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

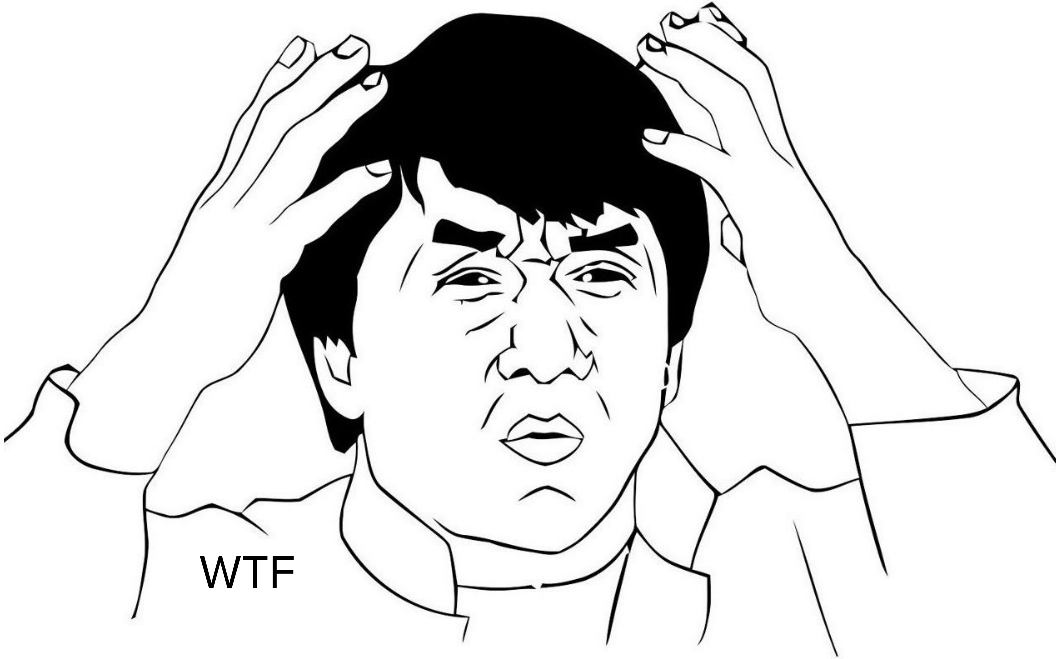
KOSSLAB Researcher

CONTENTS

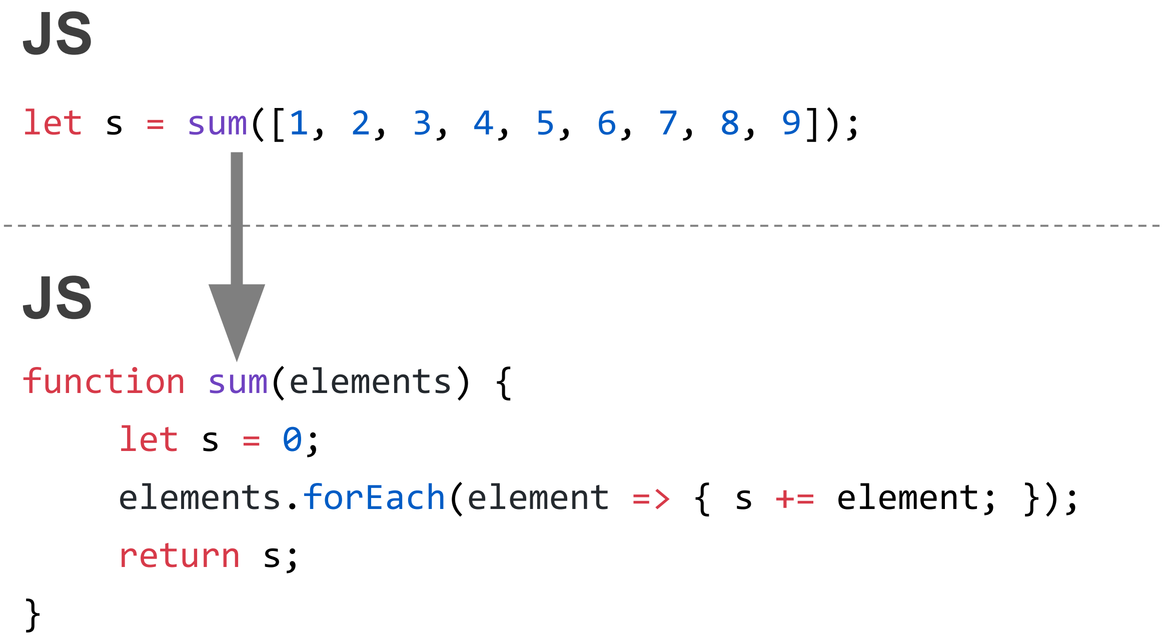
- Motivation
- What’s differences with C++ Addon?
- N-API ABI Stability
- How to implement N-API
- WebIDL Binding Generator (Bacardi Project)

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

Native Module VS WASM (Web Assembly)

