KOSSCON 2018

NODE.JS N-API

JINHO BANG
zino@chromium.org

WHO AM I?

Samsung Electronics

Chromium/Blink OWNER

W3C Spec Editor

Node.js Contributor

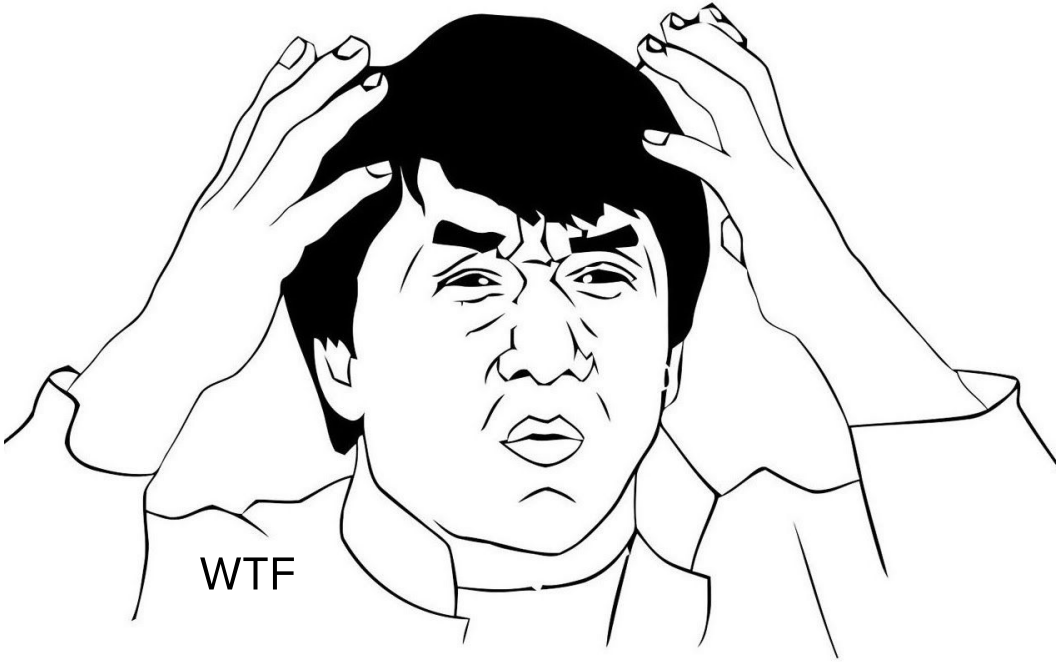
KOSSLAB Researcher

CONTENTS

- Motivation
- What’s differences with C++ Addon?
- N-API ABI Stability
- Performance
- How to implement N-API
- WebIDL Binding Generator (Bacardi Project)

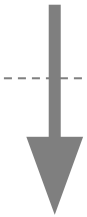
MOTIVATION

N-API is a stable Node API layer for Native Modules


WTF

JS

```
let s = sum([1, 2, 3, 4, 5, 6, 7, 8, 9]);
```



JS

```
function sum(elements) {
    let s = 0;
    elements.forEach(element => { s += element; });
    return s;
}
```

JS

```
let s = sum([1, 2, 3, 4, 5, 6, 7, 8, 9]);
```

Native

```
int sum(std::vector<int> elements) {
    int s = 0;
    for (int i = 0; i < elements.size(); i++)
        s += elements[i];
    return s;
}
```

Why Native Modules?

- Performance
- Access physical devices, for example a serial port
- expose functionality from OS not otherwise available
- Use existing third_party components written in Native Code

N-API native binding VS WASM

- Native VS VM (Virtual Machine)
- WASM code should be portable
- Low Level APIs (System calls)

WHAT'S DIFFERENCES WITH ADDON?

C++ Addons

Node.js Addons are dynamically-linked **shared objects**, written in C++, that can be **loaded into Node.js using the require() function**, and used just as if they were an ordinary Node.js module.

Requires the following knowledges

- V8
- libuv
- Internal Node.js libraries

```
module.exports.hello = () => 'world';
```

Hello World in C++ Addons

```
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

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

void Initialize(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)

}  // namespace demo
```

What’s the problems?

