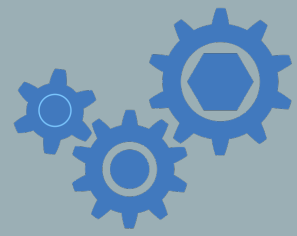


IBM Runtime Technologies

NODE.JS NATIVE ADDONS

A presentation by John Barboza, Muntasir Mallick, and Anisha Rohra

Trademarks, Copyrights, Disclaimers

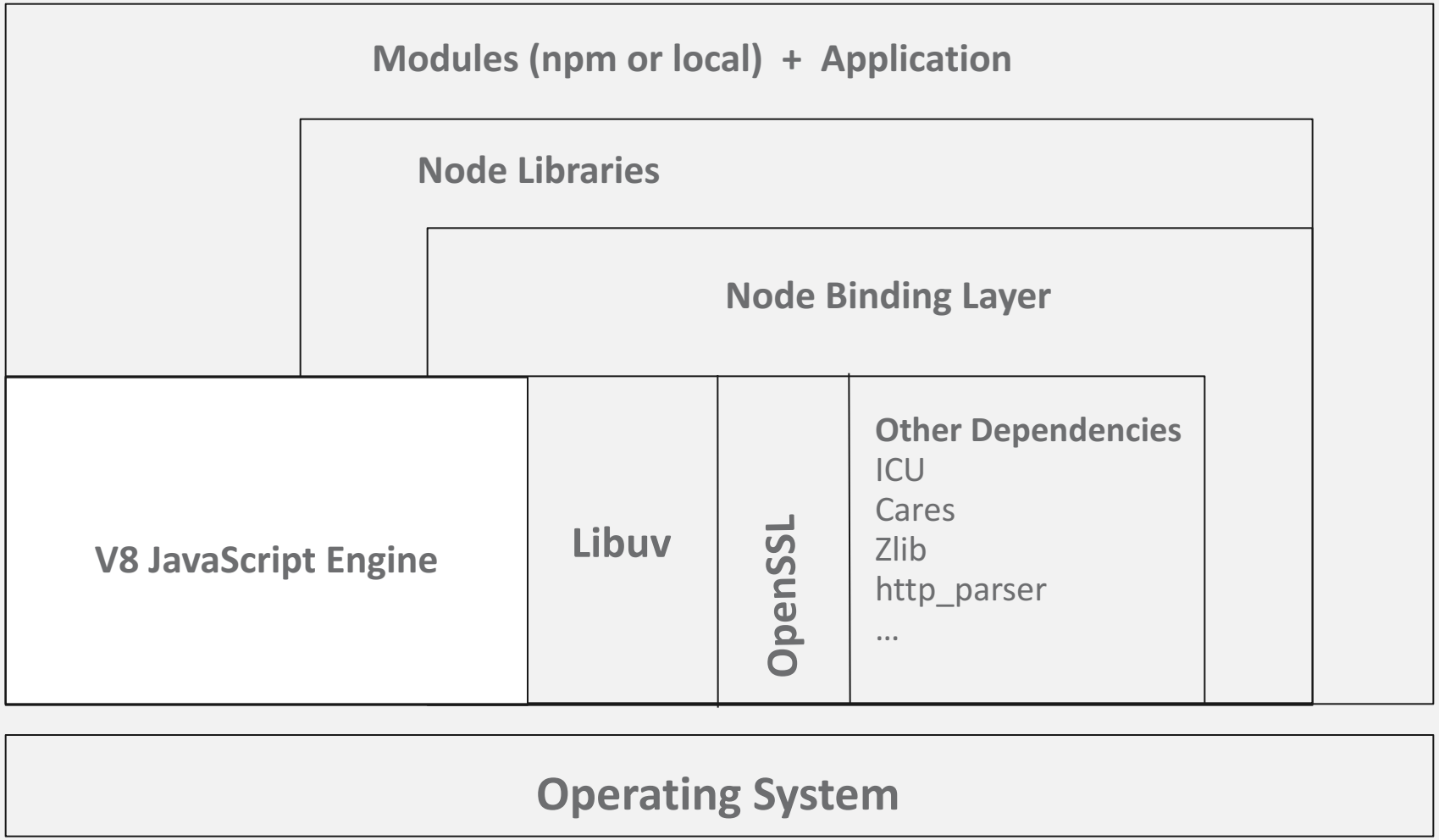
IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.

