October 4-6, 2017 | Vancouver, BC

Node.js With Steroids

Make Better Node.js Application with Native Add-Ons

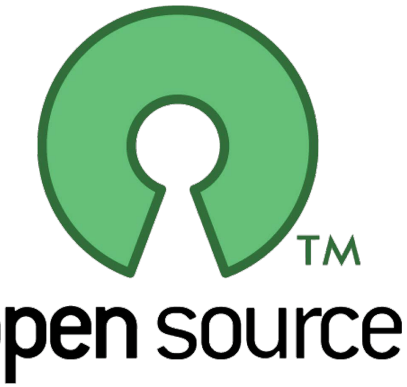
Nicola Del Gobbo, Developer, *Packly*

October 4-6, 2017 | Vancouver, BC

Who is Nicola Del Gobbo?

Developer

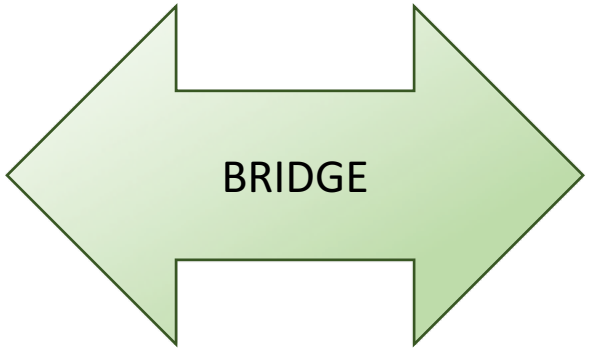
Contribute to



What is Node.js Native Addon?

In a very simple way Native Addons could be considered as
C / C++ code called from JavaScript

Native environment



JavaScript environment

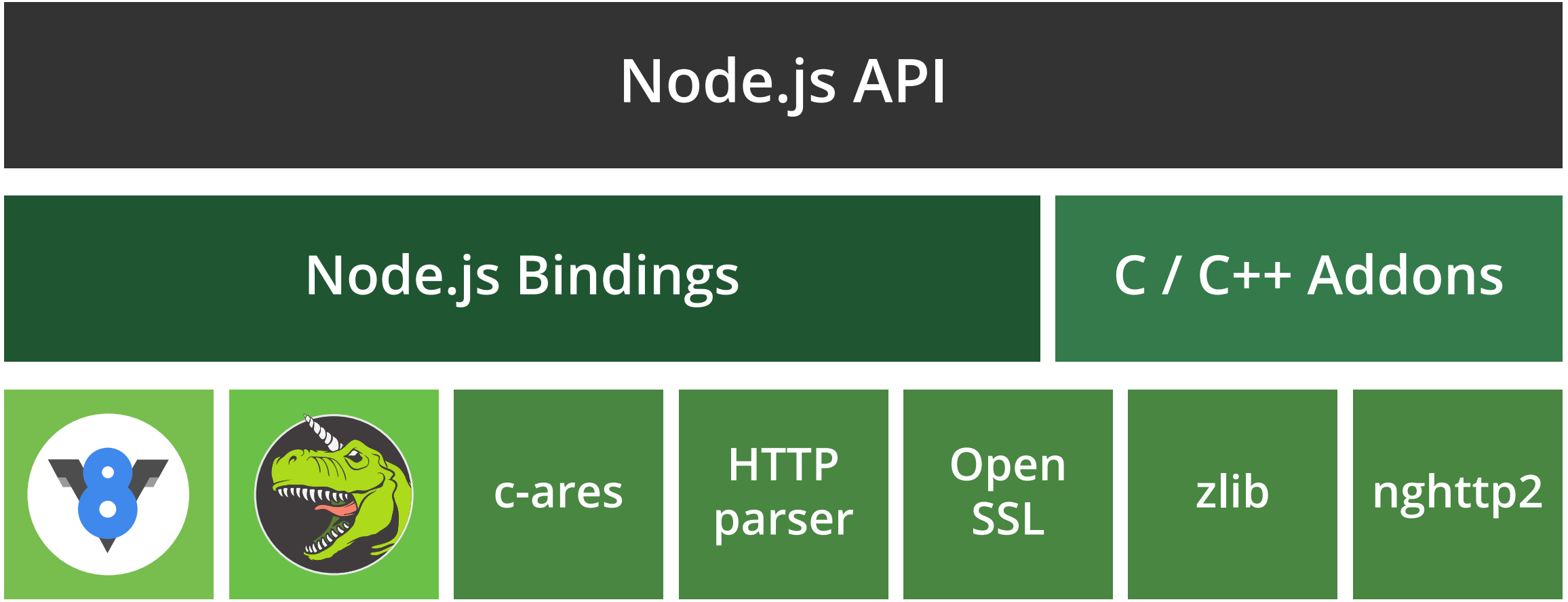
From Node.js documentation

*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.*

*They are used primarily to provide an **interface** between **JavaScript** running in Node.js and **C/C++** libraries.*

How is it possible?