- Too complicated
- Much knowledges required
- API Unstable

NAN

(Native Abstraction for Node.js)

Thanks to the **crazy changes in V8** (and some in Node core), keeping native addons **compiling happily across versions**, particularly 0.10 to 0.12 to 4.0, is a minor nightmare.

The goal of this project is to store all logic necessary to develop native Node.js addons **without** having to inspect **NODE_MODULE_VERSION** and get yourself into a **macro-tangle**.

It looks great!

BUT..

What's the problems?

- Native modules must be recompiled for each version of Node.js (ABI Unstable)
- The code within modules may need to be modified for a new version
- It's not clear which parts of the V8 API the Node.js community believes are safe/unsafe to use in terms of long term support for that use
- Modules written against the V8 APIs may or may not be able to work with alternate JS engines when/if Node.js supports them

N-API is key solution for everything

PERFORMANCE

Perf Leveldown

Ported using C style API
Benchmark includes 1M entries
DB Size 110 MB
N-API adds 5% perf delta

Leveldown Nan on V8-Node 8.x	Leveldown NAPI on V8-Node-Napi 8.x
Elapsed: 45.867s	Elapsed: 47.619s
Elapsed: 44.805s	Elapsed: 47.535s
Elapsed: 45.134s	Elapsed: 47.506s
Elapsed: 45.054s	Elapsed: 46.482s
Elapsed: 44.739s	Elapsed: 47.694s
avg(elapsed) 45.1198s	**avg(elapsed) 47.3672s (+4.98%)**

System Info:
Windows 10 x64
Intel Xeon E5-1620 v3 @ 3.50GHz
16GB DDR4 @ 2133MHz
Samsung XP941 SSD

Details at: https://github.com/nodejs/abi-stable-node/issues/55

Perf Nanomsg

Ported using C style API
Workload size 1 byte message
Performance within expected range

Items	Non N-API	N-API	Delta
Latency [us]	107.1128	115.5018	7.83%
Throughput [msg/s]	4679.6	4683.6	0.09%
Throughput [Mb/s]	0.0374	0.0376	0.53%

System Info:
Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-55-generic x86_64)
Intel CPU @ 2400 MHz

Details at: https://github.com/nodejs/abi-stable-node/issues/57

Perf
Node-Sass

Ported using C style API
N-API adds 1.9% perf delta

System Info:
Windows 10 x64
Intel Xeon E5-1620 v3 @ 3.50GHz
16GB DDR4 @ 2133MHz
Samsung XP941 SSD

node-sass Nan on V8-Node 8.x	node-sass NAPI on V8-Node-Napi 8.x
12ms	13ms
12ms	14ms
13ms	12ms
12ms	17ms
13ms	13ms
17ms	12ms
13ms	15ms
12ms	12ms
12ms	12ms
12ms	12ms
24ms	12ms
11ms	12ms
13ms	24ms
15ms	13ms
13ms	13ms
12ms	12ms
13ms	15ms
11ms	12ms
12ms	12ms
12ms	12ms
avg = 13.2ms	avg = 13.45ms (+1.9%)

Details at: https://github.com/nodejs/abi-stable-node/issues/82

Perf Canvas

Ported using C++ wrapper
Perf regression in chatty benchmarks

Scenario	baseline ops/s	napi ops/s	baseline µs/op	napi µs/op	change %
lineTo()	13,332,181	2,642,581	0.08	0.38	505%
arc()	798,976	498,373	1.25	2.01	160%
fillStyle= hex	2,301,528	1,785,114	0.43	0.56	129%
fillStyle= rgba()	1,998,049	1,421,991	0.50	0.70	141%
strokeRect()	7,535,580	1,962,890	0.13	0.51	384%
linear gradients	432,867	182,450	2.31	5.48	237%
toBuffer() 200x200	257	258	3889.06	3875.00	100%
toBuffer() 1000x1000	10	10	99100.00	99850.00	101%
toBuffer() async 200x200	838	837	1192.97	1194.14	100%
PNGStream 200x200	252	254	3960.94	3935.94	99%
getImageData(0 0 100 100)	17,847	17,709	56.03	56.47	101%
createImageData(300x300)	11,683	9,961	85.60	100.39	117%
moveTo() / arc() / stroke()	1,203,047	261,725	0.83	3.82	460%
toDataURL() 200x200	257	257	3892.19	3895.31	100%
toBuffer().toString("base64")	258	259	3876.56	3859.38	100%
toBuffer() async 1000x1000	33	33	30050.00	29975.00	100%

System Info:
Windows 10 x64
Intel Xeon W3530 @2.8GHz, 20 GB RAM

Details at: https://github.com/nodejs/abi-stable-node/issues/77

HOW TO IMPLEMENT N-API?

