



## Rust Berlin Meetup

# Embedding Rust in NodeJS Applications

## Rust and Tell

*Kiffin Gish*  
*26 May, 2020*

# Who am I?

Please allow me to introduce myself:

- Born and raised in sunny California
- Survived 35+ years hacking thru life
- Started with Assembler, Unix and C
- Ended up a “full stack” developer
- Got the Rust itch and can't get enough



Long live Rust!

@kiffin

# https://github.com/kgish



**Kiffin Gish**  
kgish

Edit profile

Never too old to learn new stuff ...

👤 Gishtech

📍 Gouda, The Netherlands

✉ [kiffin.gish@planet.nl](mailto:kiffin.gish@planet.nl)

🌐 <http://gishtech.com>

★ PRO

Overview Repositories 87 Projects 0 Packages 0 Stars 41 Followers 21 Following 31

## Pinned

Customize your pins

📄 [assembla-to-jira-migration](#)

A comprehensive toolset for migrating data from Assembla to Jira.

● Ruby ★ 3 🔗 7

📄 [angular-challenge](#)

Challenge to build a web app using the Angular and NestJS frameworks

● TypeScript

📄 [angular-unit-testing-workshop](#)

Angular unit testing workshop

● TypeScript

📄 [rust-amethyst-pong](#)

Game of pong written in Rust and using the Amethyst game engine.

● Rust

📄 [rust-webgl-template](#)

Rust and WebGL sample template

● Rust

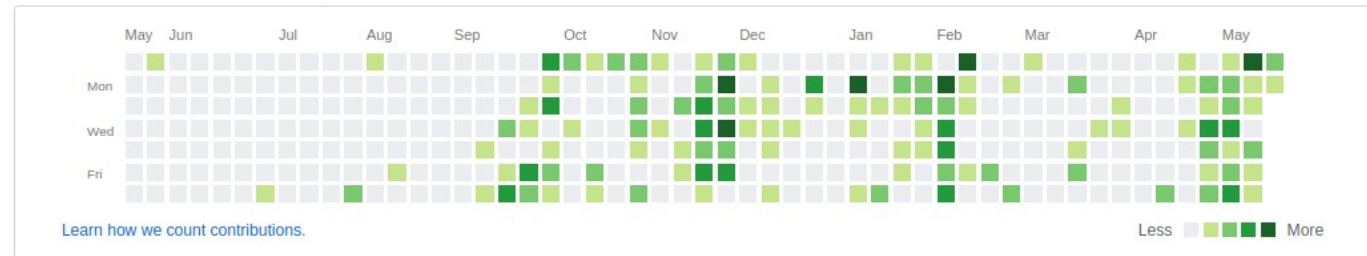
📄 [rust-node-addon-template](#)

Rust Node Addon example template using N-API

● Rust

521 contributions in the last year

Contribution settings ▾



Long live Rust!

@kiffin

# Why use Rust?

We already know why Rust is such a fantastic and fun programming language:

- Safe
- Fast
- Concurrent
- No garbage collection (GC)



Long live Rust!

@kiffin

# Why use Rust with Node.js?

More specifically, using Rust means:

- High computational speed
- Interoperability with external libraries (FFI)
- No runtime overhead
- Predictable performance
- Low-level access to hardware (device drivers)



Long live Rust!

@kiffin

# Why use Rust with Node.js?

More specifically, using Rust means:

- High computational speed
- Interoperability with external libraries (FFI)
- No runtime overhead
- Predictable performance
- Low-level access to hardware (device drivers)

Foreign  
Function  
Interface

“Start by rewriting specific performance critical modules in Rust ...”



Long live Rust!

@kiffin

# Foreign Function Interface (FFI)

```
extern "C" {  
    fn sqrt(x: f64) -> f64;  
}  
  
#[link(name = "m")]  
fn main() {  
  
    let x: f64 = 2.;  
    let result: f64 = unsafe { sqrt(x) };  
  
    println!("The square root of {} is {}", x, result);  
}
```



# Foreign Function Interface (FFI)

```
extern "C" {  
    fn sqrt(x: f64) -> f64;  
}  
  
#[link(name = "m")]  
fn main() {  
  
    let x: f64 = 2.;  
    let result: f64 = unsafe { sqrt(x) };  
  
    println!("The square root of {} is {}", x, result);  
}
```

All FFI functions are unsafe to call because the other language can do operations the the Rust compiler cannot check.