All the **magic** is behind the **Node.js** architecture



Why should you implement new Native Add-on?

Performance

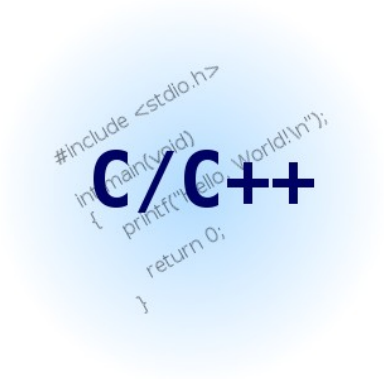
In general C / C++ code performs better than JavaScript code, but it's really true for **CPU bounds** operations

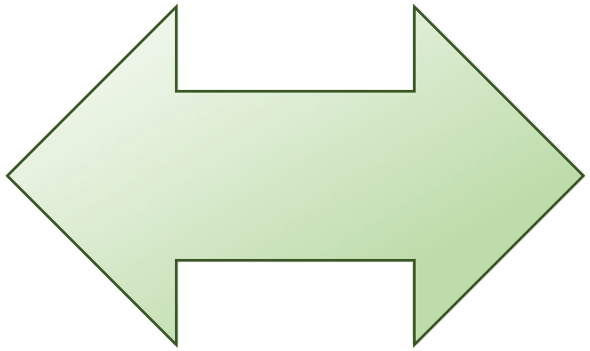
- **Image** processing (in average 6 times faster)
- **Video** processing
- **CRC** cyclic redundancy check (in average 125 time faster)
- **Compression**
- **Scientific calculus**
- **Algorithms that execute CPU heavy tasks**

Why should you implement new Native Add-on?

Integrate legacy application

You have the source code of an **old C / C++** application and want to expose something of its functionalities through the new Node.js application





Why should you implement new Native Add-on?

You don't find what fits your specific needs on **npm**

Sometimes completely reimplementing the module in **JavaScript** is not the right solution

Think at libraries like:

- **ImageMagick**
- **Ghostscript**
- **FFmpeg**
- **TensorFlow**

You have to concentrate on business logic of your application and not on developing a single module

Why should you implement new Native Add-on?

Desktop application

You want to use the local underlying **CPU** or **GPU** resources to execute the most complex tasks

- Better performance on the **local** side
- Better performance on the **server**

Es. **Electron** or **NW.js**

Why should you implement new Native Add-on?

Better error handling

Even if you will use **an old C / C++** library you will get an error code that explains the error that just happened.

In **new C / C++** library you have the **exception**

You don’t have to parse some string to identify if there was an error and what kind of it

Problems

Fragmentation API

The API to implement Native Add-Ons have been changed across different version of Node.js

Most of the changes were on **V8 API** and **ObjectWrap API**

0.8 - 0.10.x - 0.12.x - 1.x - 2.x - 3.x - 4.x - 5.x - 6.x - 7.x - 8.x

For more info http://v8docs.nodesource.com/

Problems

Fragmentation API

Need an adapter to stay compatible across different version of Node.js

- **NAN** - Native Abstraction for Node.js
  - **API compatibility**
  - Strong bonded with **V8 API**
  - You have to recompile your native add-ons switching to different version of Node.js

Problems

Write portable C / C++ code

At one point your native code **must compile** on different:

ARCHITECTURE

PLATFORM

COMPILER

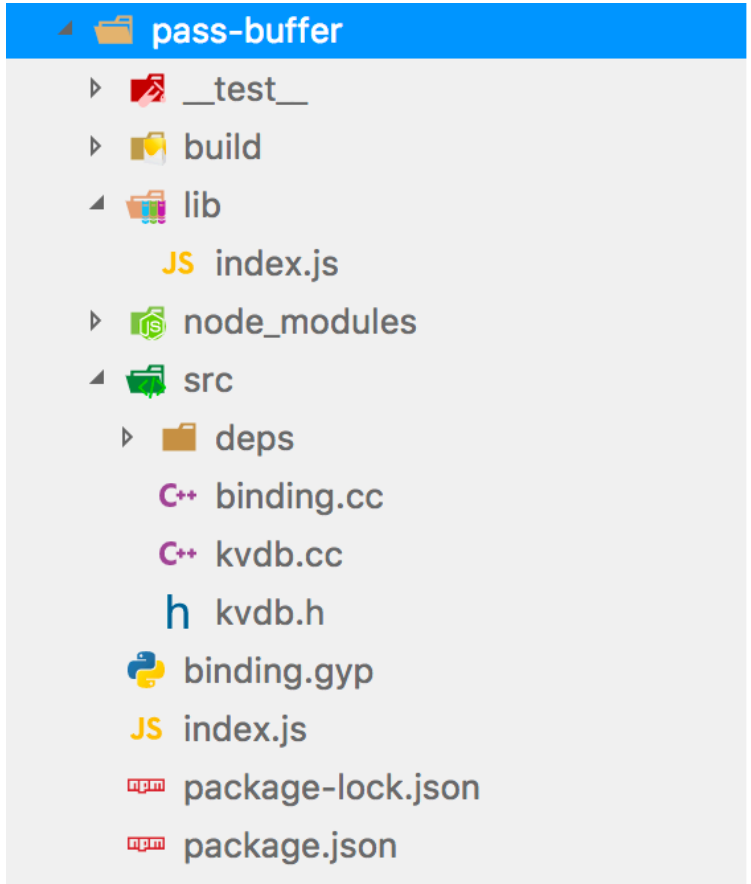
Problems

Documentation

- **C / C++** libraries that you are integrating are not well documented
- There are good **references** but not so much practical guide focusing on concepts about Native Add-Ons

node Interactive
October 4-6, 2017 | Vancouver, BC

How to organize your native Addon project



src C/C++ code and maybe dependencies

binding.gyp building configurations

lib JavaScript code

package.json

Example: the JavaScript part

```
'use strict';

var binary = require('node-pre-gyp');
var path = require('path');
var binding_path = binary.find(path.resolve(path.join(__dirname, './package.json')));
```

Import the Addon

```
var bindings = require(binding_path);

var crypto = require('crypto');

var promises = require('./lib/promises');

/// generate a salt (sync)
/// @param {Number} [rounds] number of rounds (default 10)
/// @return {String} salt
module.exports.genSaltSync = function genSaltSync(rounds) {
    // default 10 rounds
    if (!rounds) {
        rounds = 10;
    } else if (typeof rounds !== 'number') {
        throw new Error('rounds must be a number');
    }
```

Call the Addon's function

```
    return bindings.gen_salt_sync(rounds, crypto.randomBytes(16));
};
```

```
{
  "name": "conf-ni-2017-4",
  "description": "This is a project used as example for Node Interactive 2017",
  "version": "0.0.1",
  "private": true,
  "author": "Nicola Del Gobbo <nicoladelgobbo@gmail.com>",
  "keywords": [
    "c++",
    "native",
    "module",
    "nodejs",
    "addon",
    "v8"
  ],
  "license": "Apache-2.0",
  "scripts": {
    "test": "node node_modules/jasmine/bin/jasmine.js JASMINE_CONFIG_PATH=__test__/jasmine.json",
    "build": "node-gyp rebuild"
  },
  "engines": {
    "node": ">= 0.10.0"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/NickNaso/conf-ni-2017.git"
  },
  "bugs": {
    "url": "https://github.com/NickNaso/conf-ni-2017/issues"
  },
  "main": "index.js",
  "devDependencies": {
    "jasmine": "^2.8.0"
  },
```

npm will build the Addon automatically

```
  "gypfile": true,

  "dependencies": {
    "bindings": "^1.3.0",
    "nan": "^2.7.0"
  }
}
```

Example: the C / C++ part

```
NAN_METHOD(GenerateSaltSync) {
    Nan::HandleScope scope;

    if (info.Length() < 2) {
        Nan::ThrowTypeError("2 arguments expected");
        return;
    }

    if (!Buffer::HasInstance(info[1]) || Buffer::Length(info[1].As<Object>()) != 16) {
        Nan::ThrowTypeError("Second argument must be a 16 byte Buffer");
        return;
    }

    const int32_t rounds = Nan::To<int32_t>(info[0]).FromMaybe(0);
    u_int8_t* seed = (u_int8_t*)Buffer::Data(info[1].As<Object>());

    char salt[_SALT_LEN];
    bcrypt_gensalt(rounds, seed, salt);

    info.GetReturnValue().Set(Nan::Encode(salt, strlen(salt), Nan::BINARY));

}
```

