Extending node.js using C++

Kenneth Geisshirt
kg@realm.io

Realm Inc.
@realm
http://github.com/Realm/
http://realm.io/

Today's goal

- Learn
  - the basics of V8 internals and API
  - how to wrap C++ classes
- Go home and write an extension

Agenda

1. Why write extensions in C++
2. My demo C++ classes
3. Building extensions
4. Wrapping classes
    - Setting up class
    - Instantiating objects
    - Setters, getters, etc.
    - Methods
    - Callbacks/anonymous functions
    - Exceptions

Why Extensions in C++?

- You get access to system resources
  - I/O, peripheral devices, GPUs, etc,
- To get the performance of C++
- Cross-Language
  - Common core features in C++, many language bindings
- Legacy code
  - Tons of old and useful C/C++/Fortran code

Demo C++ Classes

- Person
  - `firstname()`
  - `lastname()`
  - `birthday()`
  - `to_str()`

Getters and setters

- Book
  - `add(Person* p)`
  - `Person *lookup(string name)`
  - operator `[size_t index]`
  - `remove(size_t index)`
  - `size_t size()`

https://github.com/kneth/DemoNodeExtension

V8 concepts

- `Isolate` is an isolated instance of V8
- *Handles* are references to JavaScript objects, and V8's garbage collector reclaims them
    - `Local` handles are allocated on the stack; life-time is scope based
    - Persistent handles are allocated on the heap; life-time can span multiple function calls
- You don't return an object - you set the return value using `GetReturnValue().Set()`
    - and you cannot return a `Local` object (I'll return to it later on)
- V8 has classes to represent JavaScript types (`String`, `Number`, `Array`, `Object`, ...)

Breaking changes 0.10 → 0.12

- V8 API changes in node.js 0.12 (February 2015)
- How to return values from C++ to JavaScript
- Type name for methods' arguments
- Creating objects require an `Isolate`
- String encoding is strict (UTF-8 is common)
- Extensions cannot support both 0.10 and 0.12+
- https://strongloop.com/strongblog/node-js-v0-12-c-apis-breaking/

Building extensions

- Wrapping classes (`person.cpp` → `person_wrap.cpp`, ...)
- Build process described in `bindings.gyp`
- `node-gyp configure build`

```
#include <node.h>
#include "person_wrap.hpp"
#include "book_wrap.hpp"

using namespace v8;

void InitAll (Handle<Object> exports) {
    PersonWrap::Init(exports);
    BookWrap::Init(exports);
}

NODE_MODULE(funstuff, InitAll)
```

Name of extension

OS X specific options

```
{
  'targets': [{
    'target_name': 'funstuff',
    'sources': [ 'person.cpp', 'person_wrap.cpp', 'book.cpp',
                 'book_wrap.cpp', 'funstuff.cpp' ],
    'xcode_settings': {
      'OTHER_CFLAGS': [ '-mmacosx-version-min=10.8', '-std=c++11',
                        '-stdlib=libc++', '-fexceptions', '-frtti' ]
    }
  }]
}
```

Wrapping a class

- Wrapper classes inherent from `node::ObjectWrap`
- All methods are static

Add class to V8

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

#include "book.hpp"
#include "person.hpp"

class BookWrap : public node::ObjectWrap {
public:
    static void Init(v8::Handle<v8::Object> exports);
    static void New(const v8::FunctionCallbackInfo<v8::Value>& args);

    BookWrap();

private:
    ~BookWrap();

    Book* m_book;
    static v8::Persistent<v8::Function> Constructor;
};
```

The class to wrap

Helper to create new objects

Adding a class to V8

```
var funstuff = require('./build/Release/funstuff');
```

- Calling `BookWrap::Init()` to register/add the class
  - Sets up constructor, methods, and basic infrastructure

```
void BookWrap::Init(Handle<Object> exports) {
  Isolate* isolate = exports->GetIsolate();

  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, BookWrap::New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "Book"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  NODE_SET_PROTOTYPE_METHOD(tpl, “add", Add);

  tpl->InstanceTemplate()->SetIndexedPropertyHandler(Getter, Setter,
                                                     0, Deleter, Enumerator);

  Constructor.Reset(isolate, tpl->GetFunction());
  exports->Set(String::NewFromUtf8(isolate, "Book"), tpl->GetFunction());
}
```

Setting the class name

Adding a method

Getter, setter, etc.

Preparing constructor

Instantiate an object

```
var book = new funstuff.Book();
```

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

  if (args.IsConstructCall()) {
    if (args.Length() == 0) {

      BookWrap* bw = new BookWrap();
      Book *b      = new Book();
      bw->m_book   = b;

      bw->Wrap(args.This());

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

Create wrapper and object

Add wrapper to V8 runtime

Return the object

Methods

```
var book = new funstuff.Book();
console.log('Book: ' + book.length());
```

- Methods are implemented in C++
  - Input validation is important (`IsString`, `IsNumber`, …)

Arguments: `args.Length()`, `args[0]`

Get `this` as wrapper object

Get current scope

Create an object to return

Return the new object

```
void BookWrap::Length(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  HandleScope scope(isolate);

  BookWrap* bw = ObjectWrap::Unwrap<BookWrap>(args.This());
  const int count = bw->m_book->size();
  Local<Integer> result = Integer::New(isolate, count);
  args.GetReturnValue().Set(result);
}
```

Instantiate objects

- Instantiating wrapper object in C++
  - Method of one class returns object of other class
  - For example: `var person = book[4];`