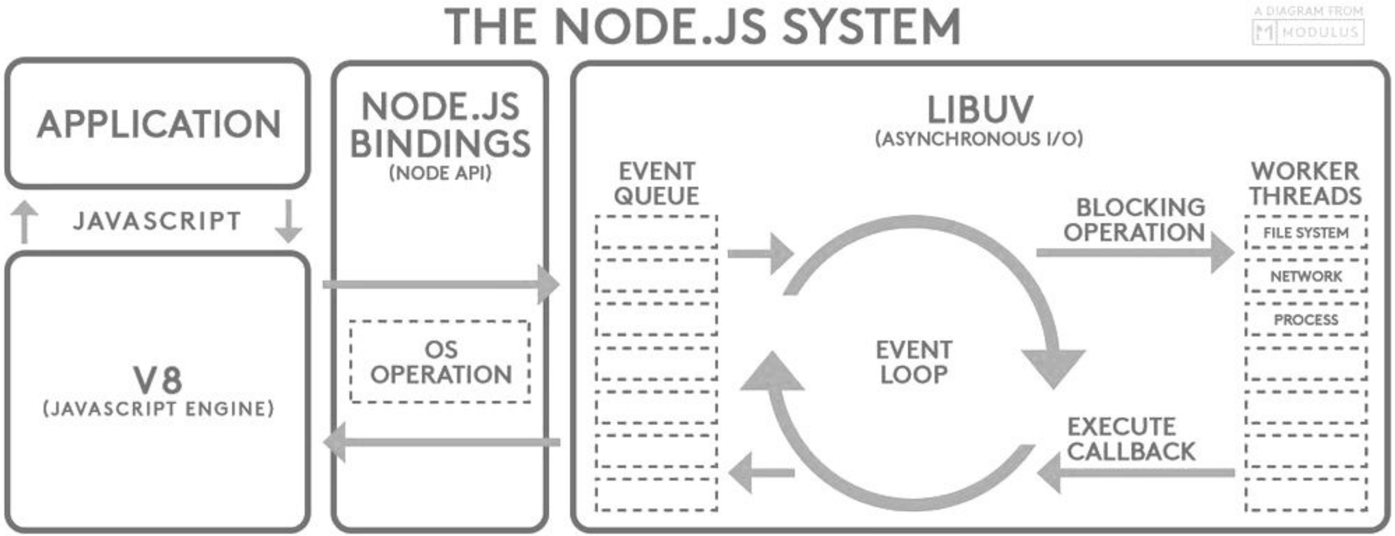
- Native VS VM (Virtual Machine)
- WASM code should be portable
- Low Level APIs (System calls)

WHAT’S DIFFERENCES WITH ADDON?

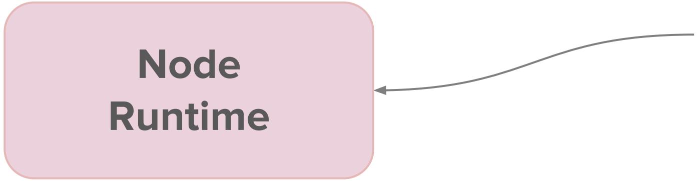
C++ Addons

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.

How Node.js works?



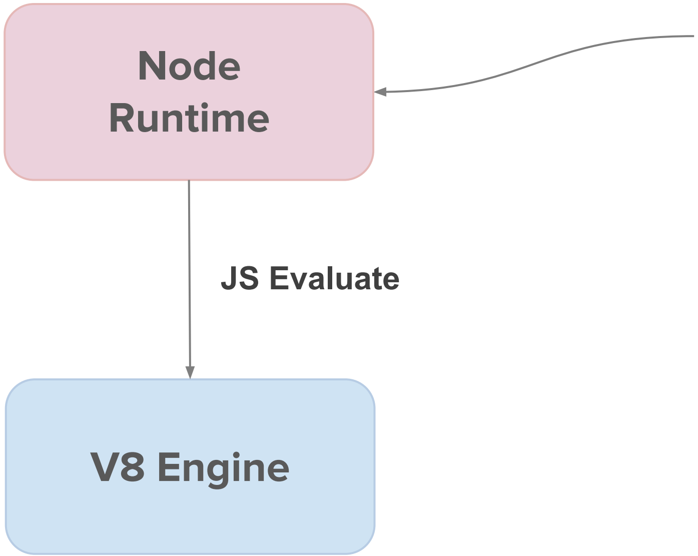
How Node.js works?



Node Runtime

```
const fs = require('fs');
let contents = fs.readFileSync('temp.txt', 'utf8');
```

How Node.js works?



