

Olivier Lalonde's blog

How to write your own native Node.js extension

3

Tweet

Like

May 12, 2011

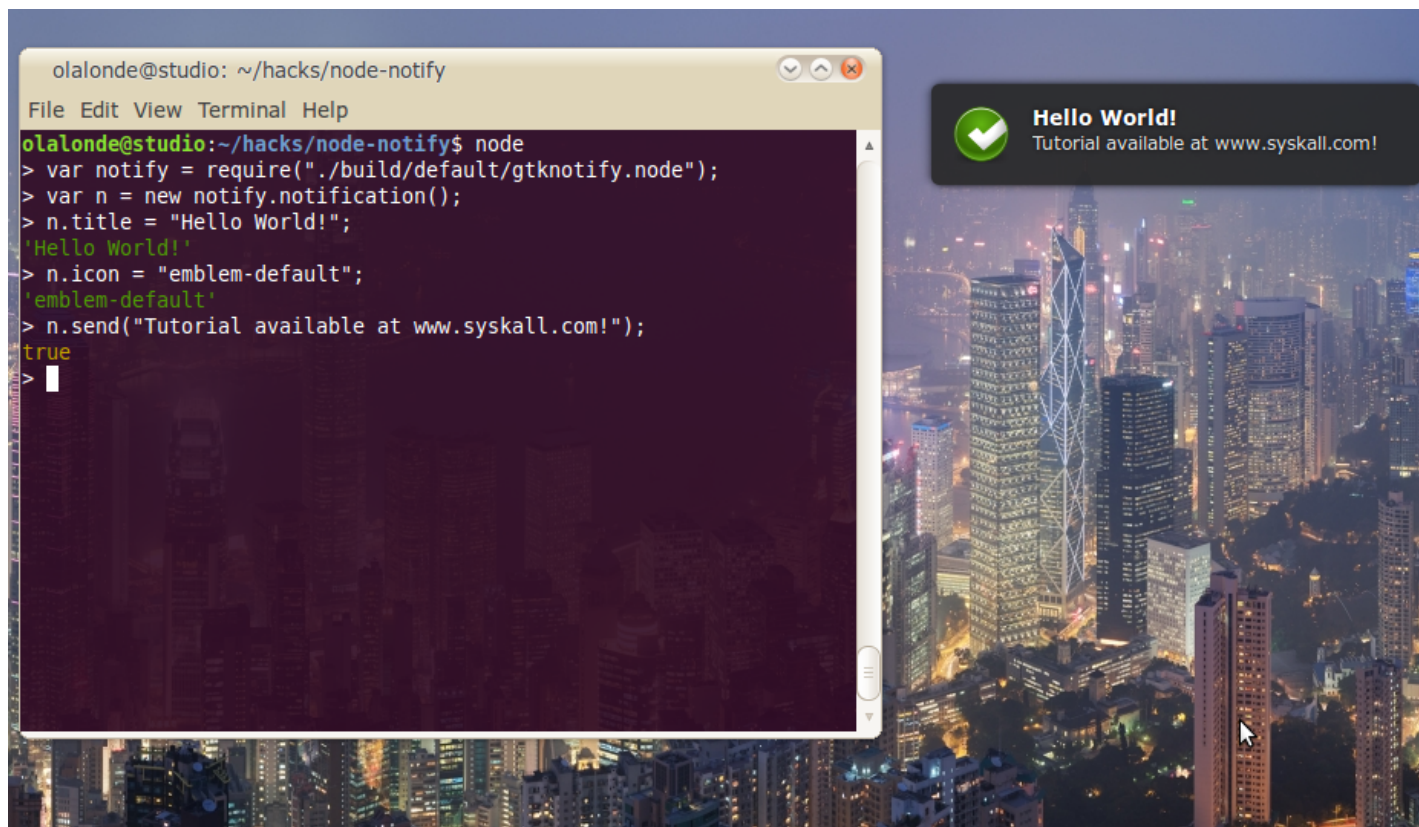
UPDATE: There is now a Node.js addon for loading and calling dynamic libraries using pure JavaScript: [node-ffi](#). Also, `node-waf` is no longer being used to compile Node.js extensions.

TRANSLATIONS: This post was translated to Chinese: <http://www.oschina.net/translate/how-to-write-your-own-native-nodejs-extension>

##Introduction##

This is a follow up to [How to roll out your own Javascript API with V8](#). You should still be able to follow if you haven't read it.

We will now port the [code we have written for V8](#) to Node.js and package it for npm.



The full source code of this tutorial is available [from github](#):

```
git clone git://github.com/olalonde/jsnotify.git
```

You can also install it through `npm`:

```
npm install notify
```

The code was tested on Ubuntu 10.10 64-bit and Node.js v0.5.0-pre.

