      "dependencies": [
        "<!(node -p \"require('node-addon-api').gyp\")"
      ],
      "defines": [ "NAPI_DISABLE_CPP_EXCEPTIONS" ],
    }
  ]
}
```

@BinaryMuse

To define a native addon, we'll add some info to this `targets` array. You can have many targets in your binding.gyp file; here we just have one.

Step 1: add one `binding.gyp`

```
{
  "targets": [
    {
      "target_name": "my_addon",
      "sources": [ "addon_src.cc" ],
      "include_dirs": [
        "<!@(node -p \"require('node-addon-api').include\")"
      ],
      "dependencies": [
        "<!(node -p \"require('node-addon-api').gyp\")"
      ],
      "defines": [ "NAPI_DISABLE_CPP_EXCEPTIONS" ],
    }
  ]
}
```

@BinaryMuse

We'll start by specifying the addon's name…

Step 1: add one `binding.gyp`

```
{
  "targets": [
    {
      "target_name": "my_addon",
      "sources": [ "addon_src.cc" ],
      "include_dirs": [
        "<!@(node -p \"require('node-addon-api').include\")"
      ],
      "dependencies": [
        "<!(node -p \"require('node-addon-api').gyp\")"
      ],
      "defines": [ "NAPI_DISABLE_CPP_EXCEPTIONS" ],
    }
  ]
}
```

@BinaryMuse

…and listing all the C++ files that need to be compiled as a part of this addon.

Step 1: add one `binding.gyp`

```
{
  "targets": [
    {
      "target_name": "my_addon",
      "sources": [ "addon_src.cc" ],
      "include_dirs": [
        "<!@(node -p \"require('node-addon-api').include\")"
      ],
      "dependencies": [
        "<!(node -p \"require('node-addon-api').gyp\")"
      ],
      "defines": [ "NAPI_DISABLE_CPP_EXCEPTIONS" ],
    }
  ]
}
```

@BinaryMuse

Since we’re using the node-addon-api wrapper, we’ll need to add its include directory into our own.

Step 1: add one `binding.gyp`

```
{
  "targets": [
    {
      "target_name": "my_addon",
      "sources": [ "addon_src.cc" ],
      "include_dirs": [
        "<!@(node -p \"require('node-addon-api').include\")"
      ],
      "dependencies": [
        "<!(node -p \"require('node-addon-api').gyp\")"
      ],
      "defines": [ "NAPI_DISABLE_CPP_EXCEPTIONS" ],
    }
  ]
}
```

@BinaryMuse

We'll also need to add its own gyp file as a dependency of ours.

Step 1: add one `binding.gyp`

```
{
  "targets": [
    {
      "target_name": "my_addon",
      "sources": [ "addon_src.cc" ],
      "include_dirs": [
        "<!@(node -p \"require('node-addon-api').include\")"
      ],
      "dependencies": [
        "<!(node -p \"require('node-addon-api').gyp\")"
      ],
      "defines": [ "NAPI_DISABLE_CPP_EXCEPTIONS" ],
    }
  ]
}
```

@BinaryMuse

Finally, we need to decide if we want to use C++ exceptions in our addon. The docs explain this in more detail; I've opted to just turn them off here.

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

Now that our binding.gyp file is done, let’s start on the C++ file we specified.

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

First, we'll include `napi.h` to get access to the API that N-API .

Step 2: sprinkle a little C++

```
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

You declare an N-API-based native addon using this NODE_API_MODULE macro. The first parameter is the name of the addon — here we use a special value NODE_GYP_MODULE_NAME which will automatically use the name we provided in binding.gyp — and the second argument is a function that Node will call when initializing our addon.

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

That initialization function takes two arguments: the first is a N-API environment, which is a data structure that represents the underlying JS environment. You need this in order to create new JavaScript values in that same environment. The second argument is an `exports` object that we can attach things to (just like a regular Node module).

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

Inside our initialize method, we'll set a value on the exports object at the key "hello," and that value will be a function. That function definition will come from the C++ function called "Method". We'll also return the exports object.