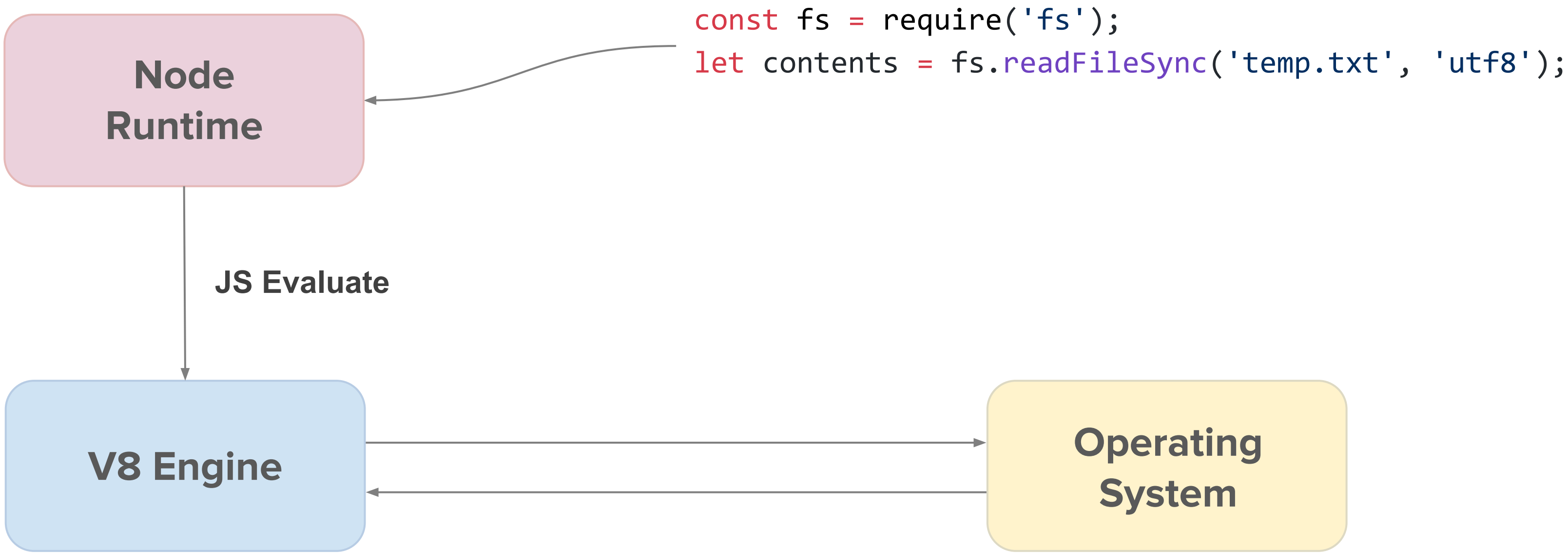
Node Runtime

JS Evaluate

V8 Engine

```
const fs = require('fs');
let contents = fs.readFileSync('temp.txt', 'utf8');
```

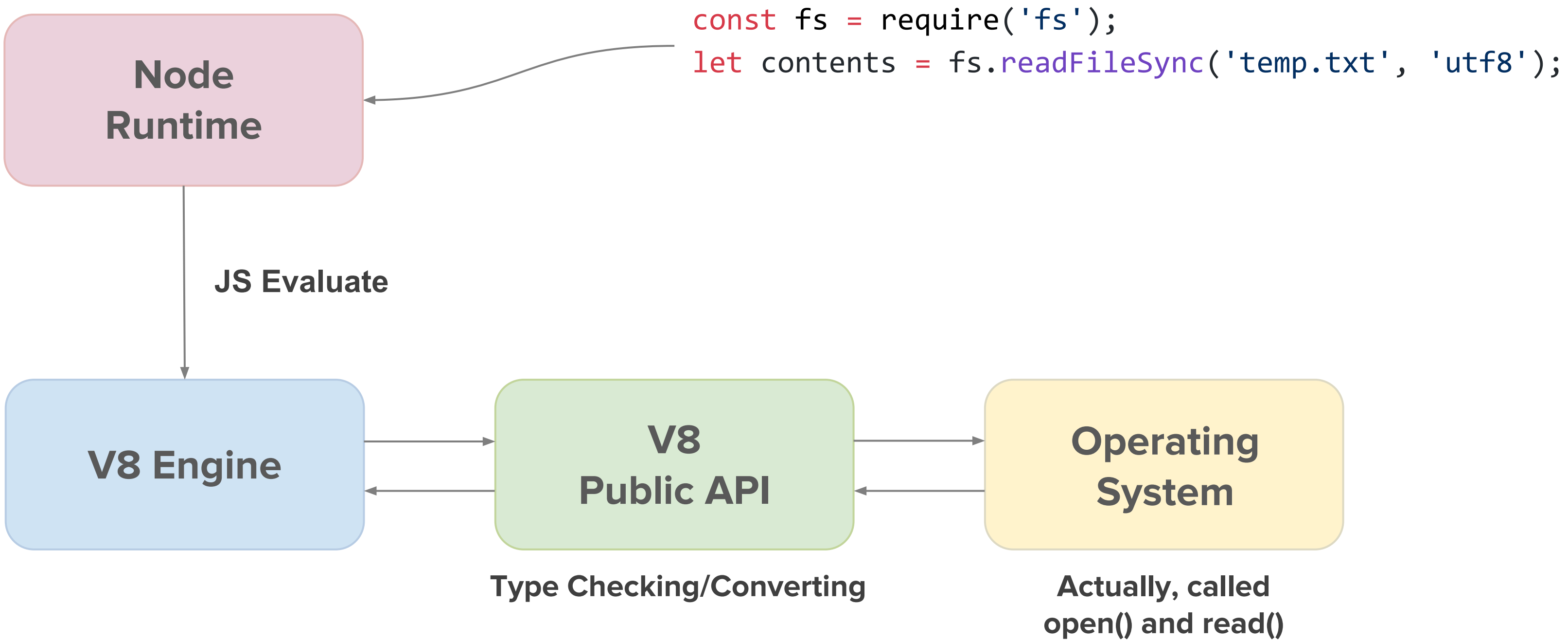
How Node.js works?



```
const fs = require('fs');
let contents = fs.readFileSync('temp.txt', 'utf8');
```

