

A low-angle, perspective shot of a cable-stayed bridge under construction. The bridge deck is a wide, flat concrete surface. Several large, white, cylindrical stay cables are visible, extending from the top of a dark, steel pylon down to the deck. A construction crane is perched atop the pylon. The sky is a clear, pale blue. The text "Build the C link" is overlaid in large, bold, yellow letters, and "Nikhil Marathe" is written below it in smaller, white letters.

Build the C link

Nikhil Marathe

Introduction

- V8 is a powerful, **fast** JavaScript engine
- It is self contained and easy to embed
 - JS is the new Lua?
- node.js is a thin wrapper around V8 and evented I/O (*libuv*)

Code

```
git clone git://github.com/nikhilm/jsfoo-pune-2012.git
```

We want to

Use C/C++ libraries in node.js

Exchange data between C++ \Leftrightarrow JS

Do asynchronous I/O

Getting started

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

using namespace v8;

extern "C" {
    static void Init(Handle<Object> target) {
    }
    NODE_MODULE(firststep, Init)
}
```

Build

```
# firststep/wscript
import Options

def set_options(opt):
    opt.tool_options("compiler_cxx")

def configure(conf):
    conf.check_tool("compiler_cxx")
    conf.check_tool("node_addon")

def build(bld):
    obj = bld.new_task_gen("cxx", "shlib", "node_addon")
    obj.target = "firststep"
    obj.source = "firststep.cc"
```

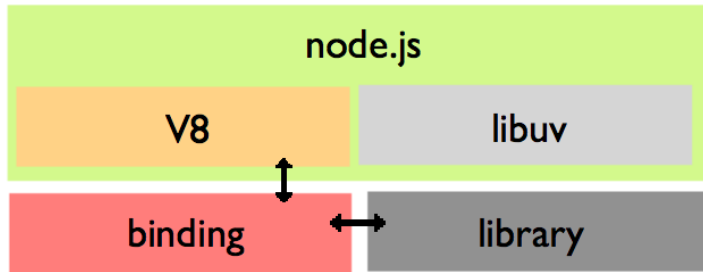
Run

```
$ node-waf configure build
...
'build' finished successfully (0.327s)

$ node
```

```
> require('./build/Release/firststep')
{}
```

Architecture



Handles

- Think of them as **smart pointers**, GCed by V8
- Also encode scope (Use *HandleScope* to manage handles)
- Local - GCed as they go out of scope:

```
Local<String> name; // also Handle<...>
```

- Persistent - Must be manually disposed:

```
Persistent<String> globalVariable;
```

Injecting primitives

```
#include <math.h>
#include <v8.h>
#include <node.h>
#include <node_version.h>
using namespace v8;

extern "C" {
    static void Init(Handle<Object> target) {
        target->Set(String::NewSymbol("pi"),
                    Number::New(M_PI));

        NODE_DEFINE_CONSTANT(target, NODE_MINOR_VERSION);

        target->Set(String::New("name"), String::New("Nikhil"));
    }
    NODE_MODULE(primitives, Init)
}
```

Simple functions

```
exports.square = function(n) {  
    return n * n;  
}
```

We want to do this in C++

Simple functions

Registering with V8:

```
Handle<Value> Square(const Arguments &args)
```

```
static void Init(Handle<Object> target) {  
    HandleScope scope;  
  
    Handle<FunctionTemplate> squareTpl =  
        FunctionTemplate::New(Square);  
  
    target->Set(String::New("square"),  
               squareTpl->GetFunction());  
}
```

Simple functions

Implementation:

```
Handle<Value> Square(const Arguments &args) {  
    HandleScope scope;  
  
    int a = args[0]->Int32Value();  
    int sq = a * a;  
  
    return scope.Close(Integer::New(sq));  
}
```

explain scope.Close

Templates

FunctionTemplate	???
FunctionTemplate::GetFunction	square [Function]
FunctionTemplate::InstanceTemplate	What <code>this`</code> would be <code>in</code> <code>'new square()'</code>
FunctionTemplate::PrototypeTemplate	square.prototype

Simple objects

```
exports.Inventory = function() {  
    this.items = 257;  
}
```

```
// later
```

```
var iv = new Inventory();  
console.log(iv.items);
```

This is the classic object oriented JS style

Simple objects

```
static void Init(Handle<Object> target) {  
    HandleScope scope;  
  
    Handle<FunctionTemplate> inventoryTpl =  
        FunctionTemplate::New(Inventory);  
  