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

We'll define Method to return a JavaScript string, but notice this argument: it's a reference to something called CallbackInfo. CallbackInfo is a data structure that has information about the JS function call that was made to invoke this C++ function; it has things like the number of arguments that were passed, the value of those arguments, the value of `this`, and so on.

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

We'll use this CallbackInfo object to get the current environment…

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

…and then we'll use it to create a new JS string with the value "world," which we'll return from the function.

Step 2: sprinkle a little C++

```cpp
#include "napi.h"

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

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

NODE_API_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

@BinaryMuse

So here’s the full source for our tiny native addon.

Step 3: we did the thing

```
$ node-gyp rebuild
$ node

> my_addon = require('./build/Release/my_addon')
{ hello: [Function: hello] }

> my_addon.hello()
'world'
```

@BinaryMuse

Now we can build our addon with `node-gyp rebuild` and require it, and when we call the `hello` method we get our `world` string as the return value.

How to Make a Native Node Addon

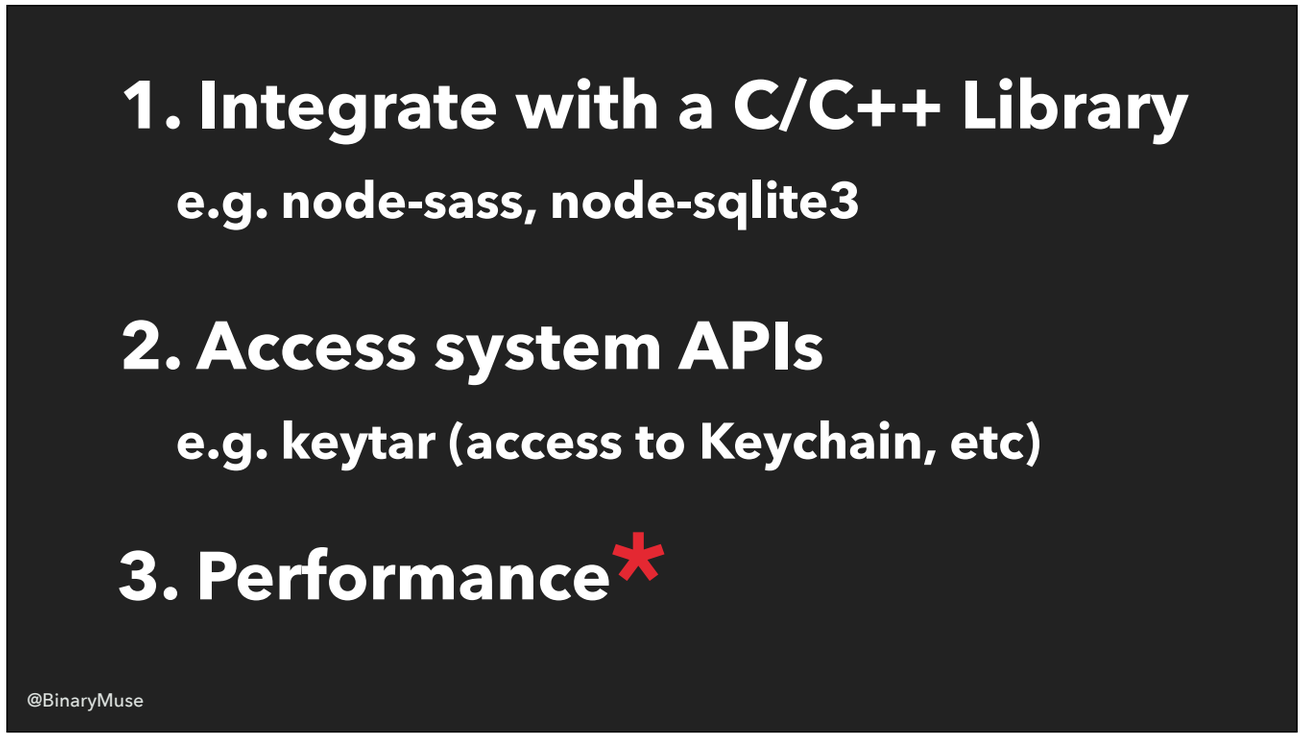
@BinaryMuse

Okay, now we know HOW to create a native addon (albeit a very simple one)…

Okay but
Why to Make a Native Node Addon

@BinaryMuse

…but we haven’t talked about WHY you might want to make a native addon.



One common reason is to integrate with a library that's written in C or C++. This is commonly referred to as writing "bindings" for a library. Popular packages that supply Node.js bindings to other libraries include node-sass (which binds to libsass) or node-sqlite3, which provides bindings for SQLite3

Another reason is to get access to the host system APIs that Node.js doesn't provide by default. For example, the keytar package integrates with the Keychain on mac, the Credential Manager on Windows, and libsecret on Linux.

Another reason people reach for native addons is for the performance gain they can get with C++. But there's a big, giant, on-fire asterisk on this point. To demonstrate, let's look at an example:

Sieve of Eratosthenes (JS)