Actually, called open() and read()

How Node.js works?



```
const fs = require('fs');
let contents = fs.readFileSync('temp.txt', 'utf8');
```

Node Runtime

JS Evaluate

V8 Engine

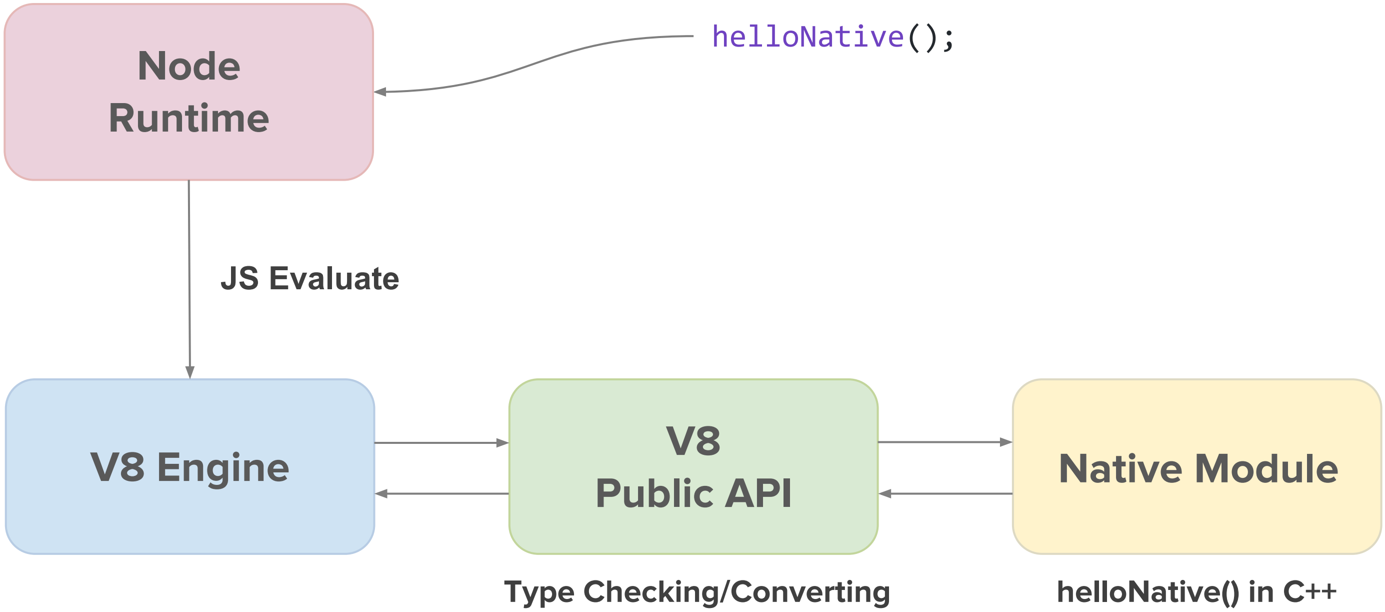
V8 Public API

Type Checking/Converting

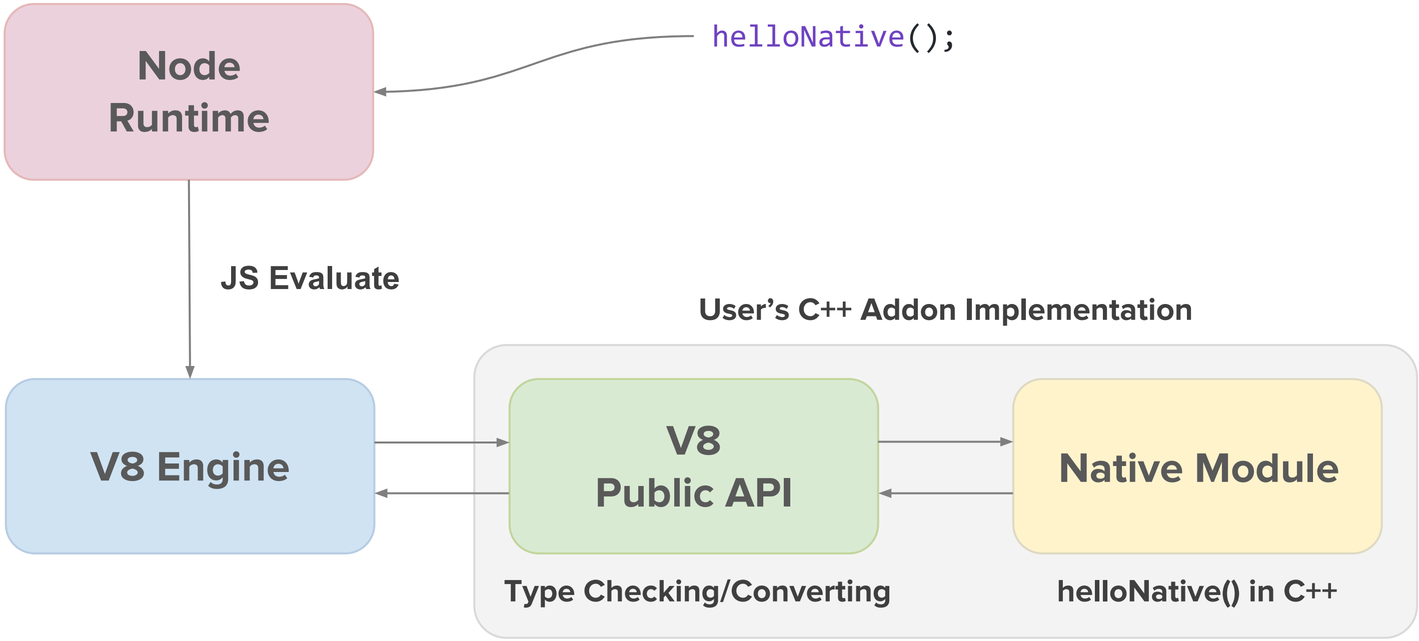
Operating System

Actually, called open() and read()

How C++ Addon Works?



How C++ Addon Works?



```
function hello() {
  return '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 Hello(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", Hello);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)

}  // namespace demo
```

What’s the problems?

- Too complicated
- Much knowledges required
- **API Unstable**

V8 Public APIs are unstable

```
// Node v0.10