```
Handle<Object> PersonWrap::New(Isolate* isolate, Book* b, uint32_t index) {
  EscapableHandleScope scope(isolate);

  Handle<Value> argv[] = { Boolean::New(isolate, true) };
  Local<Function> cons = Local<Function>::New(isolate, Constructor);
  Handle<Object> obj = cons->NewInstance(1, argv);

  PersonWrap* pw = PersonWrap::Unwrap<PersonWrap>(obj);
  pw->m_person = (*b)[size_t(index)];

  return scope.Escape(obj);
}
```

Dummy argument

Call constructor:
`PersonWrap::New(const FunctionCallbackInfo<Value>& args)`

Add object to current scope

Indexed getters and setters

```
book[6] = new Person();
var person = book[4];
```

Unwrap `this` and get C++ object

```
void BookWrap::Getter(uint32_t index, const PropertyCallbackInfo<Value>& info) {
    Isolate* isolate = info.GetIsolate();
    HandleScope scope(isolate);

    BookWrap* bw = ObjectWrap::Unwrap<BookWrap>(info.This());
    Book*     b  = bw->m_book;

    if (index >= b->size()) {
        isolate->ThrowException(Exception::RangeError(String::NewFromUtf8(isolate,
                                                      "invalid row index")));
        info.GetReturnValue().SetUndefined();
    }
    else {
        Handle<Object> result = PersonWrap::New(isolate, b, index);
        info.GetReturnValue().Set(result);
    }
}
```

Instantiate a wrapper object and return it

Validate input (`index` is in range)

Value to set; remember to validate!

```
void BookWrap::Setter(uint32_t index, Local<Value> value,
                      const PropertyCallbackInfo<Value>& info)
```

Accessors

```
var person = new funstuff.Person();
person.firstname = “Arthur”;
```

- Accessors are useful for known properties
  - C++ isn’t dynamic as JavaScript
- Added to V8 during initialisation (PersonWrap::Init())

```
tpl->InstanceTemplate()->SetAccessor(String::NewFromUtf8(isolate, "firstname"),
PersonWrap::FirstnameGetter, PersonWrap::FirstnameSetter);
```

```
void PersonWrap::FirstnameGetter(Local<String> property,
                                 const PropertyCallbackInfo<Value>& info) {
  Isolate* isolate = Isolate::GetCurrent();
  HandleScope scope(isolate);

  PersonWrap *pw = ObjectWrap::Unwrap<PersonWrap>(info.This());
  Person *p = pw->m_person;

  info.GetReturnValue().Set(String::NewFromUtf8(isolate, p->firstname().c_str()));
}
```

Callbacks

```
book.each(function (p) {
    console.log("Firstname: " + p.firstname);
});
```

- Callbacks and anonymous functions are JavaScript in a nutshell

- Functions are objects: `Function` is used in V8

```
void BookWrap::Each(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  HandleScope scope(isolate);

  Book* book = ObjectWrap::Unwrap<BookWrap>(args.This())->m_book;

  if (args.Length() == 1) {
    if (args[0]->IsFunction()) {
      Local<Function> fun = Local<Function>::Cast(args[0]);
      for(uint32_t i = 0; i < book->size(); ++i) {
          Local<Object> pw = PersonWrap::New(isolate, book, i);
          Local<Value> argv[1] = { pw };
          fun->Call(Null(isolate), 1, argv);
      }
      args.GetReturnValue().SetUndefined();
      return;
    }
  }
}
```

The anonymous function

Set up arguments

Call the function

Throwing Exceptions

- Throwing a C++ exception is a no-go
  - set node.js' state to "exception"
  - when returning to JavaScript an exception is thrown
    - V8 has a limited number of exceptions: `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `Error`

```
Isolate* isolate = Isolate::GetCurrent();
HandleScope scope(isolate);

// …

isolate->ThrowException(Exception::SyntaxError(String::NewFromUtf8(isolate,
                                                "No arguments expected")));
args.GetReturnValue().SetUndefined();
return;
```

Catching Exceptions

```
try {
  var s = book.apply(function (b) {
    throw { msg: "Error" };
  });
  console.log("  Length: " + s);
}
catch (e) {
  console.log("  Exception caught: " + e.msg);
}
```

- Callbacks might throw an exception
- V8 has a `TryCatch` class to check for it

If `HasCaught`() is true, an JS exception was thrown

Set the node.s' "exception" state

```
void BookWrap::Apply(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  HandleScope scope(isolate);

  Local<Function> fun = Local<Function>::Cast(args[0]);
  Handle<Value> argv[] = { args.This() };
  TryCatch trycatch;
  Handle<Value> v = fun->Call(Null(isolate), 1, argv);
  if (trycatch.HasCaught()) {
    trycatch.ReThrow();
  }
  args.GetReturnValue().Set(v);
}
```

NAN

- Native Abstraction for Node (NAN) makes it easier to write extensions

- Hides breaking changes in the V8 API
    - Your extension will support many versions!

- Functions and macros for common tasks

- https://github.com/nodejs/nan

Observations

- Extensions do not have to be a one-to-one mapping
- A lot of code to do input validation
  - JavaScript isn't a strongly typed language
    - Unit testing is very important
- C++ has classes - JavaScript doesn't
  - Awkward for JavaScript programmers
- Crossing language barrier during call is hard for debuggers

Learn more

- Check out my demo: https://github.com/kneth/DemoNodeExtension

- Google's documentation: https://developers.google.com/v8/embed?hl=en

- JavaScript - The Good Parts. D. Crockford. O'Reilly Media, 2008.

- Any modern C++ text book 😱