Implement native version sum()

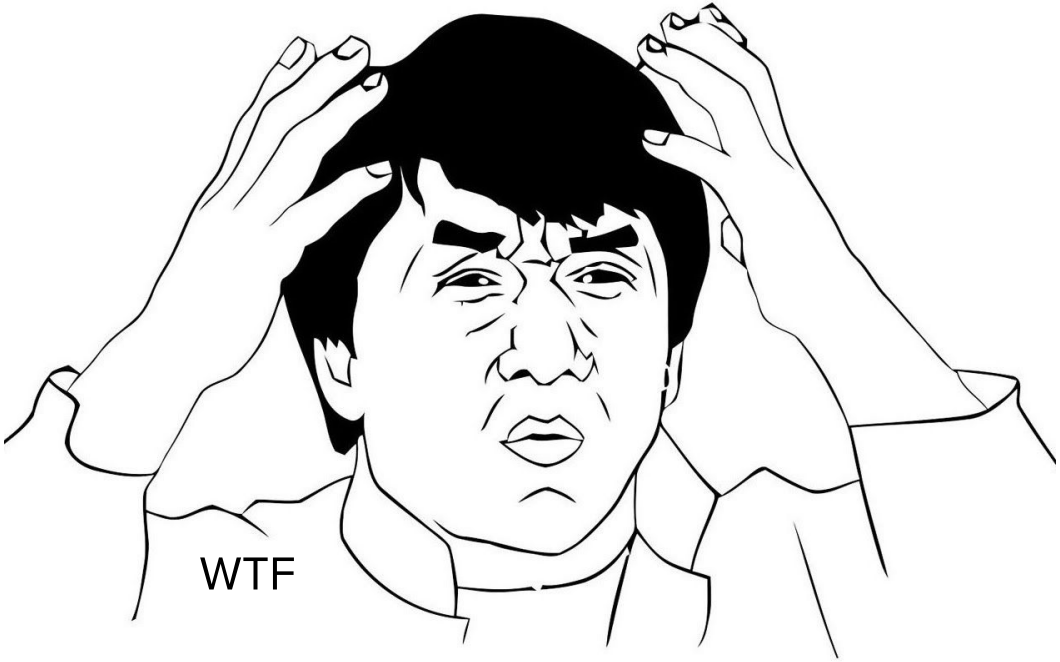
```
napi_value sum(napi_env, napi_callback_info info) {
    napi_status status;
    size_t args = 1;
    napi_value args[1];
    napi_status  = napi_get_cb_info(env, info, &argc, args, nullptr, nullptr);

    if (argc < 1) {
        napi_throw_type_error(env, nullptr, "...");
        return nullptr;
    }

    uint32_t length = 0;
    napi_get_array_length(env, args[0], &length);
```

Implement native version sum()

```
double sum = 0;
for (int i = 0; i < length; i++) {
    napi_value element;
    napi_get_element(env, i, &element);

    napi_valuetype valuetype;
    napi_typeof(env, element, &valuetype);
    if (napi_valuetype != napi_number) {
        napi_throw_type_error(env, nullptr, "...");
        return nullptr;
    }

    double value;
    napi_get_value_double(env, element, &value);
    sum += value;
}
```

Implement native version sum()

```
    napi_value js_sum;
    napi_create_double(env, sum, &js_sum);
    return js_sum;
}
```


WTF

WEB-IDL BINDING GENERATOR

What’s the problems?

