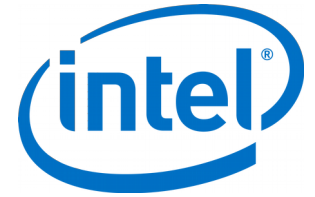


Writing Node.js Bindings



General Principles

Gabriel Schulhof
Intel Finland Oy

Assumption

- Make bindings simple – one-to-one
 - Avoid abstractions in C++ – leave those for Javascript
 - Benefit: Re-use C API documentation 😊
- C-specific artifacts need not be exposed to Javascript
 - Function pointer context
 - Variable length array via pointer-to-data plus length parameter

```
void some_api(  
    void *context,  
    void (*callback)(  
        void *context,  
        int someUsefulData));
```

```
void some_api(  
    SomeStructure *list, or  
    size_t list_length);
```

```
struct Something {  
    SomeStructure *list;  
    size_t list_length;  
};
```

General principles

- Trust data from the native side
- Validate data from the Javascript side
- Avoid argument juggling
 - Require a certain number of arguments
 - Require each argument to have a certain type

General principles – Stack

- Make the native call stack mimic Javascript
 - The binding throws a Javascript exception if parameter array length or parameter type validation fails and returns immediately, without setting the return value
 - Helpers called by the binding throw a Javascript exception and return `false`, indicating that the binding is to return immediately because an exception has occurred

General principles - Stack

```
bool some_helper_helper(...) {
    ...
    if (something_broken) {
        Nan::ThrowError("Something is broken!");
        return false;
    }
    ...
    return true;
}
bool some_binding_helper(...) {
    ...
    if (!some_helper_helper(...)) {
        return false;
    }
    ...
    if (something_wrong) {
        Nan::ThrowError("Something went wrong!");
        return false;
    }
    ...
    return true;
}
NAN_METHOD(bind_some_function) {
    /* Validate parameter list length and parameter types */
    ...
    if (!some_binding_helper(...)) {
        return;
    }
    ...
    info.GetReturnValue().Set(Nan::New(returnValue));
}
```

General principles - Data

- Write a pair of helpers for converting data structures to/from Javascript. For example, for this structure:

```
struct coordinates {  
    double lat;  
    double lon;  
    double alt;  
};
```

General principles - Data

- C converter
 - Returns `true` on success
 - Throws Javascript exception and returns `false` on validation error
 - Accepts a pointer to a structure to fill out (`destination`) and a Javascript object containing the Javascript equivalent of the structure (`jsSource`)
 - Creates a local variable of the structure type (`local`)
 - Retrieves each structure member from `jsSource`, validates it, and assigns it to `local`
 - If all members of `local` are successfully assigned,
`*destination = local;`
`return true;`

General principles - Data

```
bool c_coordinates(  
    struct coordinates *destination,  
    Local<Object> jsSource) {  
    struct coordinates local;  
  
    Local<Value> jsLat = Nan::Get(jsSource,  
        Nan::New("lat").ToLocalChecked()  
        .ToLocalChecked());  
    if (!jsLat->IsNumber()) {  
        Nan::ThrowError("lat is not a number");  
        return false;  
    }  
    local.lat = Nan::To<double>(jsLat).FromJust();  
  
    // Same for "lon" and for "alt"  
  
    *destination = local;  
    return true;  
}
```

Nan::Maybe types provide additional failure interception

General principles - Data

- Observation: Because they follow the signature recommended in previous slides, functions of the form `c_structurname()` as illustrated in the previous slide can be composed if one structure contains one or more other structures:

```
Local<Value> jsMoreDetails = Nan::Get(jsSource,  
    Nan::New("more_details").ToLocalChecked())  
    .ToLocalChecked();  
if (!jsMoreDetails->IsObject()) {  
    Nan::ThrowError(  
        "more_details must be an object!");  
    return false;  
}  
if (!c_more_details(&(local.more_details),  
    Nan::To<Object>(jsMoreDetails).ToLocalChecked())) {  
    return false;  
}
```

General principles - Data

- Javascript converter
 - Returns a Javascript object
 - Accepts a pointer to a C structure and a Javascript object, which it returns. If absent, it creates a Javascript object (done using C++ default parameter value)

General principles - Data

```
Local<Object> js_coordinates(  
    const struct coordinates *source,  
    Local<Object> jsDestination = Nan::New<Object>()) {  
    Nan::Set(jsDestination,  
        Nan::New("lat").ToLocalChecked(), Nan::New(source->lat));  
    Nan::Set(jsDestination,  
        Nan::New("lon").ToLocalChecked(), Nan::New(source->lon));  
    Nan::Set(jsDestination,  
        Nan::New("alt").ToLocalChecked(), Nan::New(source->alt));  
  
    return jsDestination;  
}
```