# Crate libm

```
use libm::sqrt;

fn main() {

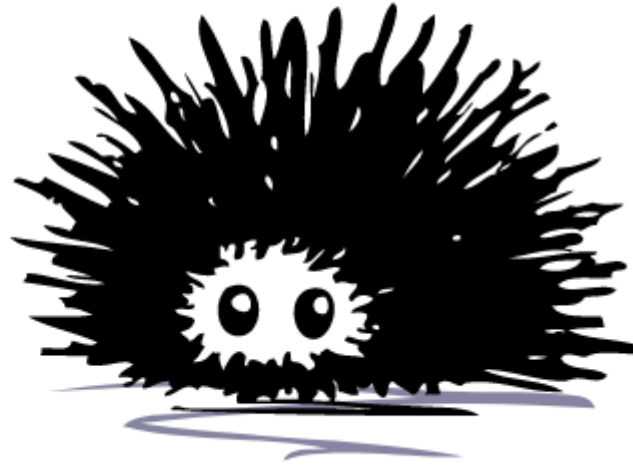
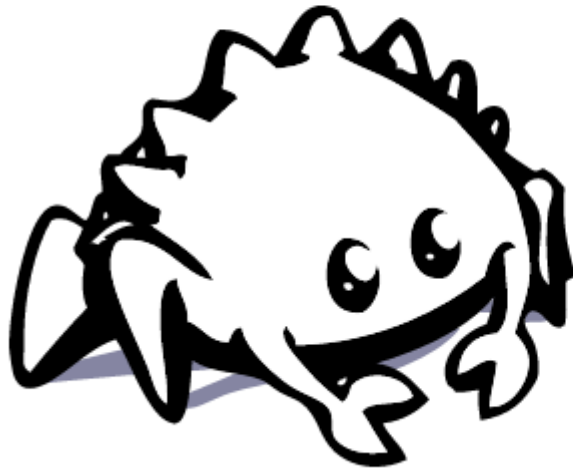
    let x: f64 = 2.;
    let result: f64 = sqrt(x);

    println!("The square root of {} is {}", x, result);
}
```



# Rustonomicon

## Meet Safe and Unsafe



<https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html>

<https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html>



Long live Rust!

@kiffin

# Node.js API (N-API)

*“... for building native Addons. It is independent from the underlying JavaScript runtime 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 major version to run on later major versions of Node.js without recompilation...”*