Validate your input

Pass input values from JS context to C++

Execute elaboration on the C++ environment

Return value to JS context

```
NAN_MODULE_INIT(init) {
    Nan::Export(target, "gen_salt_sync", GenerateSaltSync);

    Nan::Export(target, "encrypt_sync", EncryptSync);
    Nan::Export(target, "compare_sync", CompareSync);
    Nan::Export(target, "get_rounds", GetRounds);
    Nan::Export(target, "gen_salt", GenerateSalt);
    Nan::Export(target, "encrypt", Encrypt);
    Nan::Export(target, "compare", Compare);
};

NODE_MODULE(bcrypt_lib, init);
```

Export function

Registration and initialization

Example: the binding.gyp

```
{
  'targets': [
    {
      'target_name': 'bcrypt_lib',
      'sources': [
        'src/blowfish.cc',
        'src/bcrypt.cc',
        'src/bcrypt_node.cc'
      ],
      'include_dirs' : [
          "<!(node -e \"require('nan')\")"
      ],
      'conditions': [
        [ 'OS=="win"', {
          'defines': [
            'uint=unsigned int',
          ],
        }],
      ],
    },
    {
      "target_name": "action_after_build",
      "type": "none",
      "dependencies": [ "<(module_name)" ],
      "copies": [
        {
          "files": [ "<(PRODUCT_DIR)/<(module_name).node" ],
          "destination": "<(module_path)"
        }
      ]
    }
  ]
}
```