- Observation: Javascript converters can also be composed:

```
Nan::Set(jsDestination,  
    Nan::New("more_details").ToLocalChecked(),  
    js_more_details(source->more_details));
```

Automating

- Parsing the header files with the usual suspects
 - cat, grep, awk
 - Project style conventions can come in handy
- What can be extracted?
 - Precompiler constants
 - Enums
- The result:

```
myAddon.SOME_CONSTANT;  
myAddon.SomeEnum.SOME_ENUM_GREEN;  
myAddon.SomeEnum.SOME_ENUM_BLUE;
```

The Simplest Binding

- Function takes primitive data types and returns a primitive data type

```
double area_of_oval(double xRadius,  
                    double yRadius);
```

```
NAN_METHOD(bind_area_of_oval) {  
    VALIDATE_ARGUMENT_COUNT(info, 2);  
    VALIDATE_ARGUMENT_TYPE(info, 0, IsNumber);  
    VALIDATE_ARGUMENT_TYPE(info, 1, IsNumber);  
    Info.GetReturnValue().Set(Nan::New(  
        area_of_oval(  
            Nan::To<double>(info[0]).FromJust(),  
            Nan::To<double>(info[1]).FromJust())));  
}
```

Data

- Any parameter or return value that does not fit in a primitive data type → pointers
 - Structures
 - Traversable
 - The pointer itself is not important
 - Who owns the pointer? → sometimes ambiguous
 - Handles
 - “magic”, in that the pointer's value is important
 - The data to which the pointer refers is often opaque
 - The pointer is given by the native side, and is requested back on subsequent calls

Data-Structures

```
struct coordinates {  
    double latitude;  
    double longitude;  
    double altitude;  
};  
/* Both APIs return true on success */  
bool find_restaurant_by_name(const char *name, struct coordinates *location);  
bool restaurant_at_coordinates(const struct coordinates *location);  
  
NAN_METHOD(bind_find_restaurant_by_name) {  
    VALIDATE_ARGUMENT_COUNT(info, 2);  
    VALIDATE_ARGUMENT_TYPE(info, 0, IsString);  
    VALIDATE_ARGUMENT_TYPE(info, 1, IsObject);  
    struct coordinates result;  
    bool returnValue = find_restaurant_by_name(  
        (const char *)*(String::Utf8Value(info[0])), &result);  
    if (returnValue) {  
        js_coordinates(&result,  
            Nan::To<Object>(info[1]).ToLocalChecked());  
    }  
    info.GetReturnValue().Set(Nan::New(returnValue));  
}
```

Data-Structures

```
NAN_METHOD(bind_restaurant_at_coordinates) {  
    VALIDATE_ARGUMENT_COUNT(info, 1);  
    VALIDATE_ARGUMENT_TYPE(info, 0, IsObject);  
  
    struct coordinates;  
    if (!c_coordinates(&coordinates,  
        Nan::To<Object>(info[0]).ToLocalChecked())) {  
        return;  
    }  
  
    info.GetReturnValue().Set(  
        Nan::New(  
            restaurant_at_coordinates(&coordinates)));  
}
```


Data-Handles

- “magic” value – structure behind it is often opaque
 - Two APIs – one to get a new handle, one to give it back:
 - `open()/close()`
 - `setTimeout()/clearTimeout()`
 - Overwrite the return value and you have a leak:
 - `fd = open(...); fd = -1;`
 - `to = setTimeout(...); to = null;`
 - `interval = setInterval(...); interval = null;`
 - Once you release the handle, if you attempt to use it or release it again,
 - nothing happens
 - an error is returned
 - segfault

Data-Handles

- How to bind? It's a pointer, so ... array of bytes? Number?
 - Bad, because we've chosen not to trust data from Javascript:
`handle = getHandle(); handle[1] = 5;`
- If only we could assign a property to an object that is not visible from Javascript – and I don't mean `defineProperty()` with `enumerable: false` because that can still reveal pointer values
- V8 and NAN to the rescue with `SetInternalFieldCount()` - Store a pointer inside a Javascript object such that it is only accessible from C++
 - Javascript objects are copied by reference, so we get the semantics of the `setTimeout()` return value

Data-Handles

- Let's create a class template for passing handles into Javascript. We assign a `void *` to a Javascript object while giving the object a nice-looking class name to boot. What do we need of the C++ class?