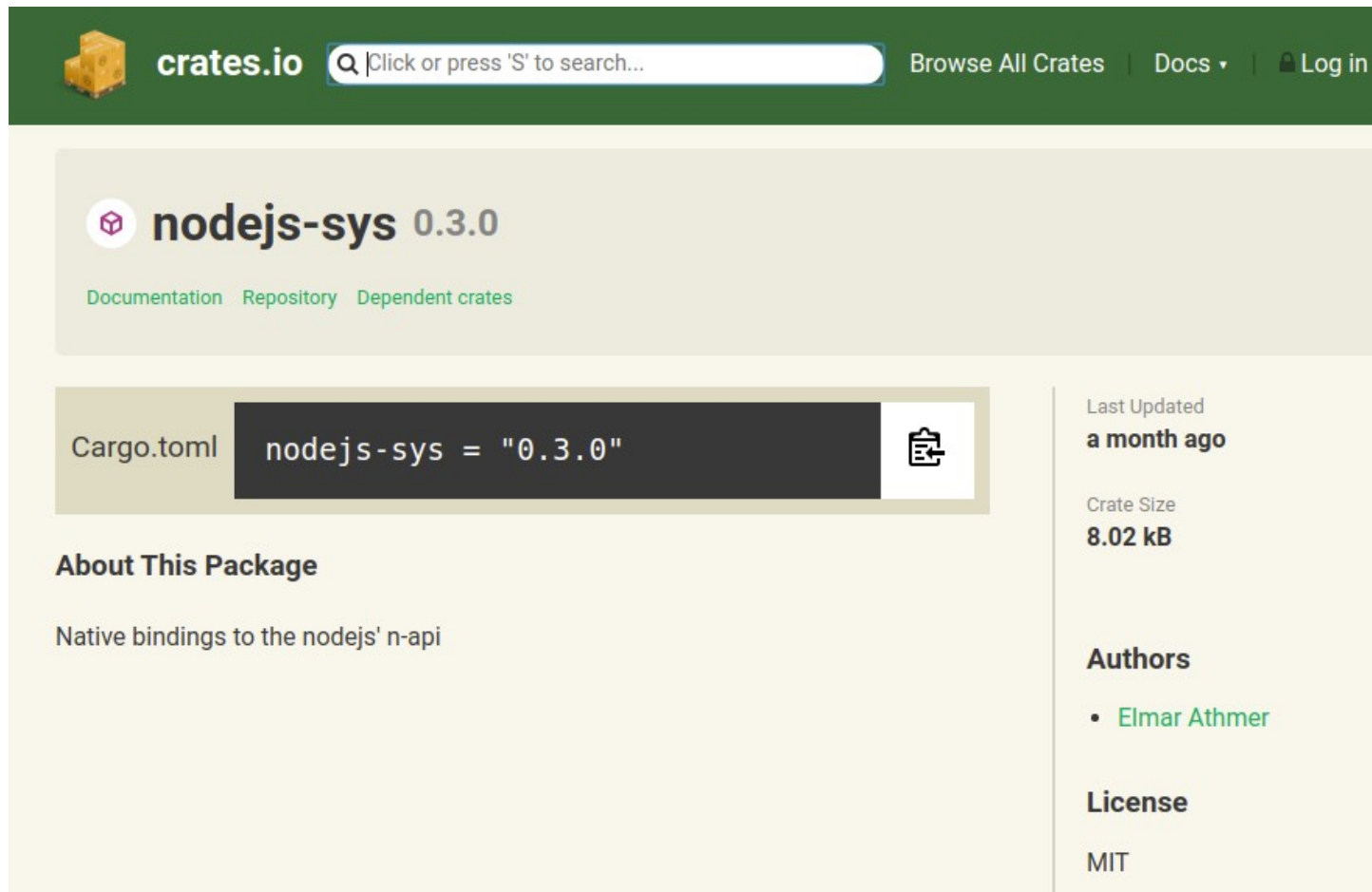
<https://nodejs.org/api/n-api.html>



Long live Rust!


@kiffin

# Nodejs-sys




The screenshot shows the crates.io interface for the `nodejs-sys` crate. At the top, there's a green header with the crates.io logo, a search bar, and links for "Browse All Crates", "Docs", and "Log in". Below the header, the crate name `nodejs-sys` and version `0.3.0` are displayed. Underneath, there are links for "Documentation", "Repository", and "Dependent crates". A code snippet is shown in a dark box: `Cargo.toml nodejs-sys = "0.3.0"` with a clipboard icon. To the right, a sidebar contains metadata: "Last Updated a month ago", "Crate Size 8.02 kB", "Authors" (listing Elmar Athmer), and "License MIT".

crates.io  Browse All Crates | Docs | Log in

 **nodejs-sys** 0.3.0

[Documentation](#) [Repository](#) [Dependent crates](#)

Cargo.toml `nodejs-sys = "0.3.0"` 

**About This Package**

Native bindings to the nodejs' n-api

Last Updated  
**a month ago**

Crate Size  
**8.02 kB**

**Authors**

- [Elmar Athmer](#)

**License**

MIT

<https://crates.io/crates/nodejs-sys>



Long live Rust!

@kiffin

# Cargo.toml

Create a Rust package with a library target `src/lib.rs` by executing.

```
$ cargo init --lib --crate-type=cdylib
```

The `--crate-type=cdylib` flag produces a dynamic system library which is used when compiling a dynamic library to be loaded from another language. This output type will create the relevant file type for other operating systems, the \*.so file for Linux.

The project directory should look like this:

```
|— Cargo.toml
|— package.json
└— src
    └— lib.rs
```



Long live Rust!