    Handle<ObjectTemplate> instance =  
        inventoryTpl->InstanceTemplate();  
  
    instance->Set(String::New("items"), Integer::New(257));  
  
    target->Set(String::NewSymbol("Inventory"),  
        inventoryTpl->GetFunction());  
}
```

```
Handle<Value> Inventory(const Arguments &args) {  
    return args.This();  
}
```

}

Methods

```
Inventory.prototype.addStock = function(newStock) {  
    this.items += newStock;  
}
```

```
Inventory.prototype.ship = function(orders) {  
    if (this.items < orders)  
        throw Exception("Not enough items");  
  
    this.items -= orders  
}
```

Methods

Registering prototype methods

```
// operating on inventoryTpl->PrototypeTemplate()  
NODE_SET_PROTOTYPE_METHOD(inventoryTpl,  
                           "addStock",  
                           AddStock);  
NODE_SET_PROTOTYPE_METHOD(inventoryTpl,  
                           "ship",  
                           Ship);  
  
target->Set(String::NewSymbol("Inventory"),  
            inventoryTpl->GetFunction());
```

Methods

Accessing object properties

```
Handle<Value> AddStock(const Arguments &args) {  
    HandleScope scope;  
  
    Handle<Object> This = args.This();  
  
    int items = This->Get(String::New("items"))->Uint32Value();  
  
    items += args[0]->Uint32Value();  
  
    This->Set(String::New("items"), Integer::New(items));  
  
    return Undefined();  
}
```

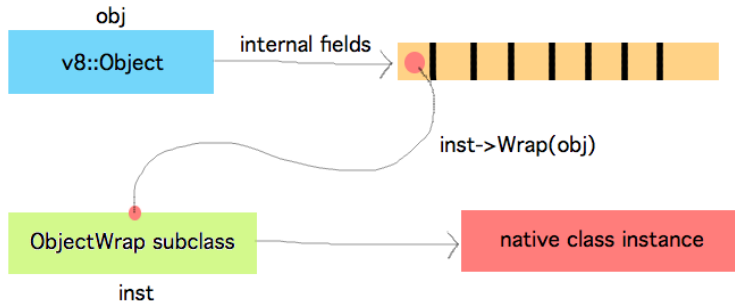
Methods

Throwing an exception

```
Handle<Value> Ship(const Arguments &args) {  
    HandleScope scope;  
  
    Handle<Object> This = args.This();  
    int items = This->Get(String::New("items"))->Uint32Value();  
  
    int orders = args[0]->Uint32Value();  
  
    if (items < orders)  
        return ThrowException(String::New("Not enough items"));  
  
    This->Set(String::New("items"), Integer::New(items - orders));  
  
    return Undefined();  
}
```

ObjectWrap

- Associate native C++ objects with JS objects
- Node specific class which manages garbage collection
- Stored internally in fields



ObjectWrap

```
// native C++ class
namespace Library {
class Inventory {
    Inventory();
    void addStock(int);
    int ship(int);
    int getItems();

    int items; // private
};
}
```


ObjectWrap

Setting internal field count

```
Handle<ObjectTemplate> instance =  
    inventoryTpl->InstanceTemplate();  
  
instance->SetInternalFieldCount(1);
```

ObjectWrap

Wrapping

```
namespace binding {  
class Inventory : public ObjectWrap {  
public:  
    static Handle<Value> New(const Arguments &args) {  
        Inventory *wrapper = new Inventory();  
        wrapper->Wrap(args.Holder());  
        return args.Holder();  
    }  
}
```

TODO: put diagram

ObjectWrap

Unwrapping

```
static Handle<Value> Ship(const Arguments &args) {  
    // extract  
    Inventory *wrapper = Unwrap<Inventory>(args.Holder());  
  
    int orders = args[0]->Uint32Value();  
    int result = wrapper->inv->ship(orders);  
  
    if (result == -1)  
        return ThrowException(String::New("Not enough items"));  
  
    return Undefined();  
}
```

Going Async

- The easiest way is to use *uv_queue_work()*
- **Every async call requires a set of 3 functions**
 1. Set up and invoke *uv_queue_work()*
 2. Do blocking task (run in separate thread)
 3. Clean up (run in main thread)
- **Use a 'baton' to pass around data**
 - *uv_request_t* is used by *libuv*
 - But it's *data* field is important to store the baton itself
- Slightly cumbersome :(

Going Async