Handle<Value> Hello(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("world"));
}
```

V8 Public APIs are unstable

```
// Node v0.10
Handle<Value> Hello(const Arguments& args) {
  HandleScope scope;
  return scope.Close(String::New("world"));
}

// Node v0.12
void Hello(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"));
}
```

NAN

(**N**ative **A**bstraction for **N**ode.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**.

Hello World in NAN

```
void Hello(const Nan::FunctionCallbackInfo<v8::Value>& info) {
  info.GetReturnValue().Set(Nan::New("world").ToLocalChecked());
}
```

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

N-API ABI STABILITY

What’s the **“ABI Stability”**?

ABI

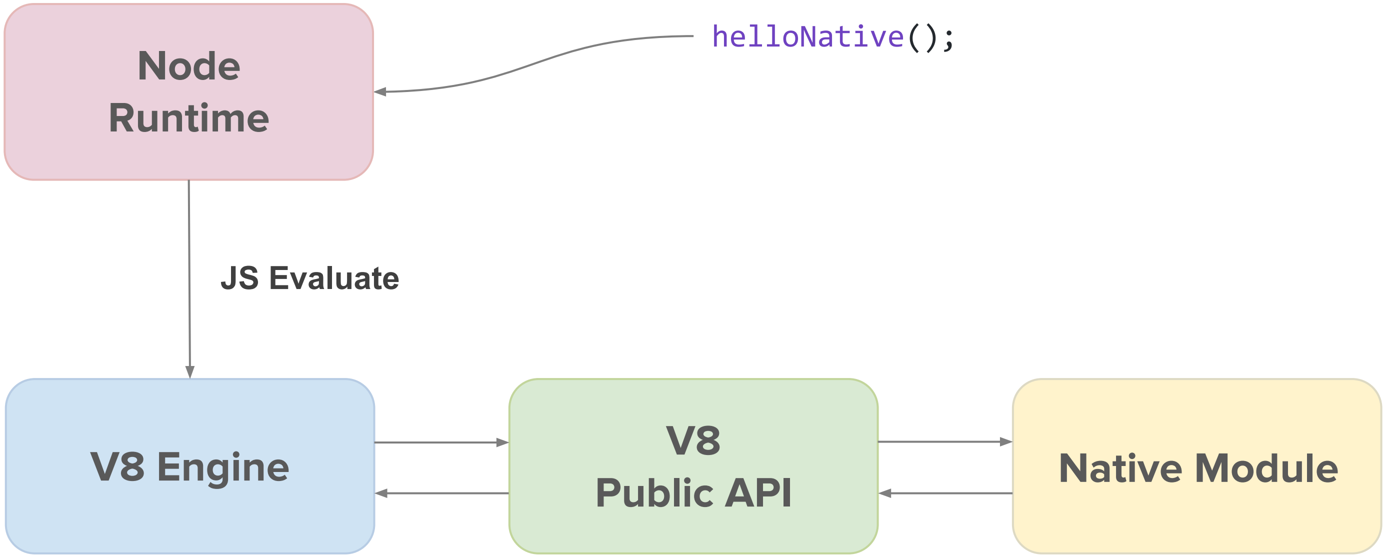
(**A**pplication **B**inary **I**nterface)