- Instantiation

```
Local<Object> MyJSHandle::New(void *handle);
```

- Give me back the native handle

```
void *MyJSHandle::Resolve(Local<Object> jsHandle);
```

- Gut the Javascript object of its native handle (invalidate)

- We need that for the case where the native handle is gone but the Javascript object is still around:

```
close(fd); /* accidentally */ close(fd);
```

So we need

```
Nan::SetInternalFieldPointer(jsHandle, 0, 0);
```

after which a second call to an API using the handle will either do nothing or throw an exception – we can tell the handle is invalid → segfault averted

```

template <class T> class JSHandle {
    static Nan::Persistent<v8::FunctionTemplate> &theTemplate() {
        static Nan::Persistent<v8::FunctionTemplate> returnValue;
        if (returnValue.IsEmpty()) {
            v8::Local<v8::FunctionTemplate> theTemplate =
                Nan::New<v8::FunctionTemplate>();
            theTemplate
                ->SetClassName(Nan::New(T::jsClassName()).ToLocalChecked());
            theTemplate->InstanceTemplate()->SetInternalFieldCount(1);
            Nan::Set(Nan::GetFunction(theTemplate).ToLocalChecked(),
                Nan::New("displayName").ToLocalChecked(),
                Nan::New(T::jsClassName()).ToLocalChecked());
            returnValue.Reset(theTemplate);
        }
        return returnValue;
    }
public:
    static v8::Local<v8::Object> New(void *data) {
        v8::Local<v8::Object> returnValue =
            Nan::GetFunction(Nan::New(theTemplate())).ToLocalChecked()
                ->NewInstance();
        Nan::SetInternalFieldPointer(returnValue, 0, data);
        return returnValue;
    }
    static void *Resolve(v8::Local<v8::Object> jsObject) {
        void *returnValue = 0;
        if (Nan::New(theTemplate())->HasInstance(jsObject)) {
            returnValue = Nan::GetInternalFieldPointer(jsObject, 0);
        }
        if (!returnValue) {
            Nan::ThrowTypeError((std::string("Object is not of type " +
                T::jsClassName()).c_str()));
        }
        return returnValue;
    }
};

```

Data-Handles

- Usage:

```
class JSRemoteResourceHandle : public JSHandle<JSRemoteResourceHandle> {  
public:  
    static const char *jsClassName() {return "RemoteResourceHandle";}  
};
```

- And in a binding:

```
NAN_METHOD(new_handle) {  
    ...  
    struct remote_resource_handle *remote =  
        get_remote_resource(...);  
    info.GetReturnValue().Set(  
        JSRemoteResourceHandle::New(remote));  
}  
NAN_METHOD(use_handle) {  
    Local<Object> jsHandle = Nan::To<Object>(info[0]).ToLocalChecked();  
    struct remote_resource_handle *remote =  
        JSRemoteResourceHandle::Resolve(jsHandle);  
    if (!remote) {  
        return;  
    }  
    do_something_with_remote_resource(remote, ...);  
}
```

Data-Handles

- Important consideration: Avoid creating two different Javascript handles for the same native handle. This can happen in conjunction with callbacks, where the C callback is given a copy of the handle as a convenience:

```
void remote_resource_set_int(
    remote_resource *remote,
    int newValue,
    void (*operation_complete)(
        void *context,
        remote_resource *remote,
        int result),
    void *context);
```

Avoid creating a new Javascript object inside
`operation_complete()` via
`JSRemoteResource::New(remote)`

Data-Handles

```
struct CallbackData {
    Nan::Callback *callback;
    Nan::Persistent<Object> *jsRemote;
};

void set_int_operation_complete(
    void *context, remote_resource *remote,
    int result) {
    Nan::HandleScope scope;
    struct CallbackData *data =
        (struct CallbackData *)context;
    Local<Value> arguments[2] = {

        // JSRemoteHandle::New(remote)
        Nan::New<Object>(*(data->jsRemote)),
        Nan::New(result)
    };
    data->callback->Call(2, arguments);
    ...
}
```

Callbacks

- In C, functions cannot be created/destroyed at runtime, so almost every API that takes a callback has a `void *context` to distinguish different callbacks
 - Javascript is the king of context so we do not need to expose this but instead use the context to store at least the Javascript function that we're supposed to call – because Javascript functions **can** be created/destroyed at runtime

Callbacks

- Two kinds of callbacks
 - Implicitly removed
 - `read()` ;
 - `setTimeout()` ;
 - Explicitly removed
 - `setInterval()` ;
 - `accept()` ;
 - These always come in a pair, forming a bracket
 - one API to attach the callback, returning a handle
 - one API to detach the callback using the handle to identify which previously attached callback to detach

Callbacks

- Explicit and implicit removal are not mutually exclusive – for example, the callback can return a certain value to indicate that it is to be removed implicitly
 - Here, we want to avoid a kind of double-free – where the Javascript callback releases the handle via a call to the explicit API for releasing, and then returns a value indicating that we are to release the handle. We need to check that the handle is not already released.