```
const sieve = max => {
  const sieve = new Array(max).fill(true)

  for (let i = 2; i < Math.sqrt(max); i++) {
    if (sieve[i]) {
      for (let j = Math.pow(i, 2); j < max; j += i) {
        sieve[j] = false
      }
    }
  }

  const primes = [];

  for (let i = 0; i < sieve.length; i++) {
    if (i <= 1) continue; // 0 and 1 are not prime
    if (sieve[i] === true) {
      primes.push(i)
    }
  }

  return primes
}
```

@BinaryMuse

Here's a simple implementation of the Sieve of Eratosthenes written in JavaScript. It takes a number and computes all the prime numbers up to the given number. Essentially, it loops over every number, and "crosses off" all the numbers that are a multiple of that number; the numbers that are left over are prime.

So this is a pretty computationally focused algorithm; nothing fancy going on, just some loops, math, and simple conditionals.

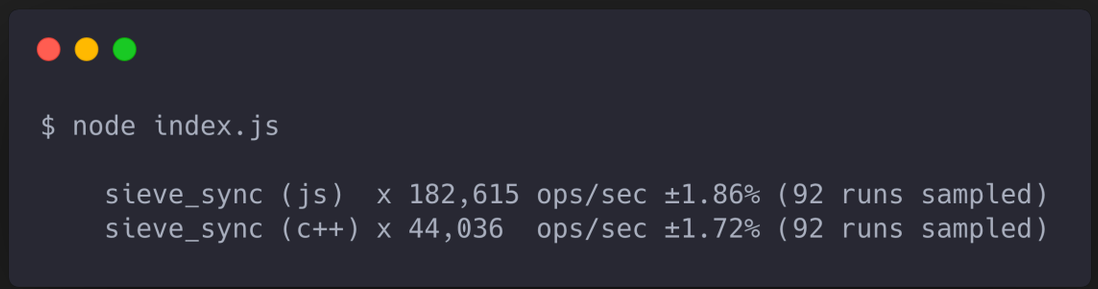
Sieve of Eratosthenes (C++)

```cpp
Napi::Value Sieve(const Napi::CallbackInfo& info) {
  int max = info[0].As<Napi::Number>().Int32Value();
  std::unique_ptr<bool[]> array(new bool[max]);
  for (int i = 0; i < max; i++) {
    array[i] = true;
  }

  for (int i = 2; i < sqrt(max); i++) {
    if (array[i] == true) {
      for (int j = pow(i, 2); j < max; j += i) {
        array[j] = false;
      }
    }
  }

  int next = 0;
  Napi::Array primes = Napi::Array::New(info.Env());
  for (int i = 0; i < max; i++) {
    if (i <= 1) continue;
    if (array[i] == true) {
      primes.Set(next++, i);
    }
  }