API VS ABI

Factors affecting ABI Stability

- Sizes, layouts, and alignments of data types
- Calling convention
- How an application should make system calls to the operating system
- Name mangling

N-API ABI Stability



helloNative();

Node Runtime

JS Evaluate

V8 Engine

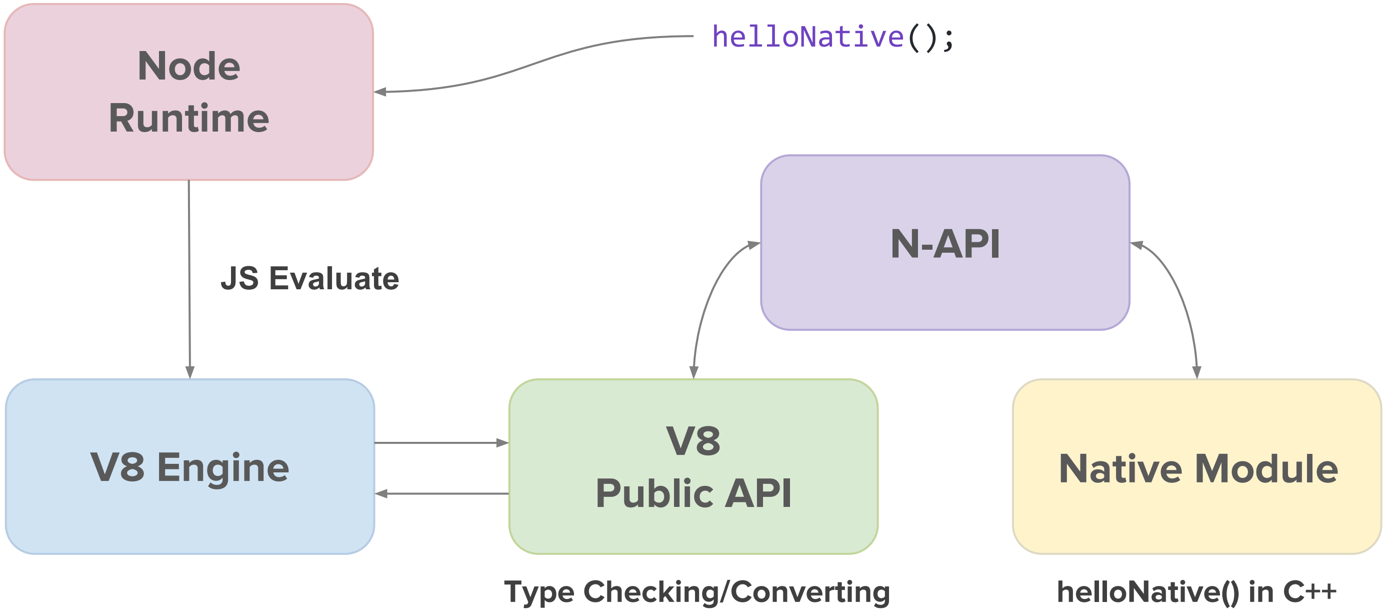
V8 Public API

Native Module

Type Checking/Converting

helloNative() in C++

N-API ABI Stability



`helloNative();`

Node Runtime

JS Evaluate

N-API

V8 Engine

V8 Public API

Native Module

Type Checking/Converting

helloNative() in C++

N-API ABI Stability

- Provide N-API in Node API layer
- Define C/C++ types for the API which are independent from V8
- No data layout changes for earlier N-API

HOW TO IMPLEMENT N-API?

What should we do?

- Arguments length checking
- Get JavaScript values
- Type checking/converting
- Memory management

Define N-API version sum()

```
napi_value Sum(napi_env, napi_callback_info info) {
}
```

Basic N-API data types

- **napi_env**
    - The object is used to represent a context that the underlying N-API implementation can use to persist VM-specific state.
- **napi_callback_info**
    - The object representing the components of the JavaScript request being made. The object is usually created and passed by the Node.js runtime infrastructure.
- **napi_value**
    - This is an opaque pointer that is used to represent a JavaScript value.

Arguments length check

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

Arguments length check

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

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

Get Javascript values

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

double sum = 0;

for (int i = 0; i < length; i++) {
  // Calculate sum
}
```

Get Javascript values

```
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);

  // Calculate sum
}
```

Type checking

```
...
  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;
    }
  }
```

Type checking

```
...
  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;
    }
  }
```

Type converting

```
...
  for (int i = 0; i < length; i++) {
    ...