Callbacks-Implicit

```
struct CallbackData {  
    Nan::Callback *callback;  
    Nan::Persistent<Object> *jsRemote;  
};  
void set_int_operation_complete(  
    void *context, remote_resource *remote,  
    int result) {  
    Nan::HandleScope scope;  
    struct CallbackData *data =  
        (struct CallbackData *)context;  
  
    /* Call JS callback as shown previously */  
  
    delete data->callback;  
    data->jsRemote->Reset();  
    delete data->jsRemote;  
    delete data;  
}
```

Callbacks - Explicit

- Structure like `CallbackData` stores everything needed to clean up, and is itself stored inside a JS handle. So, for an API like this

```
FileSystemWatch watch_file_name(  
    const char *file_name,  
    void (*file_changed)(void *context, const char *file_name, const int fileEvent),  
    void *context);  
  
bool unwatch_file_name(FileSystemWatch watch);
```

We create the following somewhat self-referential structure

```
struct CallbackData {  
    // This is a JSFileSystemWatch handle containing a pointer to  
    // this structure  
    Nan::Persistent<Object> *jsHandle;  
    Nan::Callback *jsCallback;  
    FileSystemWatch watch;  
};
```


Callbacks - Explicit

```
NAN_METHOD(bind_watch_file_name) {  
    ...  
    CallbackData *data = new CallbackData;  
    data->watch = watch_file_name(  
        (const char *)*String::Utf8Value(info[0]),  
        watch_file_name_event,  
        data);  
    data->jsCallback =  
        new Nan::Callback(Local<Function>::Cast(info[1]));  
    Local<Object> jsHandle =  
        JSFileSystemWatch::New(data);  
    data->jsHandle =  
        new Nan::Persistent<Object>(jsHandle);  
    info.GetReturnValue().Set(jsHandle);  
}
```

Callbacks - Explicit

```
NAN_METHOD(bind_unwatch_file_name) {  
    CallbackData *data = (CallbackData *)  
        JSFileSystemWatch::Resolve(Nan::To<Object>(info[0]).ToLocalChecked());  
    if (!data) {  
        return;  
    }  
    bool returnValue = unwatch_file_name(data->watch);  
    if (returnValue) {  
        delete data->jsCallback;  
        data->jsHandle->Reset();  
        delete data->jsHandle;  
        delete data;  
    }  
    info.GetReturnValue().Set(Nan::New(returnValue));  
}
```

Semantics are API-dependent



Callbacks - Explicit

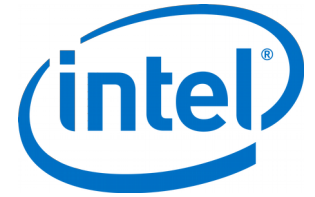
```
void watch_file_name_event(  
    void *context,  
    const char *file_name,  
    const int fileEvent) {  
    Nan::HandleScope scope;  
    CallbackData *data = (CallbackData *)context;  
    Local<Object> jsHandle =  
        Nan::New<Object>(*(data->jsHandle));  
    Local<Value> arguments[2] = {  
        Nan::New(file_name).ToLocalChecked(),  
        Nan::New(fileEvent)  
    };  
    data->jsCallback->Call(2, arguments);  
}
```

Callbacks - Hybrid

- Can be removed explicitly as well as implicitly
 - What if the Javascript callback removes the handle explicitly and then returns a value indicating that the handle is to be removed implicitly? → We ~~can~~ must use `::Resolve()`

Callbacks - Hybrid

```
bool watch_file_name_event(
    void *context,
    const char *file_name,
    const int fileEvent) {
    Nan::HandleScope scope;
    CallbackData *data = (CallbackData *)context;
    Local<Object> jsHandle = Nan::New<Object>(*(data->jsHandle));
    Local<Value> arguments[2] = {
        Nan::New(file_name).ToLocalChecked(),
        Nan::New(fileEvent)
    };
    Local<Value> jsReturnValue = data->jsCallback->Call(2, arguments);
    // validate returnValue and throw an exception if it's not a boolean
    bool returnValue =
        Nan::To<bool>(jsReturnValue).FromJust();
    if (!returnValue) {
        data = JSFileSystemWatch::Resolve(jsHandle);
        // If data is not null, free it just like in bind_unwatch_file_name()
    }
    return returnValue;
}
```



Thanks!

- Thanks @nagineni for helping me hammer out some of these issues
- Full text is at <https://github.com/gabrielschulhof/bindings-principles>

Please

- file issues
 - file PRs :)
 - Syntax highlighting using pluma
- Gabriel Schulhof <gabriel.schulhof@intel.com>