  return primes;
}
```

@BinaryMuse

Here’s the exact same algorithm implemented in C++. Which one do you think is faster?

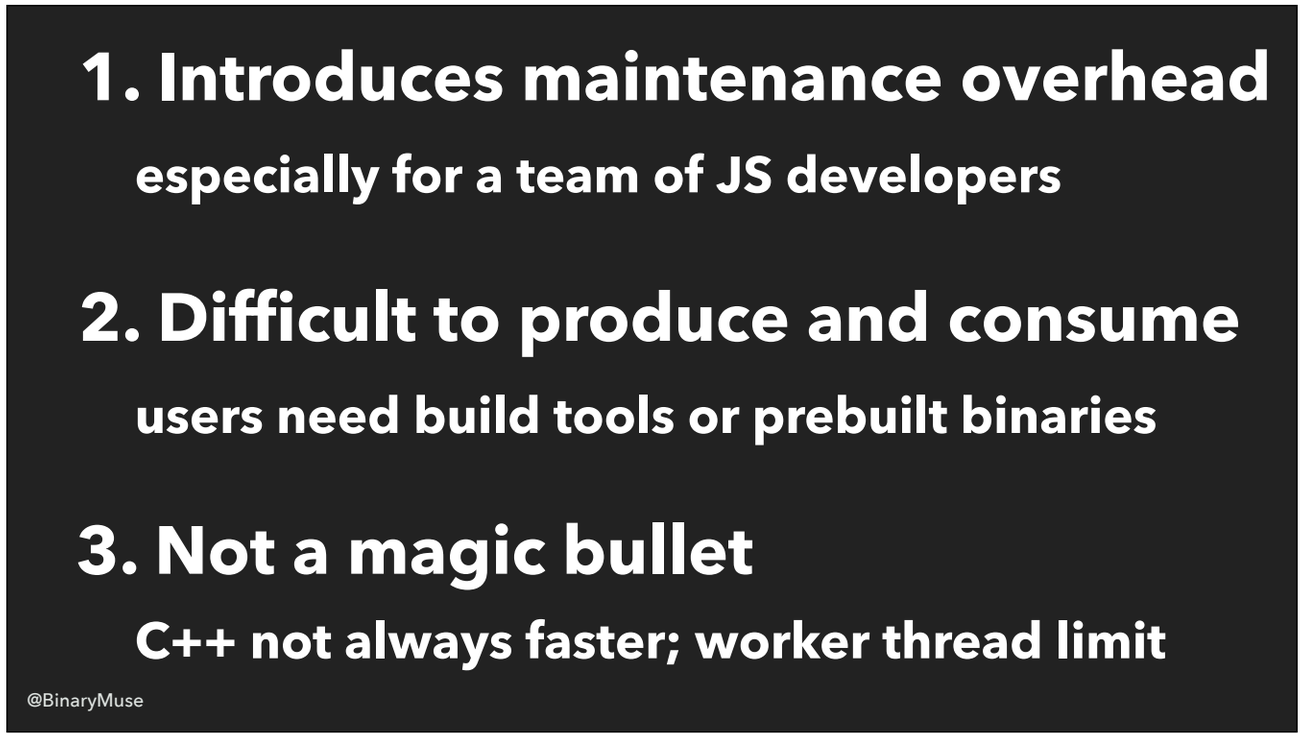
Sieve of Eratosthenes Results



@BinaryMuse

The answer, as you might have guessed from my ominous foreshadowing, is that the JavaScript version is faster. It’s more than FOUR TIMES faster than the C++ version. So this is a good lesson to take away: don’t assume your hand-rolled C++ code is faster than the optimized code V8 can generate for your JavaScript.

Okay, so we've talked about some common reasons to make native addons; now let's talk about some reasons you may NOT want to write a native addon.



First of all, introducing a native addon into a project raises its maintenance overhead, especially if your team is full of JS devs and not necessarily people with C/C++ experience. It's easy to get C or C++ code wrong, especially if you're not familiar with the language and library patterns. You have to do a lot more management of the resources your code uses, and getting it wrong can lead to your Node process crashing.

Secondly, native addons are difficult to produce and consume. On a consuming machine, you need a working compiler in order to build native addons. Alternatively, addon developers can publish prebuilt binaries using something like prebuild and prebuild-install, but creating binaries for every possible target system and Node version can be difficult. One Node.js developer told me that they think "If the best solution to a problem is a native addon, then Node is letting developers down."