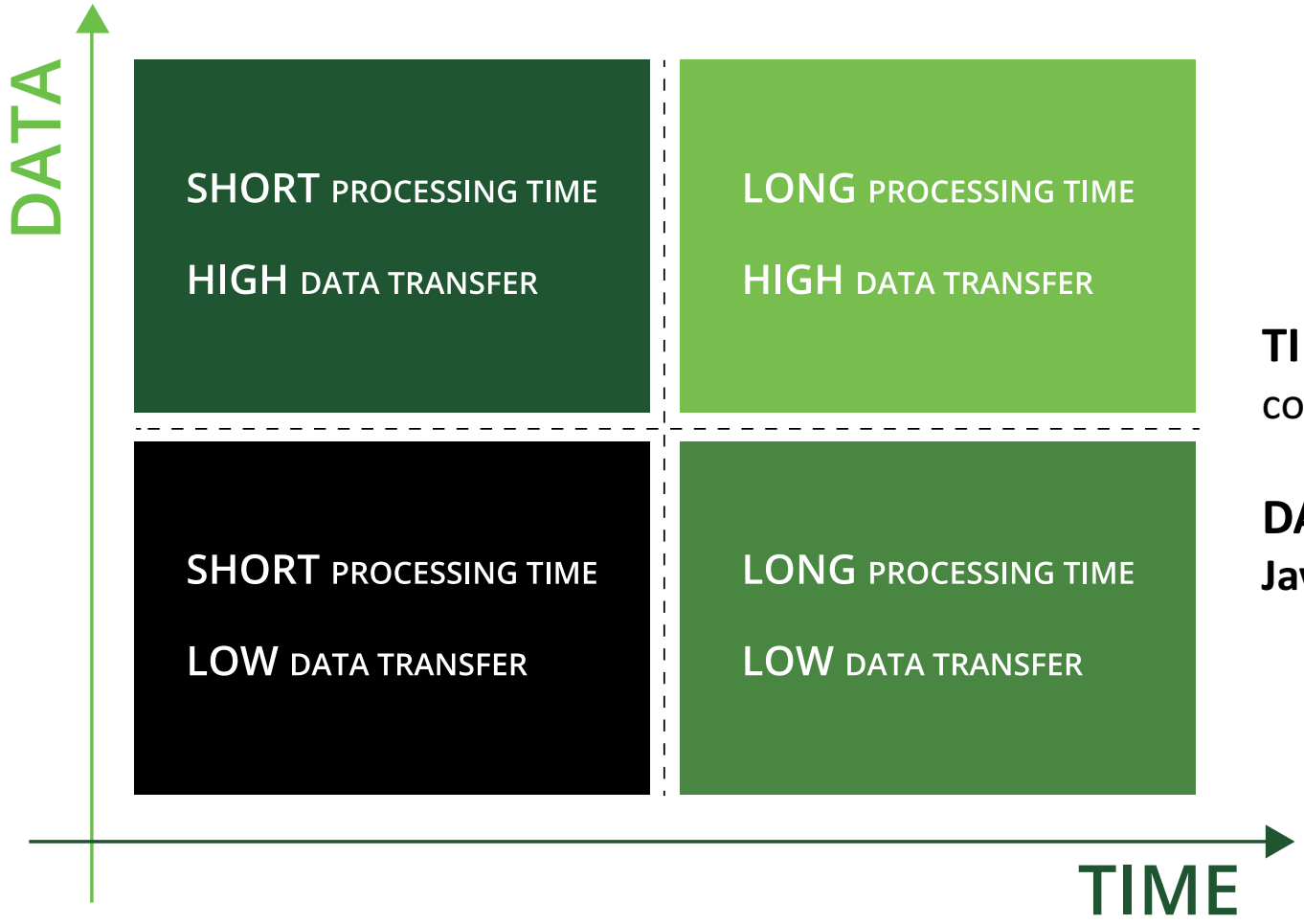
Name used to register the Addon

Sometimes finding the right **settings** for binding.gyp is not easy

Refer to **GYP** documentation
https://gyp.gsrc.io/docs/UserDocumentation.md

More useful examples to take inspiration from other developers
https://github.com/nodejs/node-gyp/wiki/”binding.gyp”-files-out-in-the-wild

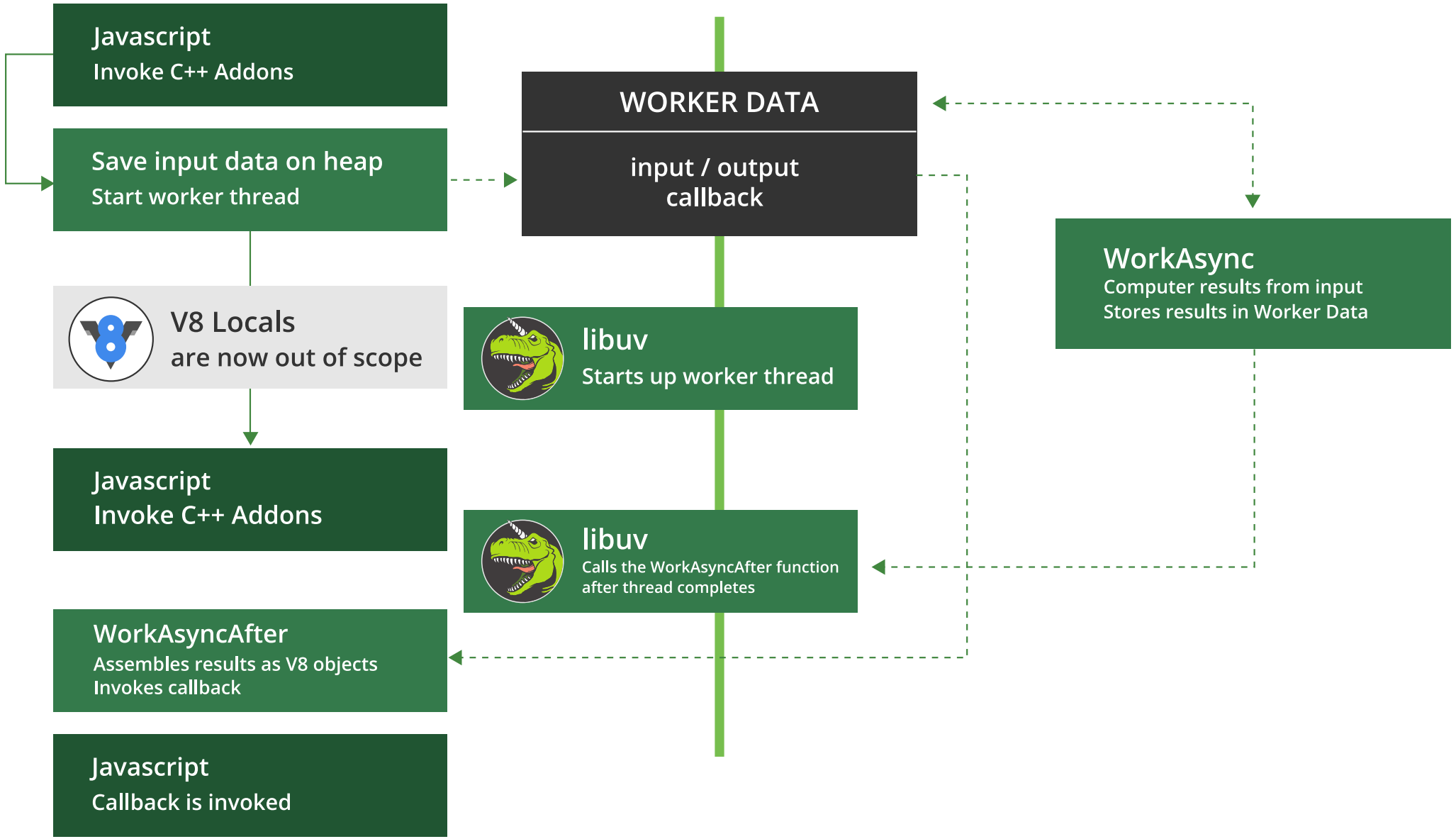
Which kind of Addon should you implement?