##Getting started##

First let's create a node-notify folder and with the following directory structure.

```
.
|-- build/                # This is where our extension is built.
|-- demo/
|   |-- demo.js           # This is a demo Node.js script to test our extension.
|-- src/
|   |-- node_gtknotify.cpp # This is the where we do the mapping from C++ to JavaS
|-- wscript               # This is our build configuration used by node-waf
```

This fine looking tree was generated with the `tree` utility.

Now let's create our test script `demo.js` and decide upfront what our extension's API should look like:

```
// This loads our extension on the notify variable.
// It will only load a constructor function, notify.notification().
var notify = require("../build/default/gtknotify.node"); // path to our extension

var notification = new notify.notification();
notification.title = "Notification title";
notification.icon = "emblem-default"; // see /usr/share/icons/gnome/16x16
notification.send("Notification message");
```

##Writing our Node.js extension##

###The Init method###

In order to create a Node.js extension, we need to write a C++ class that extends `node::ObjectWrap`. `ObjectWrap` implements some utility methods that lets us easily interface with Javascript.

Let's write the skeleton for our class:

```
#include <v8.h> // v8 is the Javascript engine used by Node
#include <node.h>
// We will need the following libraries for our GTK+ notification
#include <string>
#include <gtkmm.h>
#include <libnotifymm.h>

using namespace v8;

class Gtknotify : node::ObjectWrap {
private:
public:
    Gtknotify() {}
    ~Gtknotify() {}
    static void Init(Handle<Object> target) {
        // This is what Node will call when we load the extension through require(), so
    }
};

/*
 * WARNING: Boilerplate code ahead.
 *
 * See https://www.cloudkick.com/blog/2010/aug/23/writing-nodejs-native-extensions/
 *
 * Thats it for actual interfacing with v8, finally we need to let Node.js know how
 * Because a Node.js extension can be loaded at runtime from a shared object, we need
 * so we do the following:
 */

v8::Persistent<FunctionTemplate> Gtknotify::persistent_function_template;
extern "C" { // Cause of name mangling in C++, we use extern C here
    static void init(Handle<Object> target) {
        Gtknotify::Init(target);
    }
    // @see http://github.com/ry/node/blob/v0.2.0/src/node.h#L101
    NODE_MODULE(gtknotify, init);
}
```

Now, we'll have to write the following code in our `Init()` method:

1. Declare our constructor function and bind it to our target variable.

`var n = require("notification");` will bind `notification()` to `n`: `n.notification()`.

```
// Wrap our C++ New() method so that it's accessible from Javascript
// This will be called by the new operator in Javascript, for example: new notificat
v8::Local<FunctionTemplate> local_function_template = v8::FunctionTemplate::New(New)

// Make it persistent and assign it to persistent_function_template which is a static
Gtknotify::persistent_function_template = v8::Persistent<FunctionTemplate>::New(local_function_template)

// Each JavaScript object keeps a reference to the C++ object for which it is a wrapper
Gtknotify::persistent_function_template->InstanceTemplate()->SetInternalFieldCount(1)
// Set a "class" name for objects created with our constructor
Gtknotify::persistent_function_template->SetClassName(v8::String::NewSymbol("Notification"));

// Set the "notification" property of our target variable and assign it to our constructor
target->Set(String::NewSymbol("notification"), Gtknotify::persistent_function_template);
```

1. Declare our attributes: `n.title` and `n.icon`.

```
// Set property accessors
// SetAccessor arguments: Javascript property name, C++ method that will act as the getter
Gtknotify::persistent_function_template->InstanceTemplate()->SetAccessor(String::NewSymbol("title"), Title);
Gtknotify::persistent_function_template->InstanceTemplate()->SetAccessor(String::NewSymbol("icon"), Icon);
// For instance, n.title = "foo" will now call SetTitle("foo"), n.title will now call GetTitle()
```

1. Declare our prototype method: `n.send()`

```
// This is a Node macro to help bind C++ methods to Javascript methods (see https://nodejs.org/docs/latest/api/native_addons.html)
// Arguments: our constructor function, Javascript method name, C++ method name
NODE_SET_PROTOTYPE_METHOD(Gtknotify::persistent_function_template, "send", Send);
```