- Argument length checking
- Type checking
- Type converting
- Memory management
- Readability

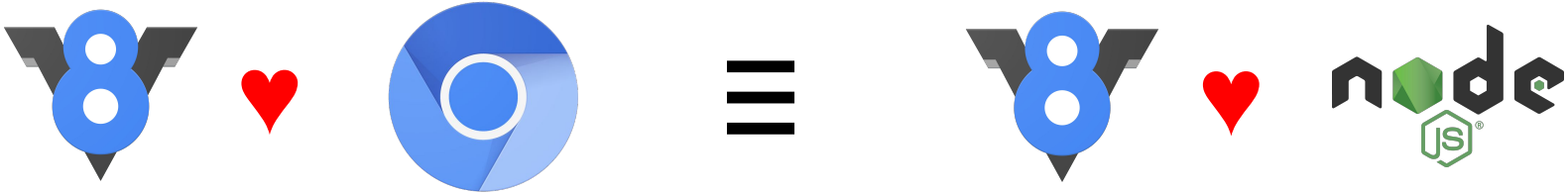
What's the problems?

```
int sum(std::vector<int> elements) {
    int s = 0;
    for (int i = 0; i < elements.size(); i++)
        s += elements[i];
    return s;
}
```

```
napi_value Sum(napi_env, napi_callback_info info) {
    napi_status status;
    size_t args = 1;
    napi_value args[1];
    napi_status  = napi_get_cb_info(env, info, &argc, args, nullptr, nullptr);

    if (argc < 1) {
        napi_throw_type_error(env, nullptr, "…");
        return nullptr;
    }

    uint32_t length = 0;
    napi_get_array_length(env, args[0], &length);

    double sum = 0;
    for (int i = 0; i < length; i++) {
        napi_value element;
        napi_get_element(env, i, &element);

        napi_valuetype valuetype;
        napi_typeof(env, element, &valuetype);
        if (napi_valuetype != napi_number) {
            napi_throw_type_error(env, nullptr, "…");
            return nullptr;
        }

        double value;
        napi_get_value_double(env, element, &value);
        sum += value;
    }

    napi_value js_sum;
    napi_create_double(env, sum, &js_sum);
    return js_sum;
}
```