WHAT IS NODE.JS

- Server-side JavaScript platform
- An open source server framework
- Goal: Efficiently build fast, scalable, 'real-time' network applications
- Runs on various platforms (Windows, Linux, AIX, Mac OS X, zOS, etc.)

WHY USE NODE.JS

- Back-end APIs and services – microservices
- IoT and “Realtime” applications


Modules (npm or local) + Application
Node Libraries
Node Binding Layer
V8 JavaScript Engine
Libuv
OpenSSL
Other Dependencies
ICU
Cares
Zlib
http_parser
…
Operating System

APPROACHING NATIVE ADDONS

What? Why? How?

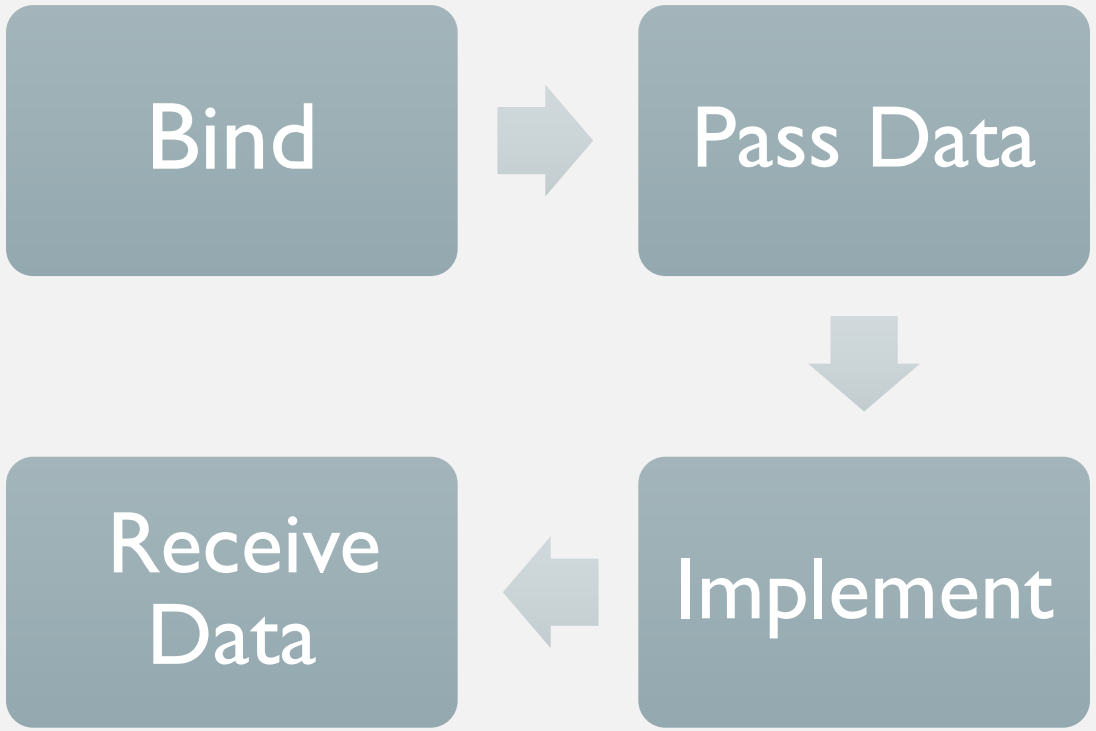
WHAT IS A NATIVE ADDON

- Connect JavaScript to C++
- Work with direct access to V8 APIs
- Dynamically linked C/C++ libraries

WHY USE NATIVE ADDONS

- Provide access to existing C++ libraries that JavaScript cannot access
- Better Performance for modules that would generally work faster in C/C++ than in JavaScript
- Example: Platform specific code is benefitted by having access to C/C++ abilities.

HOW TO MAKE A NATIVE ADDON



LET'S ADD 5

Your first native addon.

my_addon.js

FIRST, THE JAVASCRIPT

```
var addon = require('bindings')('my_addon');

var sum = addon.add5(10);
console.log(sum);
```

my_addon.cc

THEN, THE C++

```
#include <node.h>
#include <v8.h>

using namespace v8;

void Add5Method(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* v8_engine = Isolate::GetCurrent();
  HandleScope scope(v8_engine);

  double value_given = args[0]->NumberValue();
  double value_added = value_given + 5;

  Local<Number> return_value = Number::New(v8_engine, value_added);
  args.GetReturnValue().Set(return_value);
}

void Add5Init(Handle<Object> exports) {
  Isolate* v8_engine = Isolate::GetCurrent();

  Local<String> js_func = String::NewFromUtf8(v8_engine, "add5");
  Local<Function> c_func = FunctionTemplate::New(v8_engine, Add5Method)->GetFunction();

  exports->Set(js_func, c_func);
}

NODE_MODULE(my_addon, Add5Init)
```

