

The word "HELIX" is centered on a solid blue rectangular background. The letters are white and have a thin, double-lined outline, giving them a three-dimensional or wireframe appearance.

# HELIX

Hey! I'll be talking about Helix today. Helix is a library that enables you to write Rust extensions for Ruby which blur the line between Ruby and Rust.



Stanko Krtalic Rusendic

 [github.com/stankek](https://github.com/stankek)

 [@monorkin](https://twitter.com/monorkin)

 [hi@stanko.io](mailto:hi@stanko.io)

 [stanko.io](https://stanko.io)

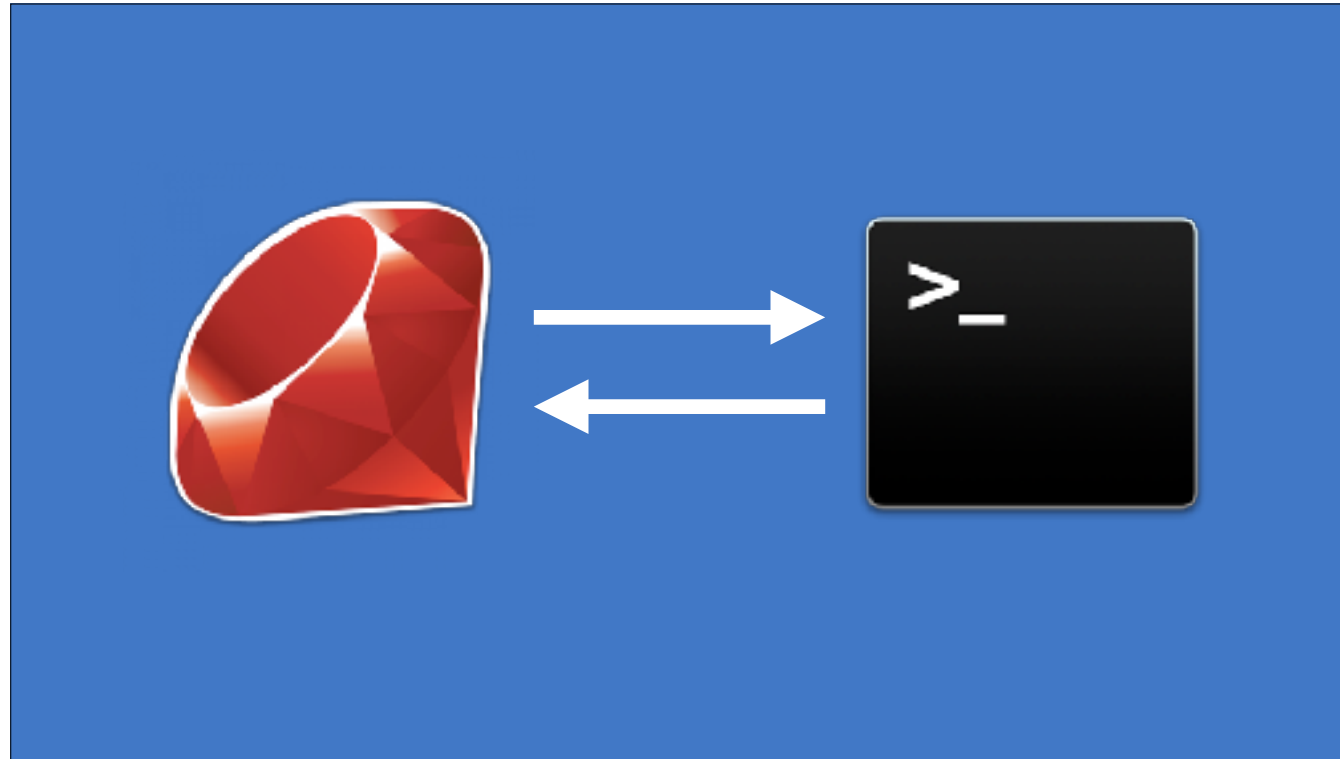
Before we continue, a quick introduction.

My name is Stanko. I am a software engineer at FloatingPoint.

I mostly work with Ruby, Elixir and Rust.



So... I guess that some of you are asking your selfs what's a Ruby extensions.  
Excellent question!



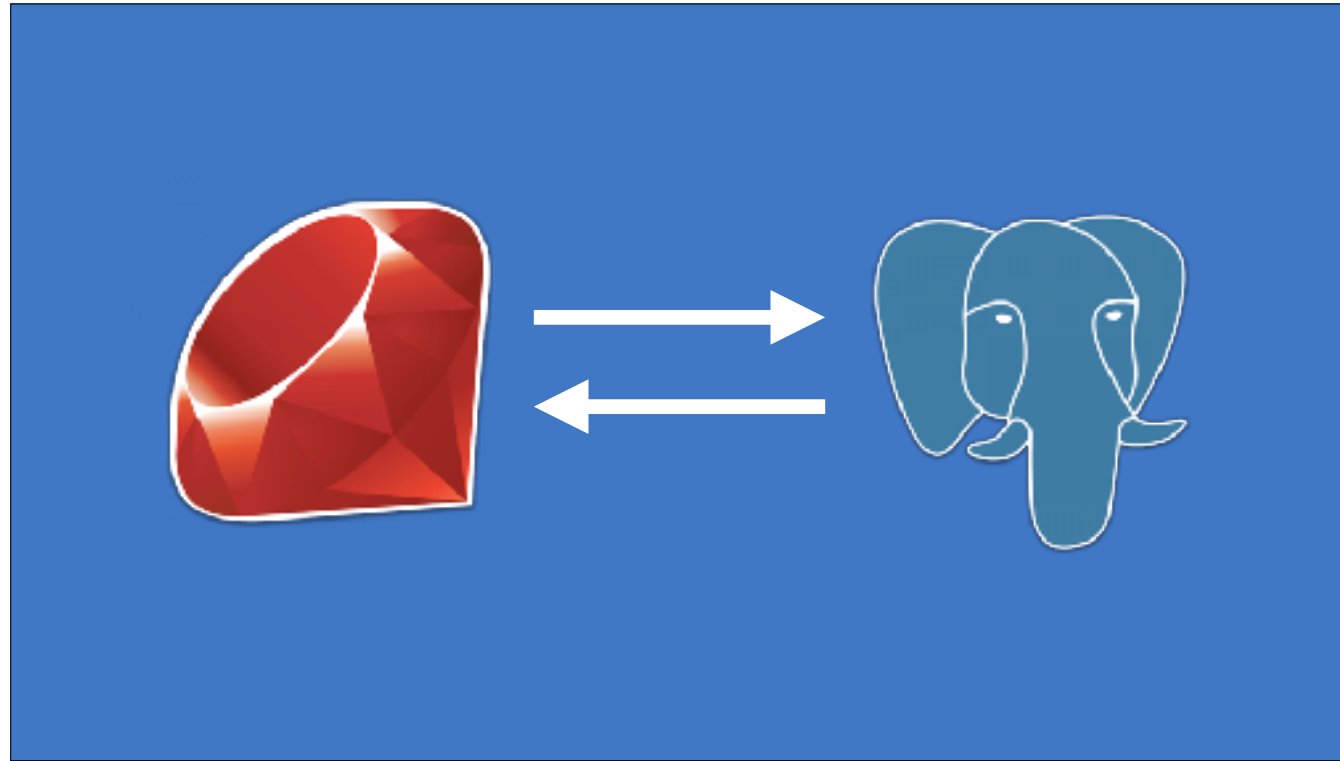
Extensions are a way for Ruby to interface with code that isn't written in Ruby.