@kiffin

# Cargo.toml

```
[package]
name = "rust-node-addon-template"
version = "0.1.0"
authors = ["Kiffin Gish <kiffin.gish@planet.nl>"]
edition = "2018"

[lib]
crate-type=["cdylib"]

[dependencies]
nodejs-sys = "0.3.0"
```



# Module registration 1/2

## Module registration

#

N-API modules are registered in a manner similar to other modules except that instead of using the `NODE_MODULE` macro the following is used:

```
NAPI_MODULE(NODE_GYP_MODULE_NAME, Init)
```

The next difference is the signature for the `Init` method. For a N-API module it is as follows:

```
napi_value Init(napi_env env, napi_value exports);
```

The return value from `Init` is treated as the `exports` object for the module. The `Init` method is passed an empty object via the `exports` parameter as a convenience. If `Init` returns `NULL`, the parameter passed as `exports` is exported by the module. N-API modules cannot modify the `module` object but can specify anything as the `exports` property of the module.

[https://nodejs.org/api/n-api.html#n\\_api\\_module\\_registration](https://nodejs.org/api/n-api.html#n_api_module_registration)



Long live Rust!

@kiffin

# Module registration 2/2

```
use nodejs_sys::{napi_env, napi_value};

#[no_mangle]
pub unsafe extern "C" fn napi_register_module_v1(
    env: napi_env,
    exports: napi_value,
) -> nodejs_sys::napi_value {

    register_functions(env, exports);

    exports
}
```





# Create function

```
type CallbackFn = unsafe extern "C" fn(napi_env, napi_callback_info) -> napi_value;

unsafe fn create_function(env: napi_env, exports: napi_value, name: &str, func: CallbackFn) {

    let cname : CString = CString::new(t name).expect(msg: "CString::new failed");
    let mut result: napi_value = std::mem::zeroed();

    napi_create_function(
        env,
        cname.as_ptr(),
        cname.as_bytes().len(),
        Some(func),
        std::ptr::null_mut(),
        &mut result,
    );

    napi_set_named_property(env, exports, cname.as_ptr(), result);
}
```



# Create function

```
type CallbackFn = unsafe extern "C" fn(napi_env, napi_callback_info) -> napi_value;

unsafe fn create_function(env: napi_env, exports: napi_value, name: &str, func: CallbackFn) {

    let cname : CString = CString::new(t name).expect(msg: "CString::new failed");
    let mut result: napi_value = std::mem::zeroed();

    napi_create_function(
        env,
        cname.as_ptr(),
        cname.as_bytes().len(),
        Some(func),
        std::ptr::null_mut(),
        &mut result,
    );

    napi_set_named_property(env, exports, cname.as_ptr(), result);
}
```

```
create_function(env, exports, name: "sayHello", func: say_hello);
```



# say\_hello() → sayHello()

```
create_function(env, exports, name: "sayHello", func: say_hello);
```

```
use nodejs_sys::{napi_callback_info, napi_create_string_utf8, napi_env, napi_value};
use std::ffi::CString;

// --- say_hello() => string --- //
pub unsafe extern "C" fn say_hello(env: napi_env, _info: napi_callback_info) -> napi_value {

    let mut result: napi_value = std::mem::zeroed();
    let s: CString = CString::new(b"Hello from the mighty kingdom of Rust!").unwrap();

    napi_create_string_utf8(env, s.as_ptr(), s.as_bytes().len(), &mut result);

    result
}
```



# Build and run

```
$ cargo build --release  
$ cp ./target/release/librust_node_addon_template.so index.node
```

```
// index.js  
const addon = require('./index.node');  
  
console.log(`sayHello() => '${addon.sayHello()}'`);
```

```
$ node ./index.js
```