my_addon.cc

BIND

```
void Add5Init(Handle<Object> exports) {
  Isolate* v8_engine = Isolate::GetCurrent();

  Local<String> js_func = String::NewFromUtf8(v8_engine, "add5");
  Local<Function> c_func = FunctionTemplate::New(v8_engine, Add5Method)->GetFunction();

  exports->Set(js_func, c_func);
}

NODE_MODULE(my_addon, Add5Init)
```

my_addon.cc

BIND

```
void Add5Init(Handle<Object> exports) {




  exports->Set(js_func, c_func);
}

NODE_MODULE(my_addon, Add5Init)
```

my_addon.cc

BIND

```
void Add5Init(Handle<Object> exports) {
  Isolate* v8_engine = Isolate::GetCurrent();

  Local<String> js_func = String::NewFromUtf8(v8_engine, "add5");
  Local<Function> c_func = FunctionTemplate::New(v8_engine, Add5Method)->GetFunction();

  exports->Set(js_func, c_func);
}

NODE_MODULE(my_addon, Add5Init)
```

my_addon.cc

PASS DATA

```
void Add5Method(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* v8_engine = Isolate::GetCurrent();
  HandleScope scope(v8_engine);
  double value_given = args[0]->NumberValue();




}
```

my_addon.cc

IMPLEMENT

```
void Add5Method(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* v8_engine = Isolate::GetCurrent();
  HandleScope scope(v8_engine);
  double value_given = args[0]->NumberValue();

  double value_added = value_given + 5;
  Local<Number> return_value = Number::New(v8_engine, value_added);


}
```

my_addon.cc

RECEIVE DATA

```
void Add5Method(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* v8_engine = Isolate::GetCurrent();
  HandleScope scope(v8_engine);
  double value_given = args[0]->NumberValue();

  double value_added = value_given + 5;
  Local<Number> return_value = Number::New(v8_engine, value_added);

  args.GetReturnValue().Set(return_value);
}
```

my_addon.cc

```
#include <node.h>
#include <v8.h>

using namespace v8;

void Add5Method(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* v8_engine = Isolate::GetCurrent();
  HandleScope scope(v8_engine);

  double value_given = args[0]->NumberValue();
  double value_added = value_given + 5;

  Local<Number> return_value = Number::New(v8_engine, value_added);
  args.GetReturnValue().Set(return_value);
}

void Add5Init(Handle<Object> exports) {
  Isolate* v8_engine = Isolate::GetCurrent();

  Local<String> js_func = String::NewFromUtf8(v8_engine, "add5");
  Local<Function> c_func = FunctionTemplate::New(v8_engine, Add5Method)->GetFunction();

  exports->Set(js_func, c_func);
}

NODE_MODULE(my_addon, Add5Init)
```

BUILDING NATIVE ADDONS

Let's run this thing.

LET'S COMPILE!

- **Run** `npm install`.

WHAT IS NPM

- Node Package Manager
- Online repository for the publishing of open-source Node.js projects
- A command-line utility for package installation, version management, and dependency management

PACKAGE.JSON

- The dependency description for the overall project in JSON format.
- Only difference is the following line which must be included:

`"gypfile": true`

- This indicates to npm that there is a binding.gyp file present and therefore that this is a native addon module.

```
{
  "name": "my_addon",
  "version": "1.0.0",
  "description": "cascon 2017 native addon demo",
  "main": "my_addon.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "bindings": "~1.2.1"
  },
  "author": "IBM Runtimes",
  "license": "ISC",
  "gypfile": true
}
```

BINDING.GYP

- This file is the build definition for the C++ part of the node module.

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

LET'S RUN!