Our `Init()` method should now look like this:

```
// Our constructor
static v8::Persistent<FunctionTemplate> persistent_function_template;
```

```

static void Init(Handle<Object> target) {
    v8::HandleScope scope; // used by v8 for garbage collection

    // Our constructor
    v8::Local<FunctionTemplate> local_function_template = v8::FunctionTemplate::New(New(
    Gtknotify::persistent_function_template = v8::Persistent<FunctionTemplate>::New(local_function_template-
    Gtknotify::persistent_function_template->InstanceTemplate()->SetInternalFieldCount(1);
    Gtknotify::persistent_function_template->SetClassName(v8::String::NewSymbol("Notification"));

    // Our getters and setters
    Gtknotify::persistent_function_template->InstanceTemplate()->SetAccessor(String::NewSymbol("title"),
    Gtknotify::persistent_function_template->InstanceTemplate()->SetAccessor(String::NewSymbol("icon"),

    // Our methods
    NODE_SET_PROTOTYPE_METHOD(Gtknotify::persistent_function_template, "send", Send);

    // Binding our constructor function to the target variable
    target->Set(String::NewSymbol("notification"), Gtknotify::persistent_function_template);
}

```

All that is left to do is to write the C++ methods that we used in our Init method: `New`, `GetTitle`, `SetTitle`, `GetIcon`, `SetIcon`, `Send`

###Our constructor method: New()###

The New() method creates an instance of our class (a Gtknotify object), sets some default values to our properties and returns a Javascript handle to this object. This is the expected behavior when calling a constructor function with the new operator in Javascript.

```

std::string title;
std::string icon;

// new notification()
static Handle<Value> New(const Arguments& args) {
    HandleScope scope;
    Gtknotify* gtknotify_instance = new Gtknotify();
    // Set some default values
    gtknotify_instance->title = "Node.js";
    gtknotify_instance->icon = "terminal";

    // Wrap our C++ object as a Javascript object
    gtknotify_instance->Wrap(args.This());
}

```

```
    return args.This();
}
```

###Our getters and setters: GetTitle(), SetTitle(), GetIcon(), SetIcon()###

The following is pretty much boilerplate code. It boils down to back and forth conversion between C++ values to Javascript (V8) values.

```
// this.title
static v8::Handle<Value> GetTitle(v8::Local<v8::String> property, const v8::AccessorInfo& info) {
    // Extract the C++ request object from the JavaScript wrapper.
    Gtknotify* gtknotify_instance = node::ObjectWrap::Unwrap<Gtknotify>(info.Holder())
    return v8::String::New(gtknotify_instance->title.c_str());
}

// this.title=
static void SetTitle(Local<String> property, Local<Value> value, const AccessorInfo& info) {
    Gtknotify* gtknotify_instance = node::ObjectWrap::Unwrap<Gtknotify>(info.Holder())
    v8::String::Utf8Value v8str(value);
    gtknotify_instance->title = *v8str;
}

// this.icon
static v8::Handle<Value> GetIcon(v8::Local<v8::String> property, const v8::AccessorInfo& info) {
    // Extract the C++ request object from the JavaScript wrapper.
    Gtknotify* gtknotify_instance = node::ObjectWrap::Unwrap<Gtknotify>(info.Holder())
    return v8::String::New(gtknotify_instance->icon.c_str());
}

// this.icon=
static void SetIcon(Local<String> property, Local<Value> value, const AccessorInfo& info) {
    Gtknotify* gtknotify_instance = node::ObjectWrap::Unwrap<Gtknotify>(info.Holder())
    v8::String::Utf8Value v8str(value);
    gtknotify_instance->icon = *v8str;
}
```

###Our prototype method: Send()###

First we have to extract the C++ object `this` references. We then build our notification using the object's properties (title, icon) and finally display it.

```
// this.send()
static v8::Handle<Value> Send(const Arguments& args) {
    v8::HandleScope scope;
    // Extract C++ object reference from "this"
```

```

Gtknotify* gtknotify_instance = node::ObjectWrap::Unwrap<Gtknotify>(args.This());

// Convert first argument to V8 String
v8::String::Utf8Value v8str(args[0]);

// For more info on the Notify Library: http://library.gnome.org/devel/libnotify/0
Notify::init("Basic");
// Arguments: title, content, icon
Notify::Notification n(gtknotify_instance->title.c_str(), *v8str, gtknotify_instance);
// Display the notification
n.show();
// Return value
return v8::Boolean::New(true);
}

```

##Compiling our extension##

`node-waf` is the build tool used to compile Node extensions which is basically a wrapper for [waf](#). The build process can be configured with a file called `wscript` in our top directory:

```

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

def configure(conf):
    conf.check_tool("compiler_cxx")
    conf.check_tool("node_addon")
    # This will tell the compiler to link our extension with the gtkmm and libnotify
    conf.check_cfg(package='gtkmm-2.4', args='--cflags --libs', uselib_store='LIBGTKMM')
    conf.check_cfg(package='libnotify-1.0', args='--cflags --libs', uselib_store='LIBNOTIFY')

def build(bld):
    obj = bld.new_task_gen("cxx", "shlib", "node_addon")
    obj.cxxflags = ["-g", "-D_FILE_OFFSET_BITS=64", "-D_LARGEFILE_SOURCE", "-Wall"]
    # This is the name of our extension.
    obj.target = "gtknotify"
    obj.source = "src/node_gtknotify.cpp"
    obj.uselib = ['LIBGTKMM', 'LIBNOTIFY']

