7. C++ addons

Tuesday, May 28, 2024 10:44 AM

V8:
C++ library Node.js uses to provide JS implementation.
Providing mechanisms for creating objects,
calling functions etc. V8's API is
documented mostly in the v8.h header file
(deps/v8/include/v8.h in the Node.js source tree).

libuv:
The C library that implements the Node.js event loop,
its worker threads & all of the
Asynchronous behaviour of the platform.
Serving as a cross-platform abstraction library,
giving easy, POSIX-like access across all major OS &
many common system tasks
e.g. File System
Web Sockets
Timers
System Events

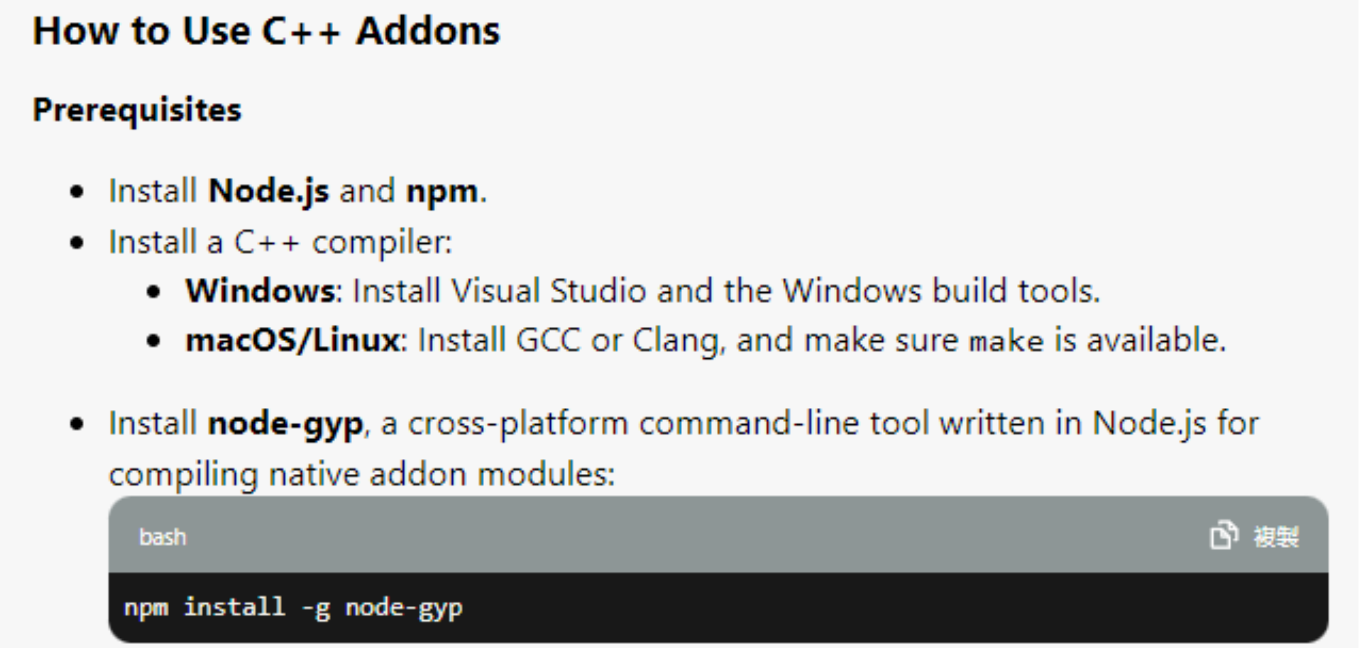
Internal Node.js libraries:
Node.js itself exports C++ APIs that addons can use,
the most important of which is the **node::ObjectWrap** class

Example
Hello World C++ addon

hello.cc

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

```
namespace demo {
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
        Isolate* isolate = args.GetIsolate();
        args.GetReturnValue().Set(String::NewFromUtf8(
                isolate, "world").ToLocalChecked());
}

void Initialize(Local<Object> exports) {
        NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)

} // namespace demo
```