This concept is more commonly known as FFI or Foreign Function Interface. As the name implies, the function being called is foreign, from another realm or language, perhaps it speaks French?

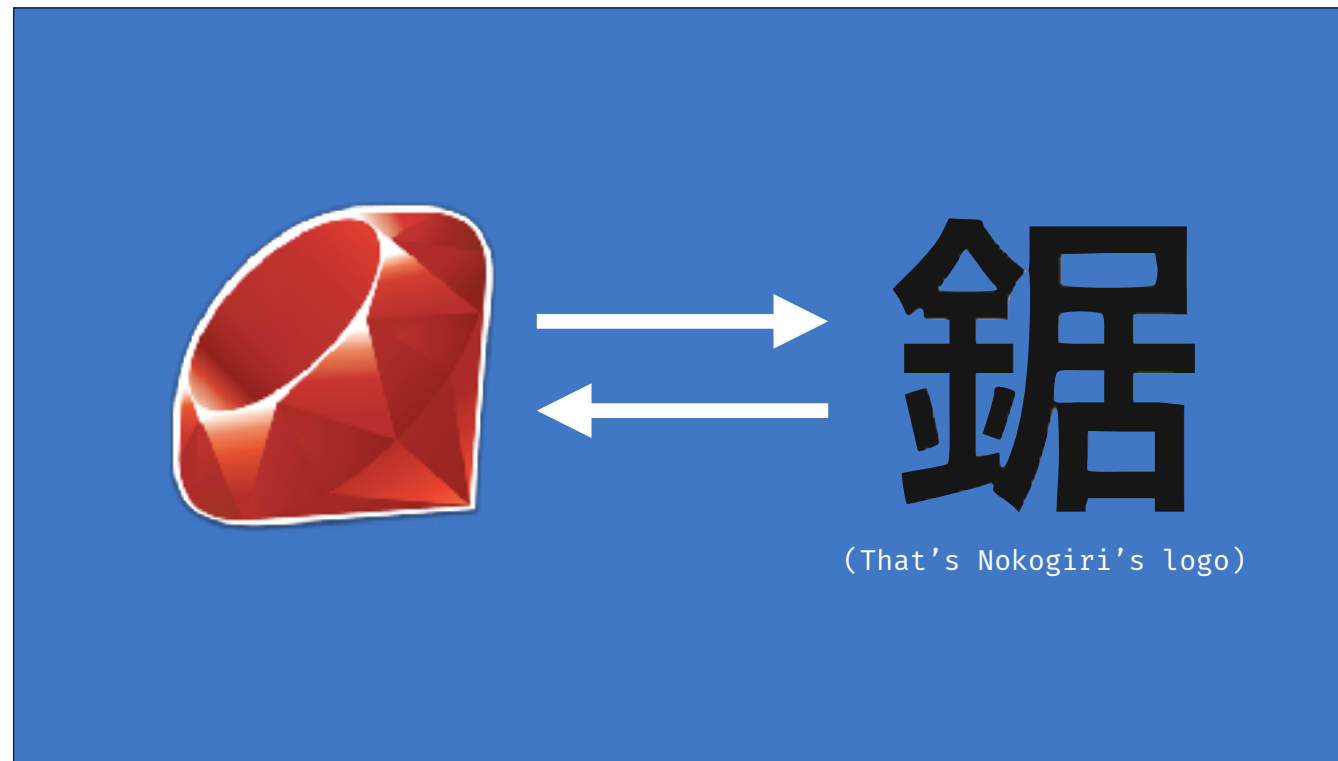
But why? 🤯

Why would you do such a thing?