# Demo

```
// --- add_numbers(x,y) => number --- //
pub unsafe extern "C" fn add_numbers(env: napi_env, info: napi_callback_info) -> napi_value {
    let mut buffer: [napi_value; 2] = std::mem::MaybeUninit::zeroed().assume_init();
    let mut argc: usize = 2 as usize;
    let mut result: napi_value = std::mem::zeroed();

    napi_get_cb_info( env, info, &mut argc, buffer.as_mut_ptr(), std::ptr::null_mut(),
    | std::ptr::null_mut(),
    );

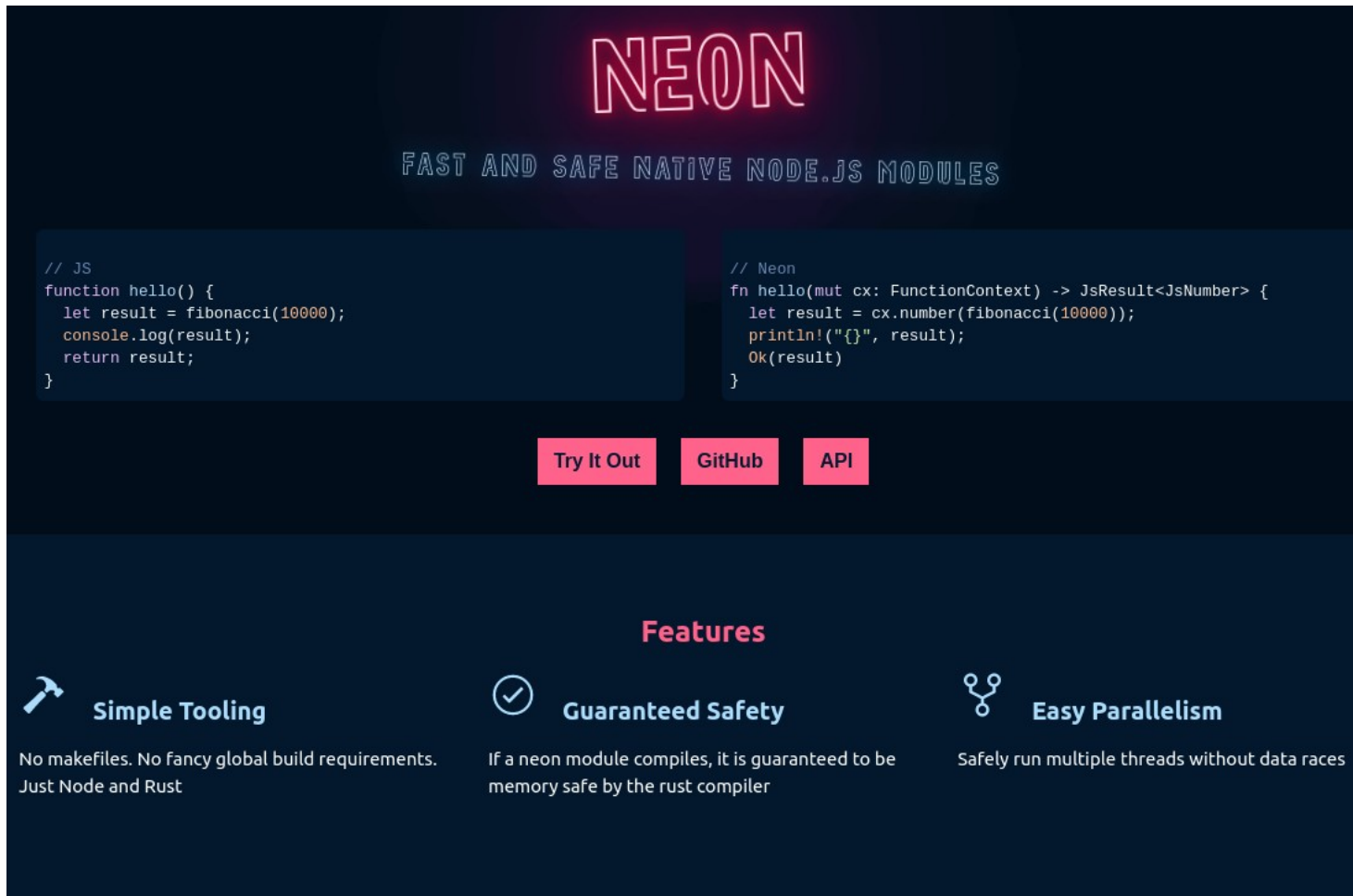
    let mut x: f64 = 0 as f64;
    let mut y: f64 = 0 as f64;

    napi_get_value_double(env, buffer[0], &mut x);
    napi_get_value_double(env, buffer[1], &mut y);
    let value: f64 = x + y;

    napi_create_double(env, value, &mut result);
    result
}
```



# Neon



**NEON**




FAST AND SAFE NATIVE NODE.JS MODULES

```
// JS
function hello() {
  let result = fibonacci(10000);
  console.log(result);
  return result;
}
```

```
// Neon
fn hello(mut cx: FunctionContext) -> JsResult<JsNumber> {
  let result = cx.number(fibonacci(10000));
  println!("{}", result);
  Ok(result)
}
```

[Try It Out](#) [GitHub](#) [API](#)

### Features

-  **Simple Tooling**  
No makefiles. No fancy global build requirements. Just Node and Rust
-  **Guaranteed Safety**  
If a neon module compiles, it is guaranteed to be memory safe by the rust compiler
-  **Easy Parallelism**  
Safely run multiple threads without data races

<https://neon-bindings.com>



Long live Rust!