TIME: Amount of processing time spent on the **C++** code

DATA: Amount of data flowing between **C++** and **JavaScript**

The async way



Async example

```
class GhostscriptWorker : public AsyncWorker
{
  public:
    GhostscriptWorker(Callback *callback, string RAWcmd)
        : AsyncWorker(callback), RAWcmd(RAWcmd), res(0) {}
    ~GhostscriptWorker() {}

    void Execute()
    {
        res = 0;
        vector<string> explodedCmd;
        istringstream iss(RAWcmd);
        for (string RAWcmd; iss >> RAWcmd;)
            explodedCmd.push_back(RAWcmd);
        int gsargc = static_cast<int>(explodedCmd.size());
        char **gsargv = new char *[gsargc];
        for (int i = 0; i < gsargc; i++)
        {
            gsargv[i] = (char *)explodedCmd[i].c_str();
        }
        try
        {
            GhostscriptManager *gm = GhostscriptManager::GetInstance();
            gm->Execute(gsargc, gsargv);
            delete[] gsargv;
            res = 0;
        }
        catch (exception &e)
        {
            delete[] gsargv;
            msg << e.what();
            res = 1;
        }
    }

    void HandleOKCallback()
    {
        Nan::HandleScope();
        Local<Value> argv[1];
        if (res == 0)
        {
            argv[0] = Null();
        }
        else
        {
            argv[0] = Error(Nan::New<String>(msg.str()).ToLocalChecked());
        }
        callback->Call(1, argv);
    }

  private:
    string RAWcmd;
    int res;
    stringstream msg;
};
```

This code will be executed on the Worker Thread

When execute method end its computation the resuls will be reassembled and passed to JS context

```
NAN_METHOD(Execute)
{
    Callback *callback = new Callback(info[1].As<Function>());
    Local<String> JScmd = Local<String>::Cast(info[0]);
    AsyncQueueWorker(new GhostscriptWorker(callback, *String::Utf8Value(JScmd)));
    info.GetReturnValue().SetUndefined();
}
```

Return immediately

Buffer example

```
void buffer_delete_callback(char* data, void* hint) {
  free(data);
}
```