Finally, native addons aren't a magic bullet. We've already seen that C++ isn't automatically faster than V8-optimized JavaScript. Furthermore, if you're making things async with libuv, it's worth noting that Node uses only a small number of threads in the libuv worker pool, so it's possible to write code that saturates them and makes other async work slower.

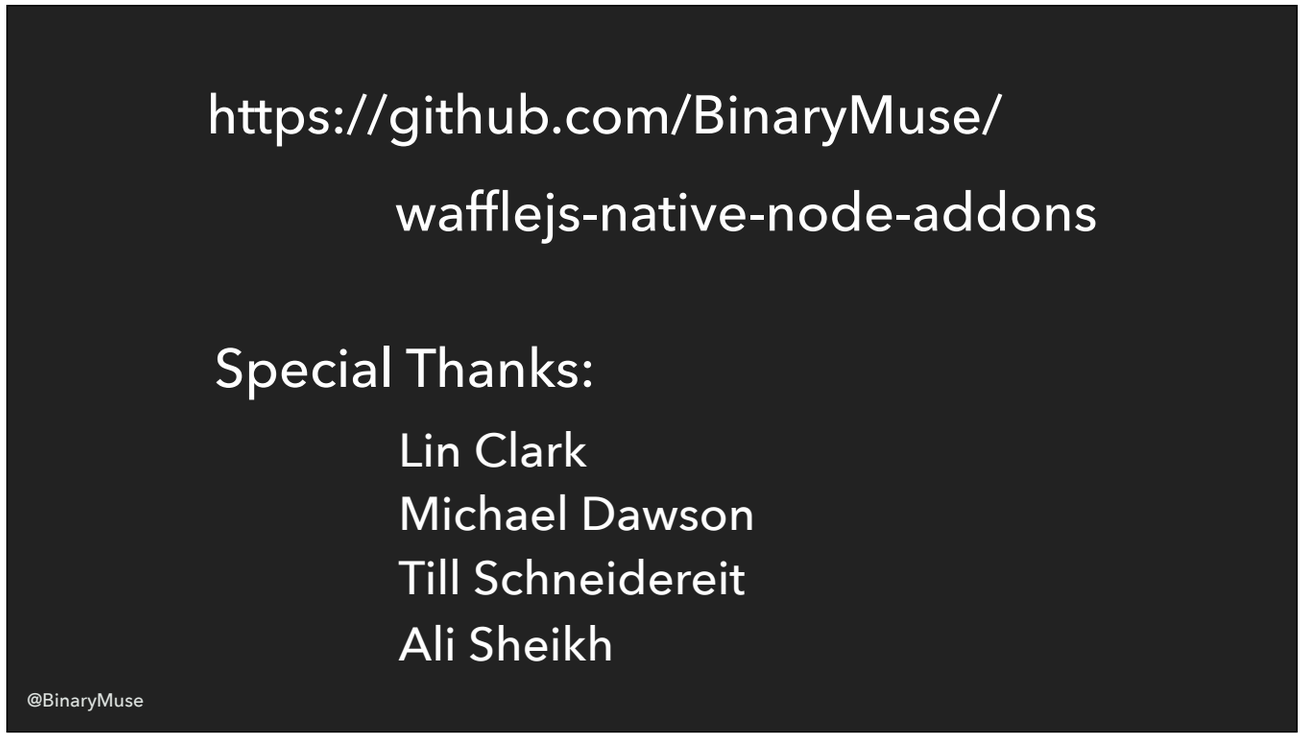
The Future

@BinaryMuse

Now that we've discussed native Addons and their benefits and drawbacks, let's look toward the future.

One big thing you should pay attention to if you're not already is Web Assembly, or wasm. Wasm already serves as a compile target for higher level languages like C, C++, and Rust, but thus far it's been limited in what it can do; in particular, there isn't a good story for wasm accessing the host system.

Just last week, however, Lin Clark posted an article on hacks.mozilla.org detailing the standardization effort for [*] WASI - the web assembly system interface. WASI would provide portable and secure access to a host system, which could open the door for using wasm instead of native code in a lot of cases. Be sure to check out Lin's article on hacks.mozilla.org for more information on WASI.



You can find resources for this talk, including links to external articles and the source code for my examples, on my GitHub at github.com/BinaryMuse. And special thanks to these folks who helped me with research. Thanks for your time!