@kiffin

# Installation

## Neon

Rust bindings for writing safe and fast native Node.js modules.

### Installation

```
$ npm install -g neon-cli  
$ neon new my-project  
$ cd my-project  
$ npm install
```

The directory tree should look something like this.

```
.  
├── lib  
│   └── index.js  
├── native  
│   ├── build.rs  
│   ├── Cargo.toml  
│   └── src  
│       └── lib.rs  
├── package.json  
└── README.md
```



Long live Rust!

@kiffin

# Cargo.toml

```
[package]
name = "my-project"
version = "0.1.0"
authors = ["Kiffin Gish <kiffin.gish@planet.nl>"]
license = "MIT"
build = "build.rs"
edition = "2018"
exclude = ["artifacts.json", "index.node"]

[lib]
name = "my_project"
crate-type = ["cdylib"]

[build-dependencies]
neon-build = "0.4.0"

[dependencies]
neon = "0.4.0"
```



Long live Rust!

@kiffin



# Module registration

```
// src/lib.rs
use neon::register_module;

// --- Register and export all functions --- //
register_module!(mut m, {

    m.export_function("sayHello", say_hello)?;

    Ok(())
});
```



# say\_hello()

```
// say_hello.rs
use neon::prelude::*;

// --- say_hello() => string --- //
pub fn say_hello(mut cx: FunctionContext) -> JsResult<JsString> {

    Ok(cx.string("Hello from the mighty kingdom of Rust!"))

}
```



# Demo

```
// add_numbers.rs
use neon::prelude::*;

// --- add_numbers(x,y) => number --- //
pub fn add_numbers(mut cx: FunctionContext) -> JsResult<JsNumber> {
    let x = cx.argument:::<JsNumber>(0)?.value();
    let y = cx.argument:::<JsNumber>(1)?.value();

    Ok(cx.number(x + y))
}
```



# Additional examples

## Run

```
$ ./run.sh [n]
```

where:

optional n = 1 - 5 in order to run only given example:

1. Function sayHello() => void
2. Function sendMessage(str) => void
3. Function addNumbers(x,y) => number
4. Function getUser() => user
5. Async function fibonacci(n) => number



# Async Fibonacci

```
// fibonacci_async.rs
use neon::prelude::*;

pub fn fibonacci_async(mut cx: FunctionContext) -> JsResult<JsUndefined> {
    let n : usize = cx.argument::(0)?.value() as usize;
    let cb = cx.argument::(1)?;

    let task = FibonacciTask { argument: n };
    task.schedule(cb);

    Ok(cx.undefined())
}
```



# Some conclusions 1/2

## Advantages of using Rust with Node.js:

- Computational demands
- Performance is predictable
- Low-level access to GPU and GPIO
- Cheaper hardware (IoT)



2 CPU cores, 1 GB RAM



1 CPU core, 512 MB RAM)



Long live Rust!

@kiffin

# Some conclusions 2/2

There are also some disadvantages:

- Support another programming language
- Different tool chain and deploy pipeline
- Steep learning curve



## Rust Node Addon Template

A simple Rust Node addon example template using N-API

### Introduction

This is part of the presentation that I gave at a recent [Berlin Rust Meetup](#).

### Requirements

For this project template, the following is required:

- [Rust](#)
- [Node.js](#)
- [npm](#)
- [nvm](#)

In order to build the `nodejs-sys` crate (below) you need `libclang` since bindings are being generated at build-time by `bindgen`. Therefore, you will require the `clang` and `llvm-dev` libraries:

```
$ sudo apt-get install llvm-dev clang
```

Ensure that you have the correct Node version installed:

```
$ nvm install 12.16.3  
$ nvm use 12.16.3
```

<https://github.com/kgish/rust-node-addon-template>





# End

Hope you liked it!

If you are looking for an eager and enthusiastic guy to help you out ... look me up!

[kiffin.gish@planet.nl](mailto:kiffin.gish@planet.nl)

<https://github.com/kgish>

<https://www.linkedin.com/in/kiffin>



Long live Rust!

@kiffin