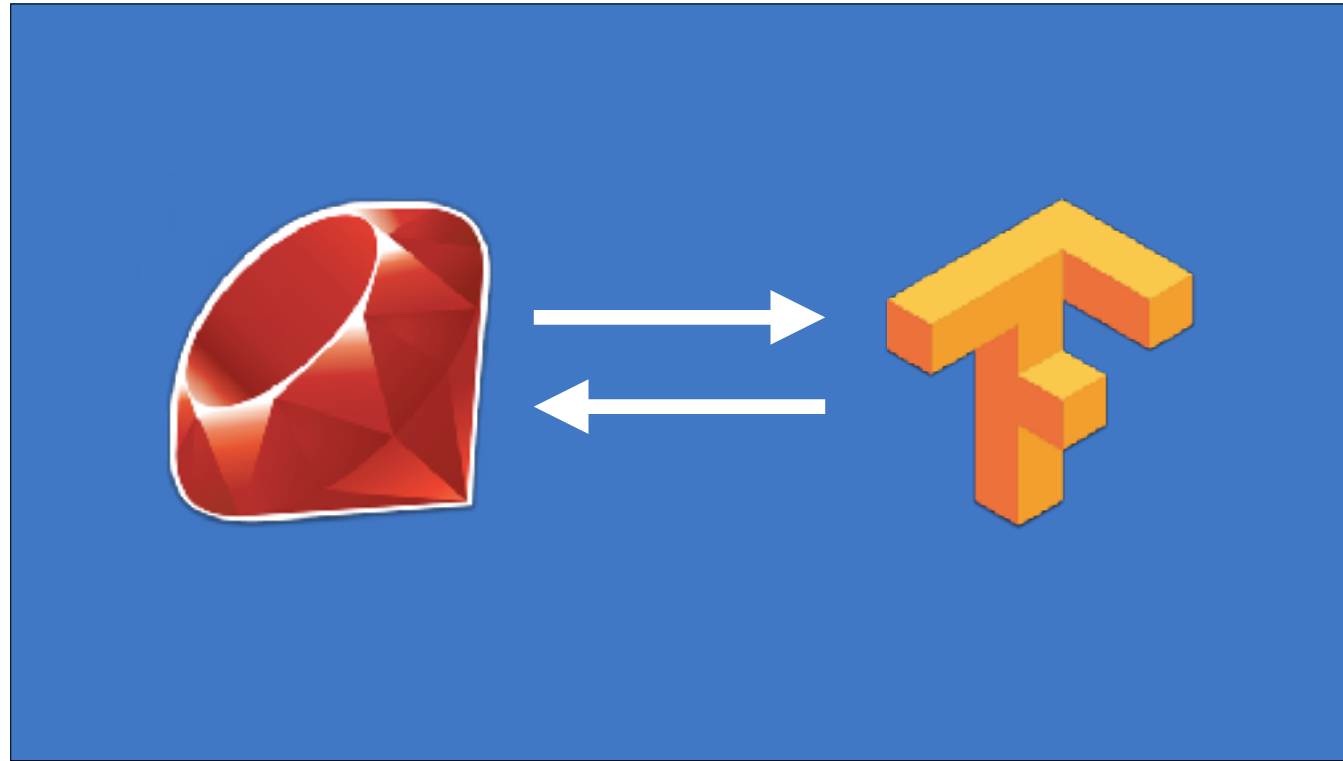
Leave the safety, warmth and loving embrace of Ruby for another, filthy, language!?



Sometimes you need to use a complex protocol which has a reliable library in another language.

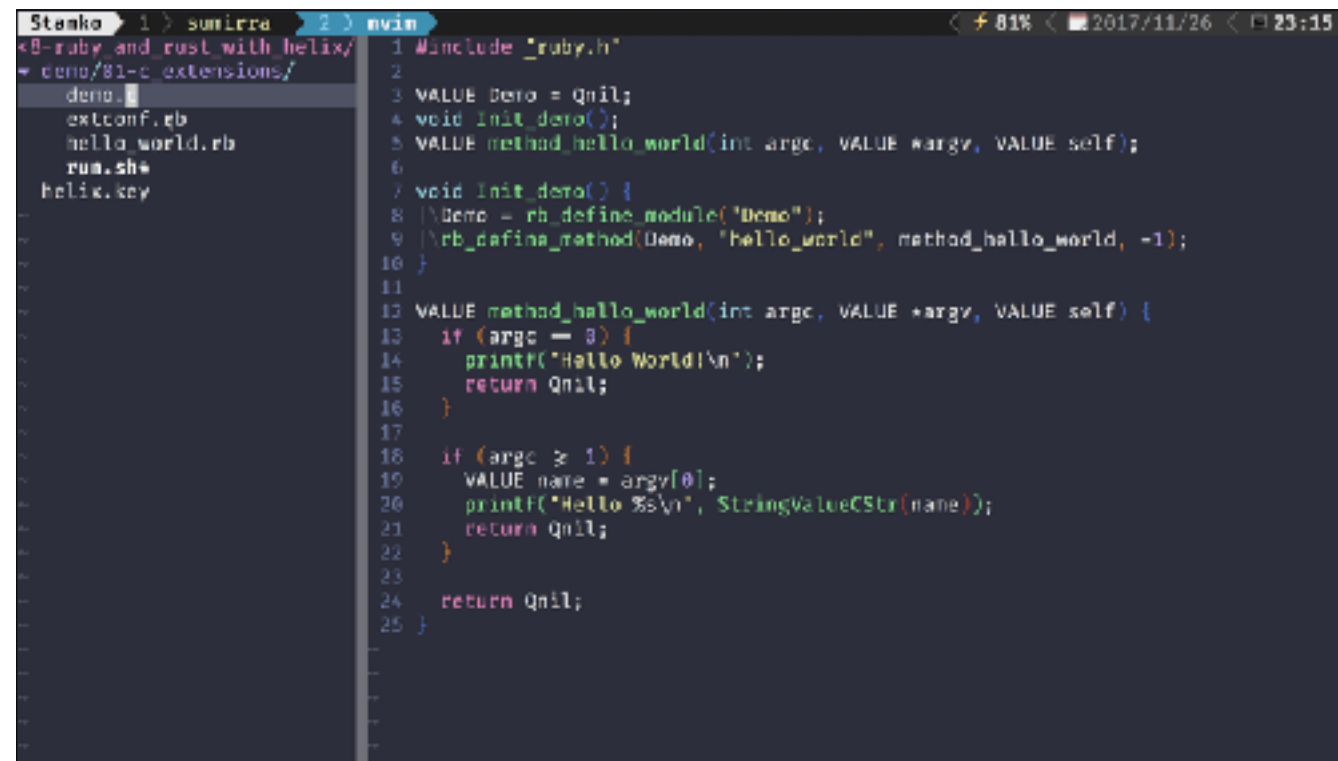


Sometimes Ruby is just too slow and you need that extra edge.  
For 99% of you wondering what that logo is, it's Nokogiri, a XML parsing and manipulation library.