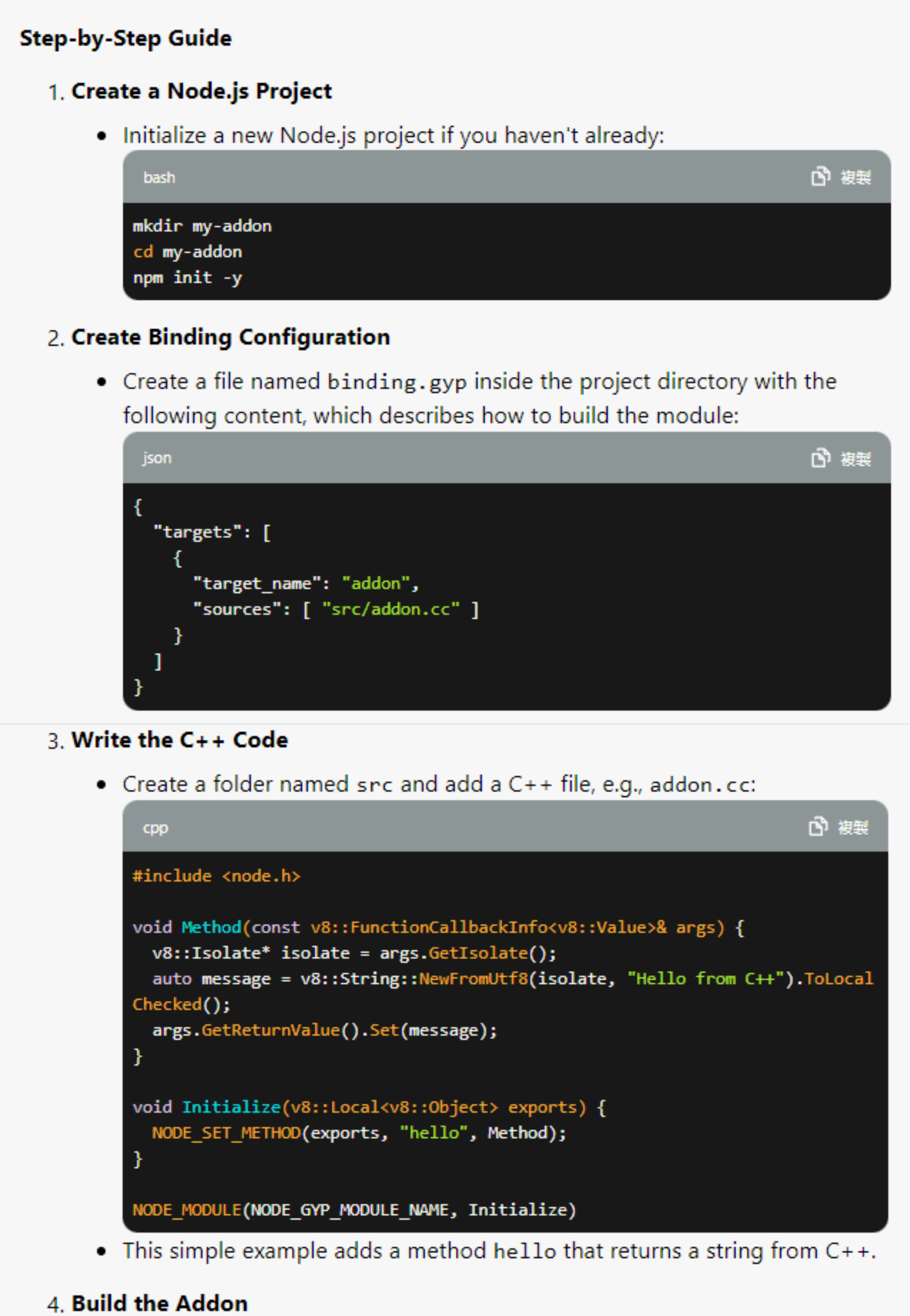


How to Use C++ Addons

Prerequisites

- Install **Node.js** and **npm**.
- Install a C++ compiler:
  - **Windows**: Install Visual Studio and the Windows build tools.
  - **macOS/Linux**: Install GCC or Clang, and make sure `make` is available.
- Install **node-gyp**, a cross-platform command-line tool written in Node.js for compiling native addon modules:

```bash
npm install -g node-gyp
```



Step-by-Step Guide

1. **Create a Node.js Project**
   - Initialize a new Node.js project if you haven't already:

```bash
mkdir my-addon
cd my-addon
npm init -y
```

2. **Create Binding Configuration**
   - Create a file named `binding.gyp` inside the project directory with the following content, which describes how to build the module:

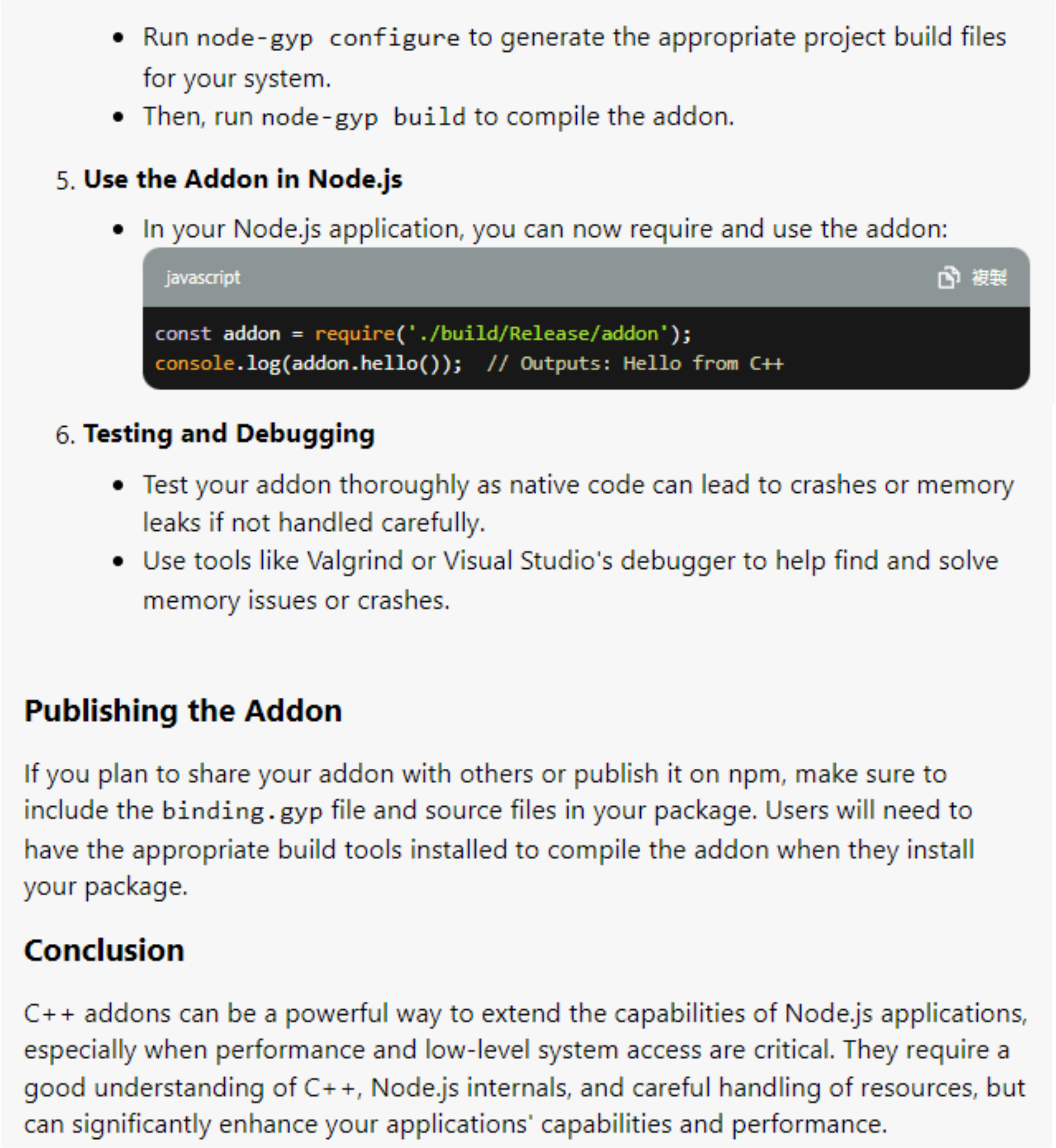
```json
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "src/addon.cc" ]
    }
  ]
}
```

3. **Write the C++ Code**
   - Create a folder named `src` and add a C++ file, e.g., `addon.cc`:

```cpp
#include <node.h>

void Method(const v8::FunctionCallbackInfo<v8::Value>& args) {
  v8::Isolate* isolate = args.GetIsolate();
  auto message = v8::String::NewFromUtf8(isolate, "Hello from C++").ToLocalChecked();
  args.GetReturnValue().Set(message);
}

void Initialize(v8::Local<v8::Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

   - This simple example adds a method `hello` that returns a string from C++.

4. **Build the Addon**



- Run node-gyp configure to generate the appropriate project build files for your system.
- Then, run node-gyp build to compile the addon.

5. **Use the Addon in Node.js**
   - In your Node.js application, you can now require and use the addon:

```javascript
const addon = require('./build/Release/addon');
console.log(addon.hello());  // Outputs: Hello from C++
```

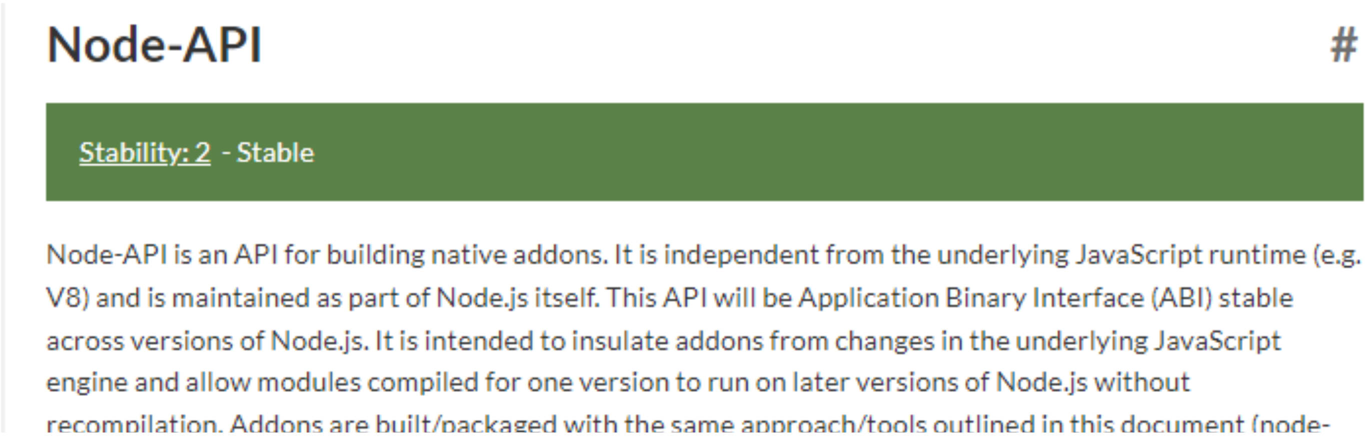
6. **Testing and Debugging**
   - Test your addon thoroughly as native code can lead to crashes or memory leaks if not handled carefully.
   - Use tools like Valgrind or Visual Studio's debugger to help find and solve memory issues or crashes.

Publishing the Addon

If you plan to share your addon with others or publish it on npm, make sure to include the binding.gyp file and source files in your package. Users will need to have the appropriate build tools installed to compile the addon when they install your package.

Conclusion

C++ addons can be a powerful way to extend the capabilities of Node.js applications, especially when performance and low-level system access are critical. They require a good understanding of C++, Node.js internals, and careful handling of resources, but can significantly enhance your applications' capabilities and performance.



Node-API

Stability: 2 - Stable

Node-API is an API for building native addons. It is independent from the underlying JavaScript runtime (e.g. V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate addons from changes in the underlying JavaScript engine and allow modules compiled for one version to run on later versions of Node.js without recompilation. Addons are built/packaged with the same approach/tools outlined in this document (node-

> gyp, etc.). The only difference is the set of APIs that are used by the native code. Instead of using the V8 or Native Abstractions for Node.js APIs, the functions available in the Node-API are used.
>
> Creating and maintaining an addon that benefits from the ABI stability provided by Node-API carries with it certain implementation considerations.

To use Node-API in the above "Hello world" example, replace the content of `hello.cc` with the following. All other instructions remain the same.

```
// hello.cc using Node-API
#include <node_api.h>

namespace demo {

napi_value Method(napi_env env, napi_callback_info args) {
  napi_value greeting;
  napi_status status;

  status = napi_create_string_utf8(env, "world", NAPI_AUTO_LENGTH, &greeting);
  if (status != napi_ok) return nullptr;
  return greeting;
}

napi_value init(napi_env env, napi_value exports) {
  napi_status status;
  napi_value fn;

  status = napi_create_function(env, nullptr, 0, Method, nullptr, &fn);
  if (status != napi_ok) return nullptr;

  status = napi_set_named_property(env, exports, "hello", fn);
  if (status != napi_ok) return nullptr;
  return exports;
}

NAPI_MODULE(NODE_GYP_MODULE_NAME, init)

}  // namespace demo
```

COPY

The functions available and how to use them are documented in C/C++ addons with Node-API.