```

We're now ready to build! In the top directory, run the following command:

```
node-waf configure && node-waf build
```


If everything goes right, we should now have our compiled extension in

```
./build/default/gtknotify.node
```

. Let's try it!

```
$ node
> var notif = require('./build/default/gtknotify.node');
> n = new notif.notification();
{ icon: 'terminal', title: 'Node.js' }
> n.send("Hello World!");
true
```

The previous code should display a notification in the top right corner of your screen!

##Packaging for npm##

That's pretty cool, but how about sharing your hard work with the Node community? That's primarily what the Node Package Manager is used for: making it easy to import extensions/modules and distribute them.

Packaging an extension for npm is very straightforward. All you have to do is create a `package.json` file in your top directory which contains some info about your extension:

```
{
  // Name of your extension (do not include node or js in the name, this is implicit)
  // This is the name that will be used to import the extension through require().

  "name" : "notify",

  // Version should be http://semver.org/ compliant

  "version" : "v0.1.0"

  // These scripts will be run when calling npm install and npm uninstall.

  , "scripts" : {
    "preinstall" : "node-waf configure && node-waf build"
    , "preuninstall" : "rm -rf build/*"
  }

  // This is the relative path to our built extension.

  , "main" : "build/default/gtknotify.node"

  // The following fields are optional:
```



```
, "description" : "Description of the extension...."  
, "homepage" : "https://github.com/olalonde/node-notify"  
, "author" : {  
  "name" : "Olivier Lalonde"  
  , "email" : "olalonde@gmail.com"  
  , "url" : "http://www.syskall.com/"  
}  
, "repository" : {  
  "type" : "git"  
  , "url" : "https://github.com/olalonde/node-notify.git"  
}  
}
```

For more details on the package.json format, documentation is available through `npm help json`.
Note that most fields are optional.

You can now install your new npm package by running `npm install` in your top directory. If everything goes right, you should be able to load your extension with a simple

`var notify = require('your-package-name');`. Another useful command is `npm link` which creates a symlink to your development directory so that any change to your code is reflected instantly - no need to install/uninstall perpetually.

Assuming you wrote a cool extension, you might want to publish it online in the central npm repository. In order to do that, you first need to create an account:

```
$ npm adduser
```

Next, go back to the root of your package code and run:

```
$ npm publish
```

That's it, your package is now available for anyone to install through the `npm install your-package-name` command.

##Conclusion##

Writing a native Node extension can be cumbersome and verbose at times but it is well worth the hard earned bragging rights!

Thanks for reading. Let me know in the comments if you run into any problem, I'll be glad to help.

If you liked this, maybe you'd also like what I [tweet on Twitter!](#) Might even want to [hire me](#)?

##References##

- [How to roll out your own Javascript API with V8](#)
- [Writing Node.js Native Extensions](#)
- [V8 JavaScript Engine Embedder's Guide](#)
- [Introduction to npm](#)
- [Node.js](#)

Found a typo? Want to improve this post? [Edit on GitHub](#).

Want to get notified about more articles like this? [Follow @o_lalonde on Twitter](#).

Comments

0 Comments **syskall****1** Login ▾ **Recommend** 1  **Tweet**  **Share****Sort by Best** ▾

Start the discussion...

LOG IN WITH


OR SIGN UP WITH DISQUS 

Be the first to comment.

ALSO ON SYSKALL

Boot2docker: Using nfs instead of vboxsf to mount /Users


4 comments • 4 years ago

 **l0sdep** — Dane Summers nfs-client is inside the Virtualbox VM, not on the host (OS X). That part of the script is run inside the VM.**How to roll out your own Javascript API with V8**


2 comments • 4 years ago

 **Diego Rosa dos Santos** — great, thanks by tuto!**Crazy And Not So Crazy Startup Ideas (2015 edition)**

7 comments • 4 years ago

 **jim** — great article. more please!**Pagination with Handlebars - Olivier Lalonde's blog**

4 comments • 6 years ago

 **Wai Hong Fong** — it's actually useful to have the following to enable 'first' and 'last' which are useful pagination options too.case 'first': if ...**Olivier Lalonde's blog**Olivier Lalonde's blog
olalonde@gmail.com [olalonde](#)
 [o_lalonde](#)

I'm a friendly hacker and startup guy living in Hong Kong and Shenzhen. Current interests include Node.js, Go, Bitcoin, distributed systems.