- **Run** `node my_addon.js`

HELLO WORLD

Your turn!

my_addon.cc

```
#include <node.h>
#include <v8.h>

using namespace v8;

void Add5Method(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* v8_engine = Isolate::GetCurrent();
  HandleScope scope(v8_engine);

  double value_given = args[0]->NumberValue();
  double value_added = value_given + 5;

  Local<Number> return_value = Number::New(v8_engine, value_added);
  args.GetReturnValue().Set(return_value);
}

void Add5Init(Handle<Object> exports) {
  Isolate* v8_engine = Isolate::GetCurrent();

  Local<String> js_func = String::NewFromUtf8(v8_engine, "add5");
  Local<Function> c_func = FunctionTemplate::New(v8_engine, Add5Method)->GetFunction();

  exports->Set(js_func, c_func);
}

NODE_MODULE(my_addon, Add5Init)
```

HELLO WORLD INIT

```
void HelloWorldInit(Handle<Object> exports) {
  Isolate* v8_engine = Isolate::GetCurrent();

  Local<String> js_func = String::NewFromUtf8(v8_engine, "hello");
  Local<Function> c_func = FunctionTemplate::New(v8_engine, HelloWorldMethod)->GetFunction();

  exports->Set(js_func, c_func);
}

NODE_MODULE(hello_addon, HelloWorldInit)
```

HELLO WORLD METHOD

```
void HelloWorldMethod(const v8::FunctionCallbackInfo<Value>& args) {
  Isolate* v8_engine = Isolate::GetCurrent();
  HandleScope scope(v8_engine);

  Local<String> hello_string = String::NewFromUtf8(v8_engine, "hello world");
  args.GetReturnValue().Set(hello_string);
}
```

LET'S BUILD

- **Run** `npm install`
- **Run** `node hello.js`

CALLBACKS

Let’s do one more.

FIRST, THE JAVASCRIPT

```
var addon = require('bindings')('addon');

addon.callback(function(msg){
  console.log(msg); // 'hello world'
});
```

BIND

```
void Init(Handle<Object> exports, Handle<Object> module) {
  Isolate* isolate = Isolate::GetCurrent();

  Local<String> js_func = String::NewFromUtf8(isolate, "callback");
  Local<Function> c_func = FunctionTemplate::New(isolate, RunCallback)->GetFunction();

  exports->Set(js_func, c_func);
}

NODE_MODULE(addon, Init)
```

PASS DATA

```
void RunCallback(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  Local<Function> cb = Local<Function>::Cast(args[0]);




}
```

IMPLEMENT

```
void RunCallback(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  Local<Function> cb = Local<Function>::Cast(args[0]);

  const unsigned argc = 1;
  Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello world") };


}
```

RECEIVE DATA

```
void RunCallback(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  Local<Function> cb = Local<Function>::Cast(args[0]);

  const unsigned argc = 1;
  Local<Value> argv[argc] = { String::NewFromUtf8(isolate, "hello world") };

  cb->Call(isolate->GetCurrentContext()->Global(), argc, argv);
}
```

LET'S BUILD

- **Run** `npm install`
- **Run** `node addon.js`

A MOTIVATION

Why native addons are important and powerful.

PWUID

- A native addon that allows Node.js access to computer information such as the username, name, home directory, shell, and gid from uid.
- Written using NAN
- A simple native addon that exposes Node.js to information that it would otherwise not have access to.

```
const pwuid = require('pwuid');

console.log(pwuid());
```

LET'S BUILD PWUID

- Run `npm install`
- Run `node test.js`

```
{ name: 'anisharohra',
  uid: 501,
  gid: 20,
  gecos: 'Anisha Rohra',
  dir: '/Users/anisharohra',
  shell: '/bin/bash' }
```

NANOMSG

- A more sophisticated native addon, featuring multiple C++ files that implement complex objects and classes
- Export these objects/classes to Node.js for the purposes of creating a socket and sending messages.
- Also written using NAN but a NAPI version is maintained as well for the purposes of providing a side-by-side example of the differences between the two methods of writing a native addon.

```
var nano = require('nanomsg');

var pub = nano.socket('pub');
var sub = nano.socket('sub');

var addr = 'tcp://127.0.0.1:7789'
pub.bind(addr);
sub.connect(addr);

sub.on('data', function (buf) {
  console.log(String(buf));
  pub.close();
  sub.close();
});

setTimeout(function () {
  pub.send("Hello from nanomsg!");
}, 100);
```

LET'S BUILD NANOMSG

- **Run** `npm install`
- **Run** `node test.js`

```
Hello from nanomsg!
```

OTHER WAYS TO NATIVE ADDON

NODE ADDON	NAN	NAPI
Built in with node	Must be installed with npm	Must be installed with npm
Uses node.h	Uses nan.h	Uses napi.h
The original C++ API	An abstraction meant to simplify writing native addons	An abstraction meant to remain stable between different node versions
Requires frequent changes between node versions.	Abstracts the changes between the node versions so that less frequent changes need to be made. However, the original binaries are not stable.	Ensures that the binaries are stable between node versions.

THANK YOU FOR PARTICIPATING

If you have any questions, feel free to ask us.