Callback executed to free the memory allocated for the Buffer

Nicola Del Gobbo, 5 days ago

```
class GetKeyBufferWorker: public Nan::AsyncWorker {
  public:
    GetKeyBufferWorker(Nan::Callback *callback, vedis *db, std::string key)
      :AsyncWorker(callback), db(db), key(key) {}
    ~GetKeyBufferWorker() {}

    void Execute() {
      int rc;

      std::stringstream strlenCmd;
      strlenCmd << "STRLEN " + key;
      rc = vedis_exec(db, (strlenCmd.str()).c_str(), -1);
      if(rc != VEDIS_OK) {
        // Handle error
      }
      /* Extract the return value of the last executed command (i.e. 'STRLEN test') " */
      vedis_value *strlen_result;
      vedis_exec_result(db, &strlen_result);
      /* Cast the vedis object to a string */
      this->buffer_length = vedis_value_to_int(strlen_result);

      std::stringstream get_cmd;
      get_cmd << "GET " + key;
      rc = vedis_exec(db, (get_cmd.str()).c_str(), -1);
      if(rc != VEDIS_OK) {
        // Handle error
      }
      /* Extract the return value of the last executed command (i.e. 'GET test') " */
      vedis_value *get_result;
      vedis_exec_result(db, &get_result);
      /* Cast the vedis object to a string */
      //this->buffer = static_cast<char *>(malloc(this->buffer_length));
      this->buffer = const_cast<char *>(vedis_value_to_string(get_result, 0));
    }

    void HandleOKCallback() {
      Nan::HandleScope();
      int argc = 2;
      Local<Value> argv[2];
      argv[0] = Nan::Null();
      // Not efficient solution to free memory
      //argv[1] = Nan::CopyBuffer(this->buffer, this->buffer_length).ToLocalChecked();
      Local<Object> data =
      Nan::NewBuffer(this->buffer, this->buffer_length, buffer_delete_callback, this->buffer).ToLocalChecked();
      argv[1] = data;
      callback->Call(argc, argv);
    }
  private:
    vedis *db;
    std::string key;
    char *buffer;
    int buffer_length;
};
```

GET command on database return a buffer

Buffers are Javascript objects, whose data is stored outside V8 anyways they are under V8's control

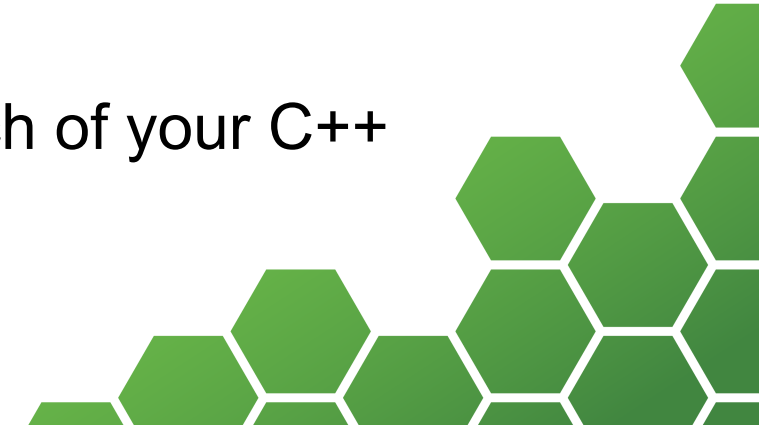
When V8 destroy the buffer we need some way to free the memory

```
NAN_PROPERTY_GETTER(Tensor::DataGetter) {
  TensorflowNode::Tensor* tensor = ObjectWrap::Unwrap<TensorflowNode::Tensor>(info.This());
  void* data = TF_TensorData(tensor->_tensor);
  size_t len = TF_TensorByteSize(tensor->_tensor);
  // NewBuffer causes data to transfer internally, for example, when v8's GC collects this
  // ArrayBuffer object, `data` would also be freed.
  // The `data` is actually freed by tensorflow internally, so here CopyBuffer is the better way.
  Local<Object> buffer = Nan::CopyBuffer((char*)data, len).ToLocalChecked();
  info.GetReturnValue().Set(buffer);
}
```

Sometimes the only way is to copy the buffer

ObjectWrap API

- **ObjectWrap** is a way to expose your C++ code to JavaScript
- You have to extend **ObjectWrap** class that includes the plumbing to connect JavaScript code to a C++ object
- Classes extending **ObjectWrap** can be instantiated from JavaScript using the **new** operator, and their methods can be **directly invoked from JavaScript**
- Unfortunately, the ***wrap*** part really refers to a way to group methods and state, not decorate existing classes
- It's **your responsibility write custom code to bridge** each of your C++ class methods.



ObjectWrap example

```
NAN_MODULE_INIT(Database::Init) {
  Local<FunctionTemplate> tpl = Nan::New<FunctionTemplate>(New);
  tpl->SetClassName(Nan::New("Database").ToLocalChecked());
  tpl->InstanceTemplate()->SetInternalFieldCount(1);
```

Initialization code

Set prototype methods

```
  Nan::SetPrototypeMethod(tpl, "getKey", GetKey);
  Nan::SetPrototypeMethod(tpl, "getKeyBuffer", GetKeyBuffer);
  Nan::SetPrototypeMethod(tpl, "getKeySync", GetKeySync);
  Nan::SetPrototypeMethod(tpl, "putKey", PutKey);
  Nan::SetPrototypeMethod(tpl, "putKeyBuffer", PutKeyBuffer);
  Nan::SetPrototypeMethod(tpl, "putKeySync", PutKeySync);
```