But most of the time your manager is a jerk and closes a contract with a company that has proprietary tech and doesn't have a Ruby library, but is the current buzzword, web 3.0, technology. So now you have to use that library but don't want to spend 30 days rewriting it to Ruby so you decide to do the pragmatic thing and write a Ruby extension for it.





The screenshot shows a code editor with two panes. The left pane displays a file explorer view of a directory named 'demo/81-c extensions/'. It contains the following files: 'demo.c', 'extconf.rb', 'hello\_world.rb', 'run.sh', and 'hello.key'. The right pane shows the C code for 'demo.c'. The code includes 'ruby.h', defines a 'Demo' module, and implements a 'hello\_world' method that prints 'Hello World' or 'Hello' followed by an argument, depending on the number of arguments passed.

```
1 #include "ruby.h"
2
3 VALUE Demo = Qnil;
4 void Init_demo();
5 VALUE method_hello_world(int argc, VALUE *argv, VALUE self);
6
7 void Init_demo() {
8     \Demo = rb_define_module("Demo");
9     \rb_define_method(Demo, "hello_world", method_hello_world, -1);
10 }
11
12 VALUE method_hello_world(int argc, VALUE *argv, VALUE self) {
13     if (argc == 0) {
14         printf("Hello World!\n");
15         return Qnil;
16     }
17     if (argc >= 1) {
18         VALUE name = argv[0];
19         printf("Hello %s\n", StringValueCStr(name));
20         return Qnil;
21     }
22     return Qnil;
23 }
24
25 }
```

To create a Ruby extensions you only need two files.

1. The C code that implements the logic
2. An extconf.rb file that specifies how to build the extension

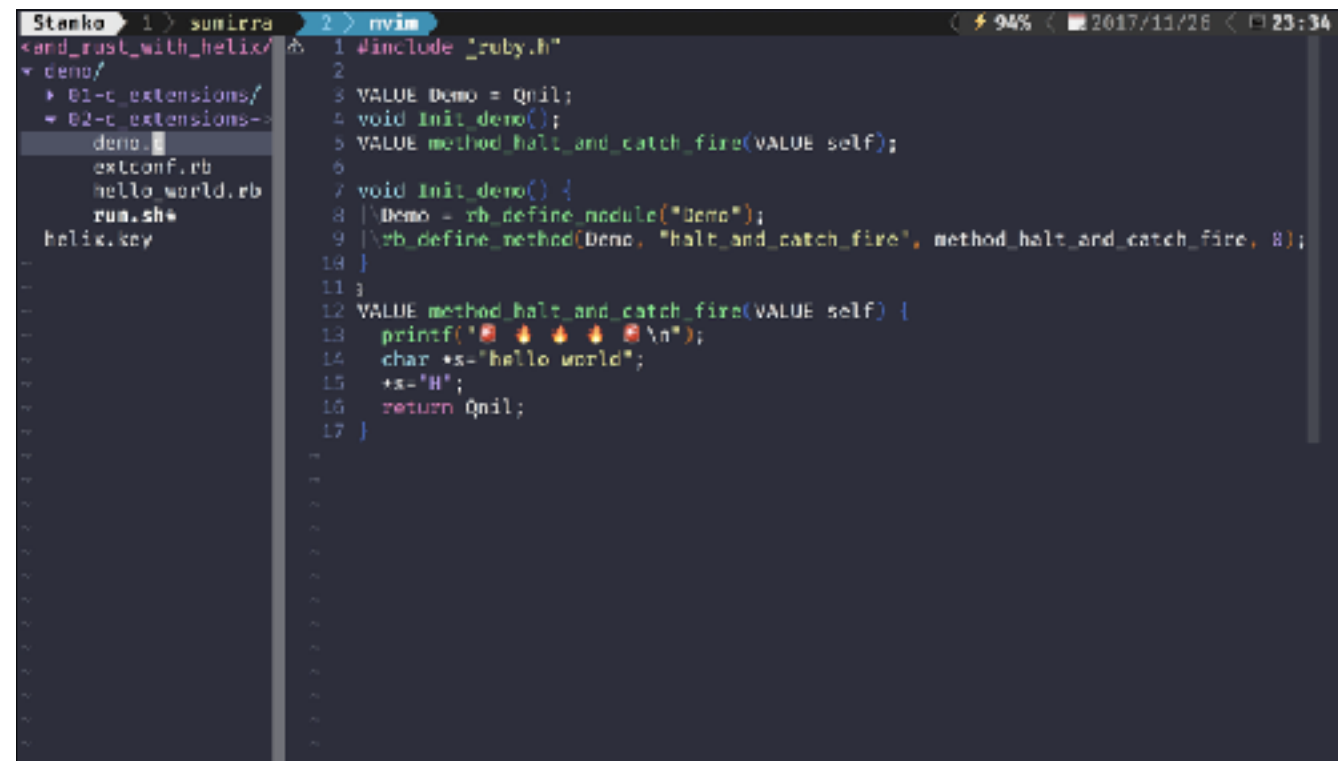
And for this example, a file that consumes to extensions

This is a really simple demo. It provides us with the Demo module, which has one method `hello\_world` which accepts any number of arguments.

If no arguments are given, it prints Hello World.

If at least one argument is given it prints Hello and then the argument

Also, if something unexpected is passed the foreign function errors as expected from a Ruby method.