```
var inv = new (require('./build/Release/async')).Inventory()  
  
inv.reshelve(function() {  
    console.log("Reshelving done");  
})  
console.log("After reshelve in source");  
for (var i = 1; i < 5; i++)  
    setTimeout(function() {  
        console.log("Tick");  
    }, i*1000);
```

Going Async

The native blocking code (method of class *Library::Inventory*)

```
void reshelve() {  
    sleep(5);  
}
```

Going Async

JS callback

```
static Handle<Value> Reshelve(const Arguments &args) {  
    Inventory *wrapper = Unwrap<Inventory>(args.Holder());  
  
    Handle<Function> cb = Handle<Function>::Cast(args[0]);  
  
    ReshelveBaton *baton = new ReshelveBaton();  
    baton->request.data = baton;  
    baton->callback = Persistent<Function>::New(cb);  
    baton->wrapper = wrapper;  
  
    uv_queue_work(Loop(), &baton->request,  
                  ReshelveAsync, ReshelveAsyncAfter);  
  
    return Undefined();  
}
```

Going Async

Thread pool function

```
static void ReshelveAsync(uv_work_t *req) {  
    // This runs in a separate thread  
    // NO V8 interaction should be done  
    ReshelveBaton *baton =  
        static_cast<ReshelveBaton*>(req->data);  
    // if you want to modify baton values  
    // do synchronous work here  
    baton->wrapper->inv->reshelve();  
}
```


Going Async

Clean up

```
static void ReshelveAsyncAfter(uv_work_t *req) {  
    ReshelveBaton *baton =  
        static_cast<ReshelveBaton*>(req->data);  
  
    Handle<Value> argv[] = { Null() }; // no error  
    baton->callback->Call(Context::GetCurrent()->Global(),  
                          1, argv);  
  
    baton->callback.Dispose();  
    delete baton;  
}
```

Going Async

Output

```
After reshelve in source  
Tick  
Tick  
Tick  
Tick  
Reshelfing done
```

Linking your library

Linking external libs in Waf:

```
def configure(conf):  
    # ...  
    # uses pkg-config  
    conf.check_cfg(package='<pkg-config name>', args='--cflags --libs',  
        uselib_store='ALIAS')  
  
def build(bld):  
    # ...  
    obj.uselib = 'ALIAS'
```

Holder vs This

- *args.This()* is always the this object passed in to the function
- *args.Holder()* runs up the prototype chain to the 'right' object
- Signatures decide the 'right' object, automatically handled by *NODE_PROTOTYPE_SET_METHOD*
- **Always** use *Holder()* to be on the safe side

Things I haven't covered

- **Accessors**

- Per property accessors
- Indexed accessors (*object[5]*)
- Named property accessors (*object.property*)
- Function Signatures and HasInstance for type safety
- Emitting events using new JS only EventEmitter
- Details of libuv
- Using V8 on its own

You might want to look at

- <https://github.com/weaver/uuidjs>
- <https://github.com/nikhilm/node-taglib>
- <https://github.com/pietern/hiredis-node>

End notes

Contact:

- @nikhilcutshort
- nsm.nikhil@gmail.com

Cover image by Munjal Savla (*by-nc-sa*)

Extra material below

Calling JS functions

```
var calljs = require("../build/Release/calljs")

var f = function() {
  console.log("This", this);
  console.log(arguments);
}

calljs.apply(f, { cool: "dude" }, "one", 2, 3.14);
```

Calling JS functions

```
Handle<Value> Apply(const Arguments &args) {  
    HandleScope scope;  
  
    Handle<Function> func = Handle<Function>::Cast(args[0]);  
    Handle<Object> receiver = args[1]->ToObject();  
  
    Handle<Value> *argv = new Handle<Value>[args.Length() - 2];  
    for (int i = 2; i < args.Length(); i++)  
        argv[i-2] = args[i];  
  
    func->Call(receiver, args.Length()-2, argv);  
  
    delete argv;  
    return Undefined();  
}
```

Strings to-and-fro

v8::String -> C string

```
Handle<Value> Print(const Arguments &args) {
    HandleScope scope;

    for (int i = 0; i < args.Length(); i++) {
        if (!args[i]->IsString())
            continue;

        // also String::AsciiValue
        String::Utf8Value val(args[i]);
        printf("%s ", *val); // <<<<<
    }
    return Undefined();
}
```

Strings to-and-fro

C string -> v8::String

```
Handle<Value> Read(const Arguments &args) {  
    HandleScope scope;  
  
    char str[1024];  
    fgets(str, 1023, stdin);  
  
    Local<String> v8String = String::New(str, strlen(str));  
    return scope.Close(v8String);  
}
```