Set accessor methods

```
  Local<ObjectTemplate> itpl = tpl->InstanceTemplate();
  Nan::SetAccessor(itpl, Nan::New("db_name").ToLocalChecked(), DbName);
```

Update constructor reference

```
  constructor().Reset(v8::Isolate::GetCurrent(), Nan::GetFunction(tpl).ToLocalChecked());

  Nan::Set(target, Nan::New("Database").ToLocalChecked(), Nan::GetFunction(tpl).ToLocalChecked());
}
```

```
NAN_METHOD(Database::New) {
  // Here we need some control
  String::Utf8Value tmpDbName(info[0]->ToString());
  std::string dbName(*tmpDbName);
  if (info.IsConstructCall()) {
    KVDB::Database *database = new KVDB::Database(dbName);
    database->Wrap(info.This());
    info.GetReturnValue().Set(info.This());
  } else {
    const int argc = 1;
    Local<Value> argv[argc] = {info[0]};
    Local<Function> cons = Nan::New(constructor());
    info.GetReturnValue().Set(Nan::NewInstance(cons, argc, argv).ToLocalChecked());
  }
}
```

Use the new operator

Call as normal function

```
NAN_METHOD(Database::GetKey) {
    // Here we need some control
    String::Utf8Value tmpKey(info[0]->ToString());
    Nan::Callback *callback = new Nan::Callback(info[1].As<Function>());
    std::string key(*tmpKey);
    std::stringstream cmd;
    cmd << "GET " + key;
    KVDB::Database* database = ObjectWrap::Unwrap<KVDB::Database>(info.This());
    AsyncQueueWorker(new GetKeyWorker(callback, database->db, cmd.str()));
    info.GetReturnValue().SetUndefined();
}
```

Prototype method

```
NAN_GETTER(Database::DbName) {
  KVDB::Database* database = ObjectWrap::Unwrap<KVDB::Database>(info.This());
  info.GetReturnValue().Set(Nan::New(database->db_name).ToLocalChecked());
}
```

Accessor method

```
const mydb = new Database('test')

mydb.db_name

mydb.getKey('username', function(err, value) {
    if (err) {
        console.error(err)
    } else {
        console.log('Value from my async API')
        console.log(value)
    }
})
```

Lesson learned from my experience

- Create **simple interface** between C / C++ and JavaScript
- Insert **complexity** on the JavaScript side
- Use **ObjectWrap** API to return C / C++ object to JavaScript
- Validate the **types** on the native side
- Expose **only the functionalities you need**
- Don’t block the event loop, stay **asynchronous**: this is a must if you will use the Addon on a distributed system
- Use **Buffer** to pass big quantity of data
- If you have many parameters use **JSON** to **pass** and **parse** data
- Consider the using of **protobuf**

Present and future

- The largest number of native addons is written using **NAN**
- **NAN** helps stay compatible with **old versions** of Node.js
- **N-API** will be a game changer on the native addon development
  - **ABI** compatibility
  - New **ES6** types
  - **Isolated** from V8 API
  - C / C++
  - Conversion tool that helps to migrate from **NAN**
  - Great **developer experience** for mantainers

LITTLE EXAMPLE

Implement high performance key value database

Binding to **Vedis** an embeddable datastore C library

```
const mydb = new Database('test')

mydb.db_name
```

```
mydb.putKeySync('username', 'NickNaso')
mydb.getKeySync('username')
```

```
mydb.putKey('password', 'keeplooking', function (err) {
    if (err) {
        console.error(err)
    } else {
        console.log("Value stored")
    }
})

mydb.getKey('username', function(err, value) {
    if (err) {
        console.error(err)
    } else {
        console.log('Value from my async API')
        console.log(value)
    }
})
```

```
const buffer = Buffer.from('qwertyuiopasdfghjklzxcvbnm1234567890')

mydb.putKeyBuffer('image', buffer, function (err) {
    if (err) {
        console.error(err)
    } else {
        console.log('Buffer stored')

    }
})

mydb.getKeyBuffer('image', function (err, buffer) {
    if (err) {
        console.error(err)
    } else {
        console.log(buffer.toString())
    }
})
```

Thanks!

nicoladelgobbo@gmail.com

@NickNaso on Twitter

All examples and materials are here

https://github.com/NickNaso/conf-ni-2017