    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;
}
```

Type converting

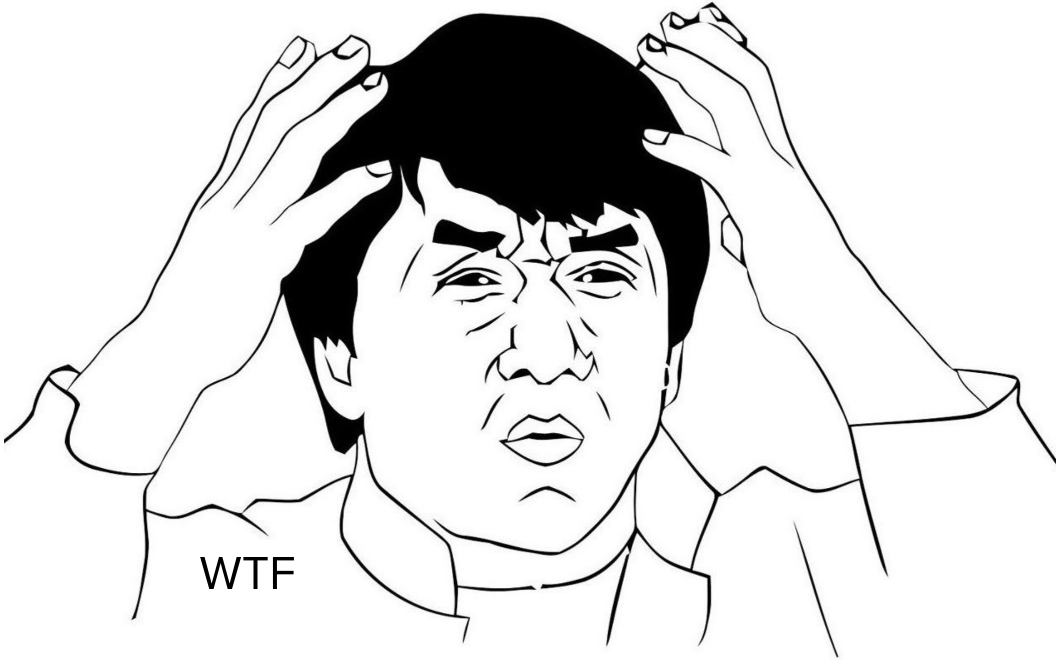
```
...
  for (int i = 0; i < length; i++) {
    ...

    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;
}
```

Memory management

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

  ...
}
```

Memory management

```
for (int i = 0; i < length; i++) {
  napi_handle_scope scope;
  napi_open_handle_scope(env, &scope);

  napi_value element;
  napi_get_element(env, i, &element);

  ...

  napi_close_handle_scope(env, scope);
}
```



WTF

WEB-IDL BINDING GENERATOR

What’s the problems?

- Arguments length checking
- Get JavaScript values
- Type checking/converting
- Memory management
- Readability

What’s the problems?