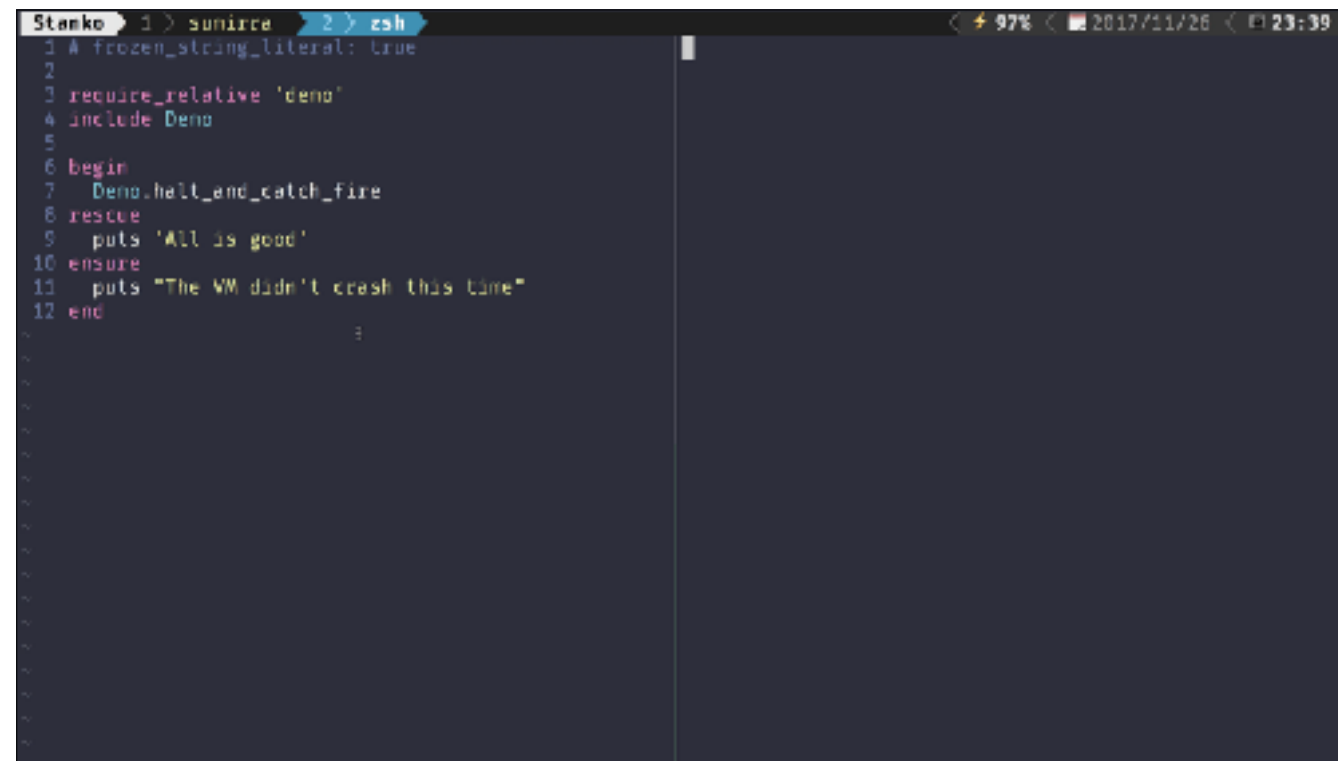
```
Stanko 1 > sunirra 2 > nvim 94% 2017/11/26 23:34
<and_rust_with_helix/ 1 #include "_ruby.h"
2
3 VALUE Demo = Qnil;
4 void Init_demo();
5 VALUE method_halt_and_catch_fire(VALUE self);
6
7 void Init_demo() {
8   |Demo = rb_define_module("Demo");
9   |rb_define_method(Demo, "halt_and_catch_fire", method_halt_and_catch_fire, 8);
10 }
11
12 VALUE method_halt_and_catch_fire(VALUE self) {
13   printf("💣💣💣💣💣\n");
14   char *s="hello world";
15   *s='H';
16   return Qnil;
17 }
```

But with all the benefits that extensions bring, they also come with a huge asterix.

Look at this code... Again the same drill as last time, but this time the C code has a bug.

[Wait for the video to end]

This isn't your standard Ruby exception, this is a Ruby VM error. Just because of an error, the whole Ruby process crashed.



The screenshot shows a terminal window titled 'Stanko' with tabs for '1 > sunirca' and '2 > zsh'. The active tab is '2 > zsh'. The terminal displays the following Ruby code:

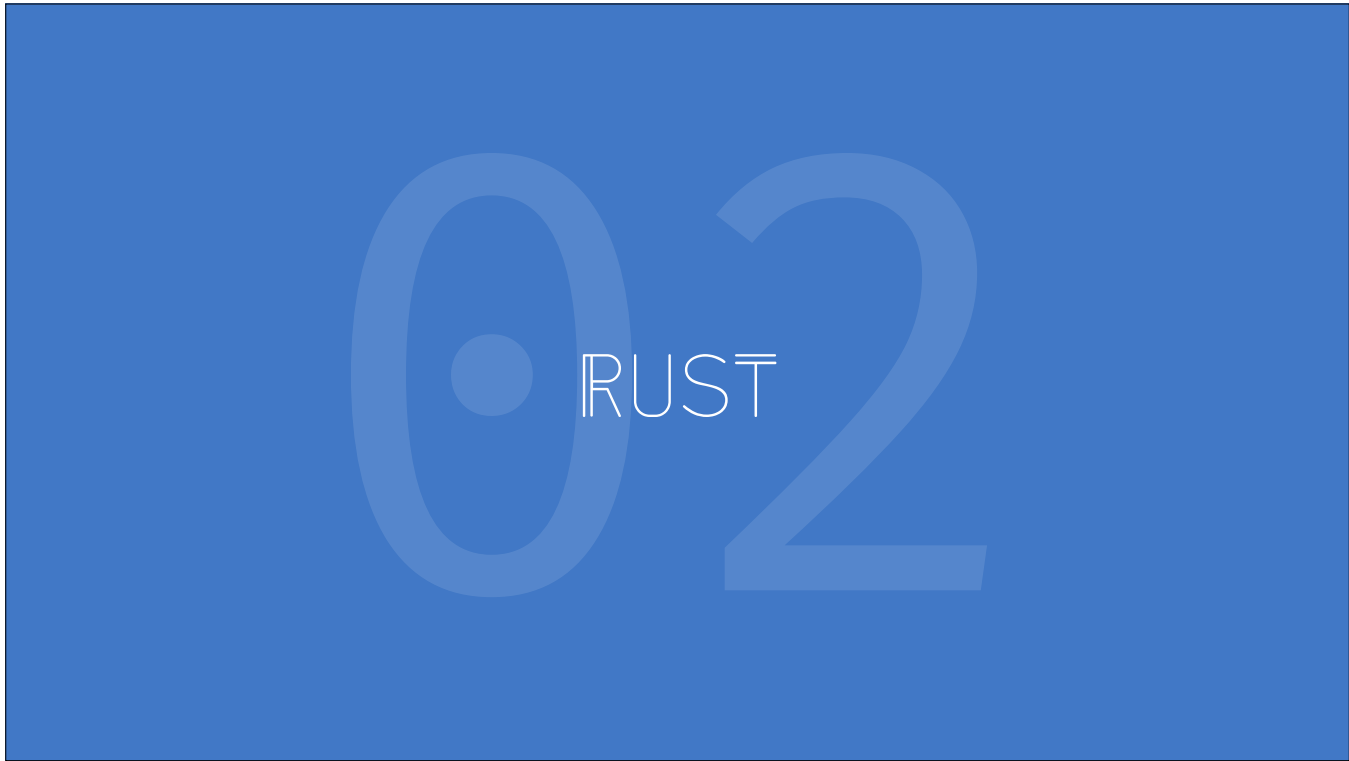
```
1 # frozen_string_literal: true
2
3 require_relative 'deno'
4 include Deno
5
6 begin
7   Deno.halt_and_catch_fire
8 rescue
9   puts 'All is good'
10 ensure
11   puts "The VM didn't crash this time"
12 end
```

The terminal window has a dark background with light-colored text. The status bar at the top right shows a battery icon at 97%, the date 2017/11/26, and the time 23:39.

And you can't prevent this in any way. If it happens, it happens, your app simple crashes and you can't do nothing about it.

[Wait for the video to end]

As you might imagine this can turn out to be a problem...



Now we come to the Rust part.



A few month back I held a talk at WebCamp Zagreb about developing web applications with Rust.  
After the talk I got an interesting question from the audience.



To be specific, from this guy. Which you might recognise.

Why do people use for Rust  
when writing FFI?



SAFETY

To sum everything up. It's because safety.



A program written in Rust  
can't crash unexpectedly

Memory safety  
(Data) Race condition safe  
Performance

Rust provides a lot of guarantees without performance penalties. When you combine all those, you can say that an app written in Rust can only have an algorithmical errors.

# Results / Options

## Ownership

## Borrowing

## Lifetimes

This kind of magic is accomplished with the following concepts.

Results and options. They are comparable to Haskell's Maybe monad.

Ownership and borrowing, which are unique to Rust, solve data race conditions and make threading more sane.

While Lifetimes abolish the need for a garbage collector.

```
1 pub enum Option<T> {  
2     None,  
3     Some(T),  
4 }  
5  
6 pub enum Result<T, E> {  
7     Ok(T),  
8     Err(E),  
9 }
```

I'll just quickly explain Options and Results. Basically, an Option is a type that wraps another type, it's rather a wrapper around that type or a None type. Results are similar, they wrap two types, one represents a success state and the other a failure state.

```


1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) => message,
9         Err(error) => return println!("ERROR: {}", error)
10    };
11
12    println!("{}", name, message)
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }

```

Here is a piece of code to demonstrate Options and Results in action

```
1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) => message,
9         Err(error) => return println!("ERROR: {}", error)
10    };
11
12    println!("{}", message);
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }
```

This method can return a string or nothing



We specify that a function returns an Option that wraps a String.

1 fn main() {  
 2 let name = match get\_name() {  
 3 Some(name) => name,  
 4 \_ => return println!("ERROR: No name given!")  
 5 };  
 6  
 7 let message = match build\_message(name) {  
 8 Ok(message) => message,  
 9 Err(error) => return println!("ERROR: {}", error)  
 10 };  
 11  
 12 println!("{}", name, message)  
 13 }  
 14  
 15 fn get\_name() -> Option<String> {  
 16 }  
 17  
 18 fn build\_message() -> Result<String, String> {  
 19 }

← Check what the method returned

In the function consuming the result we do pattern matching to figure out if the returned option returned a String or it returned nothing.

```

1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) => message,
9         Err(error) => return println!("ERROR: {}", error)
10    };
11
12    println!("{}", name, message)
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }

```

If a string was returned, unwrap it

This branch will get called if the Option contained a string



```

1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) => message,
9         Err(error) => return println!("ERROR: {}", error)
10    };
11
12    println!("{}", name, message)
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }

```

If a string wasn't returned, Error

While this branch will get called if the Option was empty. The `\_` is a catch-all clause.

```

1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) => message,
9         Err(error) => return println!("ERROR: {}", error)
10    };
11
12    println!("{}", name, message)
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }

```

Results function in a similar way.

```

1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) => message,
9         Err(error) => return println!("ERROR: {}", error)
10    };
11
12    println!("{}", name, message)
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }

```

Results have two types



Bu results have two types. The first is the OK or success type, that's the type we return if everything went Ok. The second is the error type, it's returned if an error occurred.

```

1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) => message,
9         Err(error) => return println!("ERROR: {}", error)
10    };
11
12    println!("{}", name, message)
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }

```

Checks if the result was Ok or an Error

Again, we do pattern matching to figure out what happened. But this time we have additional information in the case of an error.

```

1 fn main() {
2     let name = match get_name() {
3         Some(name) => name,
4         _ => return println!("ERROR: No name given!")
5     };
6
7     let message = match build_message(name) {
8         Ok(message) -> message,
9         Err(error) -> return println!("ERROR: {}", error)
10    };
11
12    println!("{}", name, message)
13 }
14
15 fn get_name() -> Option<String> {
16 }
17
18 fn build_message() -> Result<String, String> {
19 }

```

The option type is a replacement for the NULL value (since Rust doesn't have one), while Results are a replacement for Error throwing (since Rust doesn't have a throw-catch mechanism).