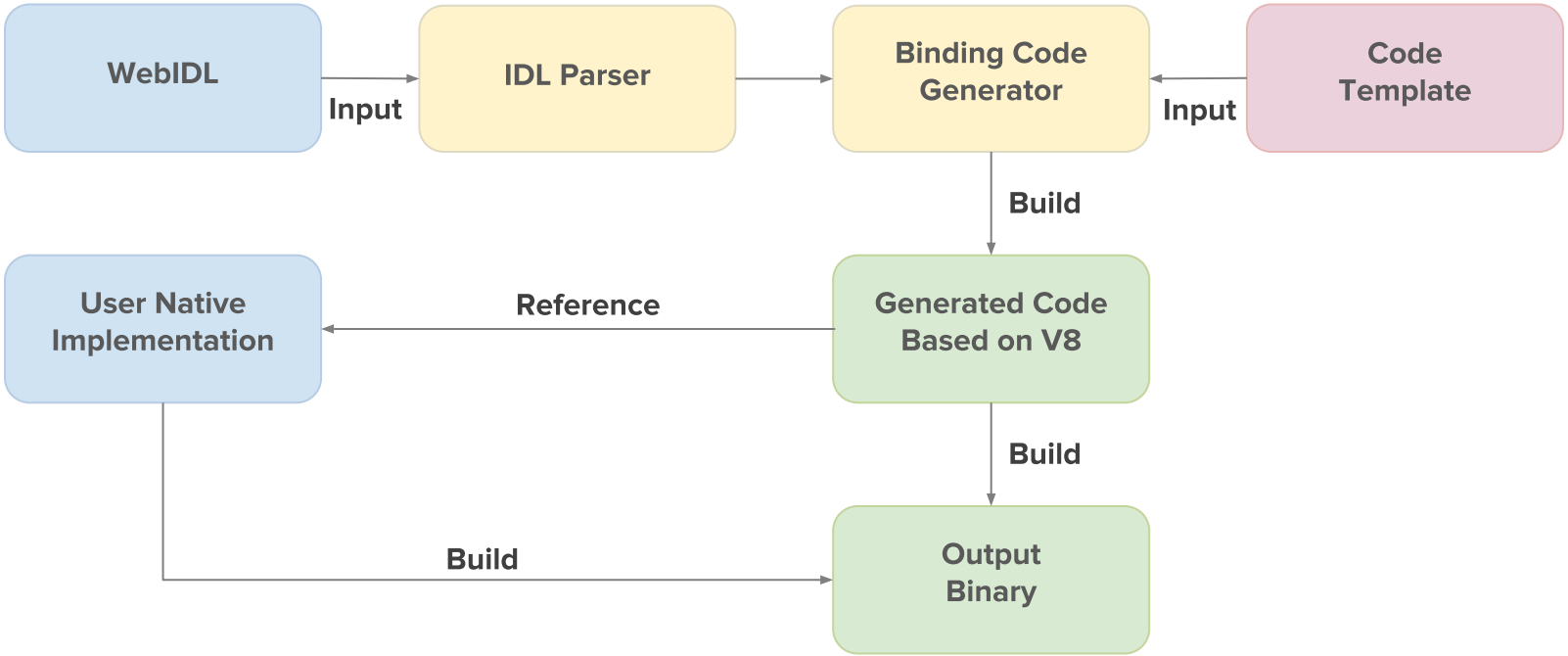
One solution is **WebIDL**

WebIDL is a language that defines how Web Platform are bound to JS


V8
(Javascript
Engine)
Blink
(Rendering
Engine)
V8
(Javascript
Engine)
Node.js
Native
Module

```
// WebIDL
[Constructor]
interface Calculator {
     double sum(sequence<long> elements);
};
```

```
napi_value Sum(napi_env, napi_callback_info info) {
    napi_status status;
    size_t args = 1;
    napi_value args[1];
    napi_status  = napi_get_cb_info(env, info, &argc, args, nullptr, nullptr);

    if (argc < 1) {
        napi_throw_type_error(env, nullptr, "...");
        return nullptr;
    }

    uint32_t length = 0;
    napi_get_array_length(env, args[0], &length);

    double sum = 0;
    for (int i = 0; i < length; i++) {
        napi_value element;
        napi_get_element(env, i, &element);

        napi_valuetype valuetype;
        napi_typeof(env, element, &valuetype);
        if (napi_valuetype != napi_number) {
            napi_throw_type_error(env, nullptr, "...");
            return nullptr;
        }

        double value;
        napi_get_value_double(env, element, &value);
        sum += value;
    }

    napi_value js_sum;
    napi_create_double(env, sum, &js_sum);
    return js_sum;
}
```


WebIDL
Input
IDL Parser
Binding Code Generator
Input
Code Template
Build
User Native Implementation
Reference
Generated Code Based on V8
Build
Build
Output Binary

https://github.com/lunchclass/bacardi

https://github.com/nodejs/node-addon-api/issues/294

Intent to implement: WebIDL Binding Generator #294

Open **romandev** opened this issue 17 days ago · 3 comments

romandev commented 17 days ago — Contributor

Hi all,

I'm Jinho Bang and a contributor in this project.

Today, I'd like to suggest a new feature a.k.a WebIDL Binding Generator. If we use it, we can just generate N-API binding code automatically. I've been implementing POC demo with **@yjaeseok** and **@nadongguri** for last few weeks.

Let's look deep into WebIDL Binding Generator.

Introduction

To help you better understand, let's see the following example. I did copy N paste an example from abi-stable-node-addon-api examples repo. As you already know, if we use it, we can invoke native `add()` function in JS side.

Example: `add()` function

```
#include <napi.h>
```

What is the WebIDL Binding Generator?

The WebIDL Binding Generator is based on WebIDL to generate a binding code automatically.

WebIDL Binding Generator is typically used in the Chromium project. Chromium uses the Blink engine and the Javascript V8 engine. They integrate these two engines, by WebIDL Binding Generator. This can avoid issues such as Type Checking, Type Converting, and Manage Isolate & Context in the Binding process and increase productivity. You can find out about using Chromium's WebIDL through the link below.
https://www.chromium.org/blink/webidl

What's the benefits of WebIDL Binding Generator?

It has the following advantages.

Code complexity is reduced

The binding code and the implementation of native code are separated so we can keep the code simple. Here's the example of comparing the code complexity when using WebIDL Binding Generator.

node-addon-api binding code

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

  if (info.Length() < 2) {
    Napi::TypeError::New(env, "Wrong number of arguments").ThrowAsJavaScriptException();
    return env.Null();
  }

  if (!info[0].IsNumber() || !info[1].IsNumber()) {
    Napi::TypeError::New(env, "Wrong arguments").ThrowAsJavaScriptException();
    return env.Null();
  }
```

WebIDL and implementation

```
// idl
interface Calculator {
    double add(double a, double b);
}

// native implementation
double Add(double a, double b) {
  return a + b;
}
```

Thank you