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

```
napi_value Sum(napi_env, napi_callback_info info) {
  napi_status status;
  size_t argc = 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)

≡

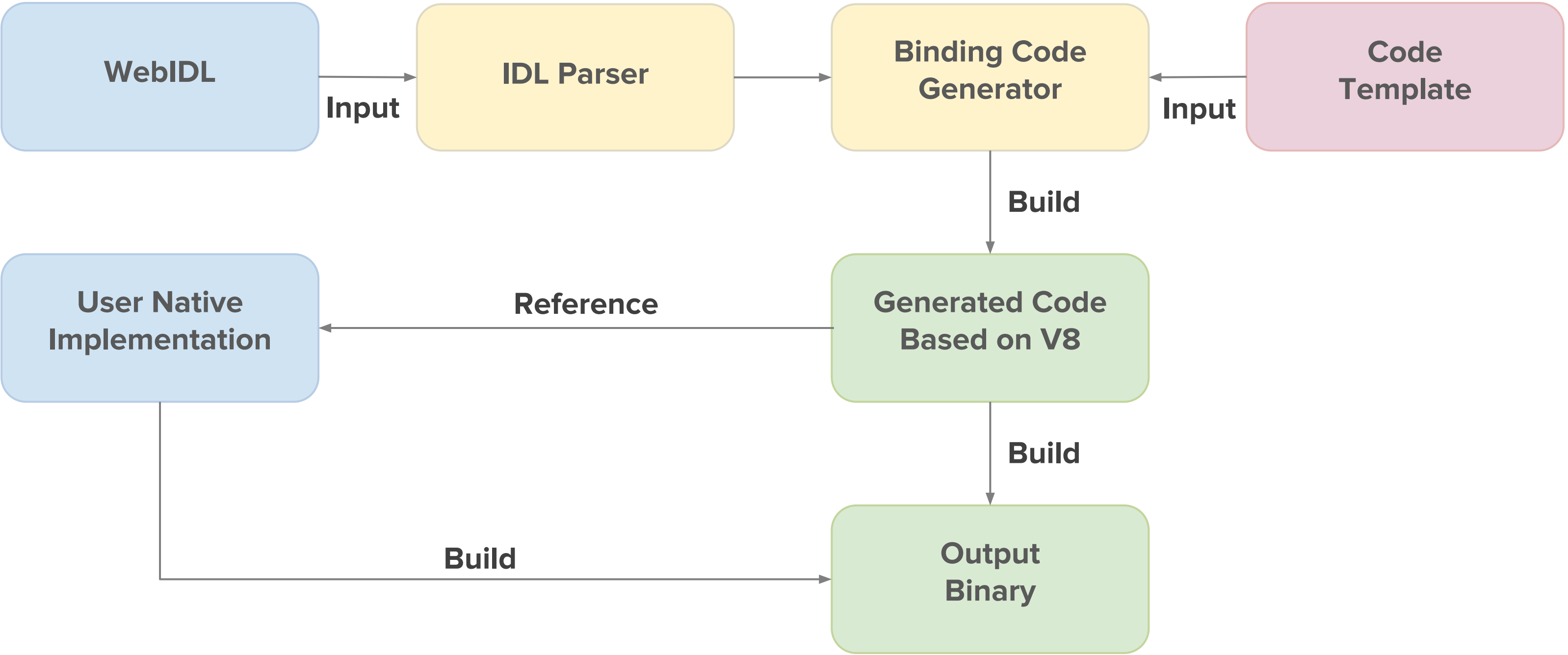
V8
(Javascript
Engine)

♥

Node.js
Native
Module

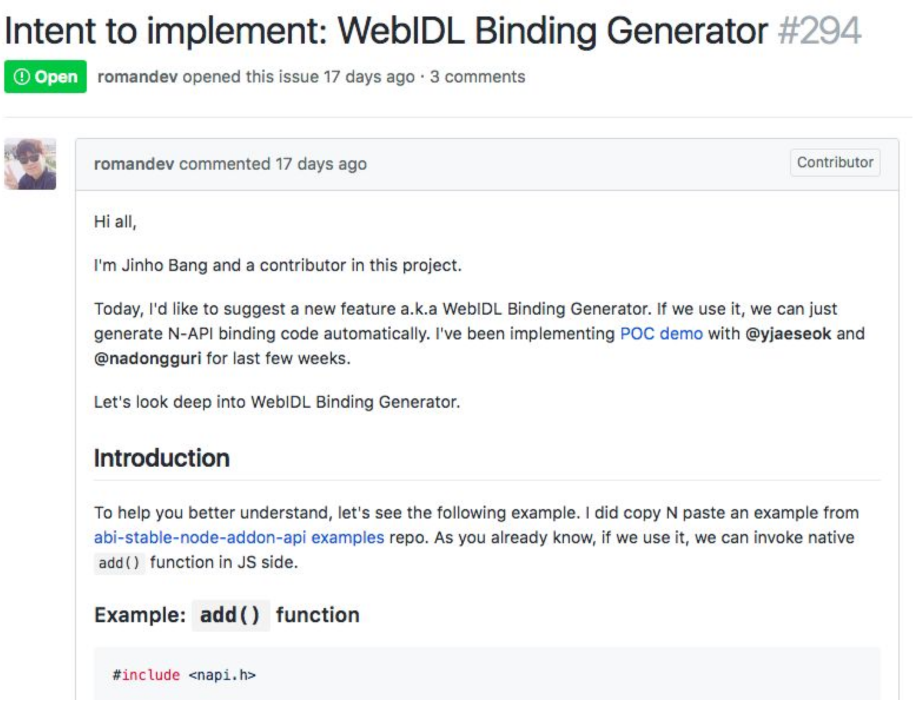
```
// WebIDL
[Constructor]
interface Calculator {
    double sum(sequence<long> elements);
};
```

```
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.

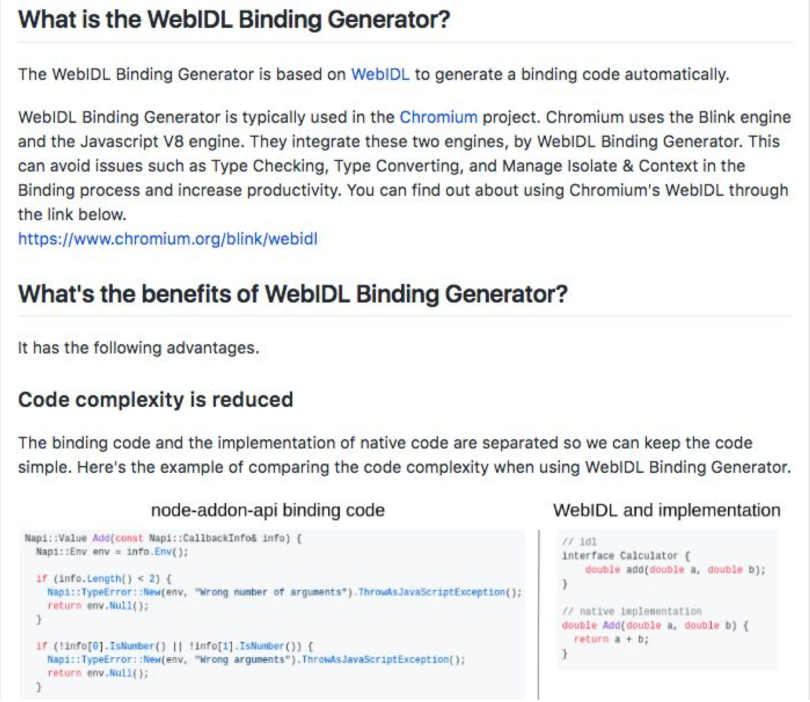
Let's look deep into WebIDL Binding Generator.

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;
}
```

\o/

Thank you