The compiler will complain if you don't unwrap Option and Result values, that's why a Rust program can't fail unexpectedly. It could fail if we explicitly told the program to panic. This makes Rust a much better candidate for FFI.



I won't go into ownership, borrowing and lifetimes since they could be hard to swallow all at once. But they are mechanisms which provide thread safety and race condition safety out-of-the-box.

Now that we've established that Rust is an excellent candidate for FFI, and thereby Ruby extensions, let's learn how to use it from Ruby

```
cargo new demo
```

First we create a new Rust project by running `cargo new`

```
demo_  
├── Cargo.lock  
├── Cargo.toml  
└── src  
    └── lib.rs
```

1 directory, 3 files

This gives us the Cargo files used for dependency management and a `src/lib.rs` file which is the entry point of our Rust code




```
1 [package]
2 name = "demo"
3 version = "0.1.0"
4 authors = ["Stanko Krtalić <stanko.krtalic@gmail.com>"]
5
6 [lib]
7 crate-type = ["cdylib"]
8
9 [dependencies]
10 helix = "0.7.2"
```



We specify that we are building a dynamically linked library. This is necessary for FFI.

```
1 [package]
2 name = "demo"
3 version = "0.1.0"
4 authors = ["Stanko Krtalić <stanko.krtalic@gmail.com>"]
5
6 [lib]
7 crate-type = ["cdylib"]
8
9 [dependencies]
10 helix = "0.7.2"
```



And we add Helix as a dependency

```
1 #[macro_use]
2 extern crate helix;
3
4 ruby! {
5     class Demo {
6         def hello_world(name: Option<String>) {
7             let name = match name {
8                 Some(string) => string,
9                 _ => "World".to_string()
10            };
11
12            println!("Hello {}", name);
13        }
14    }
15 }
```

Then we import Helix and implement our logic in lib.rs

```
1 #[macro_use]
2 extern crate helix;
3
4 ruby! {
5     class Demo {
6         def hello_world(name: Option<String>) {
7             let name = match name {
8                 Some(string) => string,
9                 _ => "World".to_string()
10             };
11
12             println!("Hello {}", name);
13         }
14     }
15 }
```



So, we import Helix here

```

1  #[macro_use]
2  extern crate helix;
3
4  ruby! {
5      class Demo {
6          def hello_world(name: Option<String>) {
7              let name = match name {
8                  Some(string) => string,
9                  _ => "World".to_string()
10             };
11
12             println!("Hello {}", name);
13         }
14     }
15 }

```

It not only provides us with an interface to the Ruby FFI, but it also introduces a macro that allows us to basically write Ruby style code inside of Rust. For those wondering, Macros are Rust's way of implementing Metaprogramming.

```
1 #[macro_use]
2 extern crate helix;
3
4 ruby! {
5     class Demo {
6         def hello_world(name: Option<String>) {
7             let name = match name {
8                 Some(string) => string,
9                 _ => "World".to_string()
10             };
11
12             println!("Hello {}", name);
13         }
14     }
15 }
```


To explain the code.

First we define a class named Demo

```
1 #[macro_use]
2 extern crate helix;
3
4 ruby! {
5     class Demo {
6         def hello_world(name: Option<String>) {
7             let name = match name {
8                 Some(string) => string,
9                 _ => "World".to_string()
10             };
11
12             println!("Hello {}", name);
13         }
14     }
15 }
```

Then we define a method named `hello_world` that accepts an argument named `name`, that can be either a `String` or `nil`


```
1 #[macro_use]
2 extern crate helix;
3
4 ruby! {
5     class Demo {
6         def hello_world(name: Option<String>) {
7             let name = match name {
8                 Some(string) => string,
9                 _ => "World".to_string()
10            };
11
12            println!("Hello {}", name);
13        }
14    }
15 }
```



Then we do pattern matching to detect if the passed name argument was nil. If it was, we set the name variable to the value “World”, else we unwrap the passed String and use it as the value for the name variable. And yes, this shadows the passed name argument.



```
1 #[macro_use]
2 extern crate helix;
3
4 ruby! {
5     class Demo {
6         def hello_world(name: Option<String>) {
7             let name = match name {
8                 Some(string) => string,
9                 _ => "World".to_string()
10             };
11
12             println!("Hello {}", name);
13         }
14     }
15 }
```



Finally, we print out the result. The curly braces are used for interpolation in Rust.

```
demo_  
├── Cargo.lock  
├── Cargo.toml  
├── Gemfile  
├── demo.gemspec  
└── src  
    └── lib.rs
```

1 directory, 5 files

In the end we add a Gemfile and gemspec file to convert the directory to a Ruby Gem, thus making it callable from Ruby.

```
demo_  
├── Cargo.lock  
├── Cargo.toml  
├── Gemfile  
├── demo.gemspec  
└── src  
    └── lib.rs
```

1 directory, 5 files


So, these two.

The gemfile references the gemspec file.

```
1 Gem::Specification.new do |s|
2   s.name = 'demo'
3   s.version = '1.0.0'
4   s.authors = ['Stanko Krtalic Rusendic <stanko.krtalic@gmail.com>']
5   s.summary = "A Helix demo project"
6   s.files = Dir['{lib/**/*,[A-Z]*}']
7
8   s.platform = Gem::Platform::RUBY
9   s.require_path = 'lib'
10
11   s.add_dependency 'helix_runtime', '~> 0.7.2'
12 end
```


And the gem spec file lists a single dependency - Helix

```
1 Gem::Specification.new do |s|
2   s.name = 'demo'
3   s.version = '1.0.0'
4   s.authors = ['Stanko Krtalic Rusendic <stanko.krtalic@gmail.com>']
5   s.summary = "A Helix demo project"
6   s.files = Dir['{lib/**/*,[A-Z]*}']
7
8   s.platform = Gem::Platform::RUBY
9   s.require_path = 'lib'
10
11   s.add_dependency 'helix_runtime', '~> 0.7.2'
12 end
```



So, this all is standard. That's what you'll get if you run `bundler new`

```
1 Gem::Specification.new do |s|
2   s.name = 'demo'
3   s.version = '1.0.0'
4   s.authors = ['Stanko Krtalic Rusendic <stanko.krtalic@gmail.com>']
5   s.summary = "A Helix demo project"
6   s.files = Dir["{lib/**/*,[A-Z]*}"]
7
8   s.platform = Gem::Platform::RUBY
9   s.require_path = 'lib'
10
11   s.add_dependency 'helix_runtime', '~> 0.7.2'
12 end
```



This is the only line added to the file.

```
rake build
```

We run rake build to build our gem. And that's it.

```
3 # Optional and KV arguments are not yet supported
4 Demo.hello_world(nil)
5 Demo.hello_world('Ruby Zagreb!')
6
7 begin
8   Demo.hello_world(Object.new)
9 rescue TypeError
10   puts 'Rescued from TypeError'
11 end
```

Now we can call out Rust code from Ruby.



```
* Press ? for help
.. (up a dir)
<d_rust_with_helix/
[X]demo/
  01-c_extension>
  02-c_extension>
  03-rust_examp>
  [X]04-helix/
    [X]crates/de>
      lib/
      src/
        lib.rs
      [X]target/
        Cargo.lock
        Cargo.toml
        demo.gemsp>
        Gemfile
        Gemfile.lo>
        Rakefile
        tags
NERD N_ ←helix/crates/demo/src/lib.rs 6% 1: 1
```

```
1 [macro_use]
2 extern crate helix;
3
4 ruby! {
5   class Demo {
6     def hello_world(name: Option<String>) {
7       let name = match name {
8         Some(string) => string,
9         _ => "World".to_string()
10      };
11
12      println!("Hello {}", name);
13    }
14  }
15 }
```

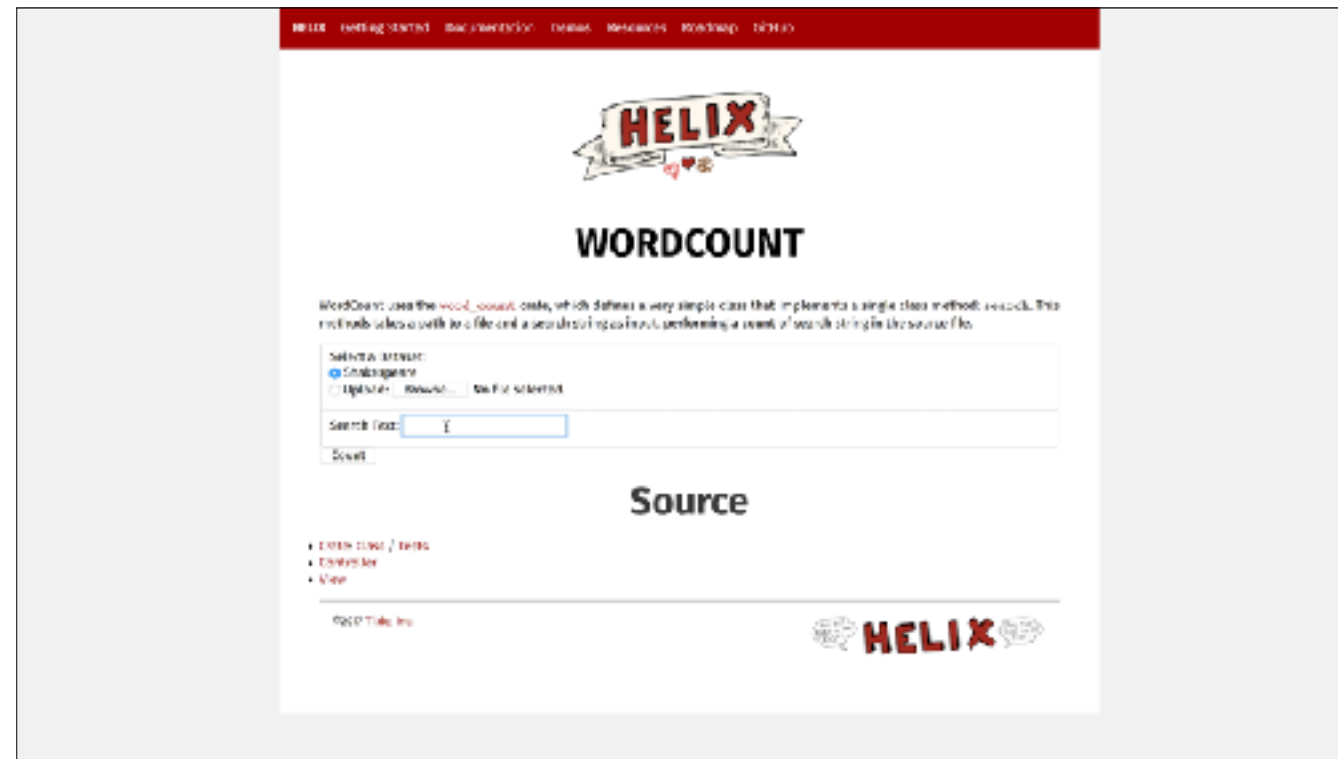
Here I moved the Rust project to a new directory called crates.  
I added a file that loads Bundler and a Gemfile where I specify a path to my Ruby-Rust project.

```
rails generate helix:crate demo
```

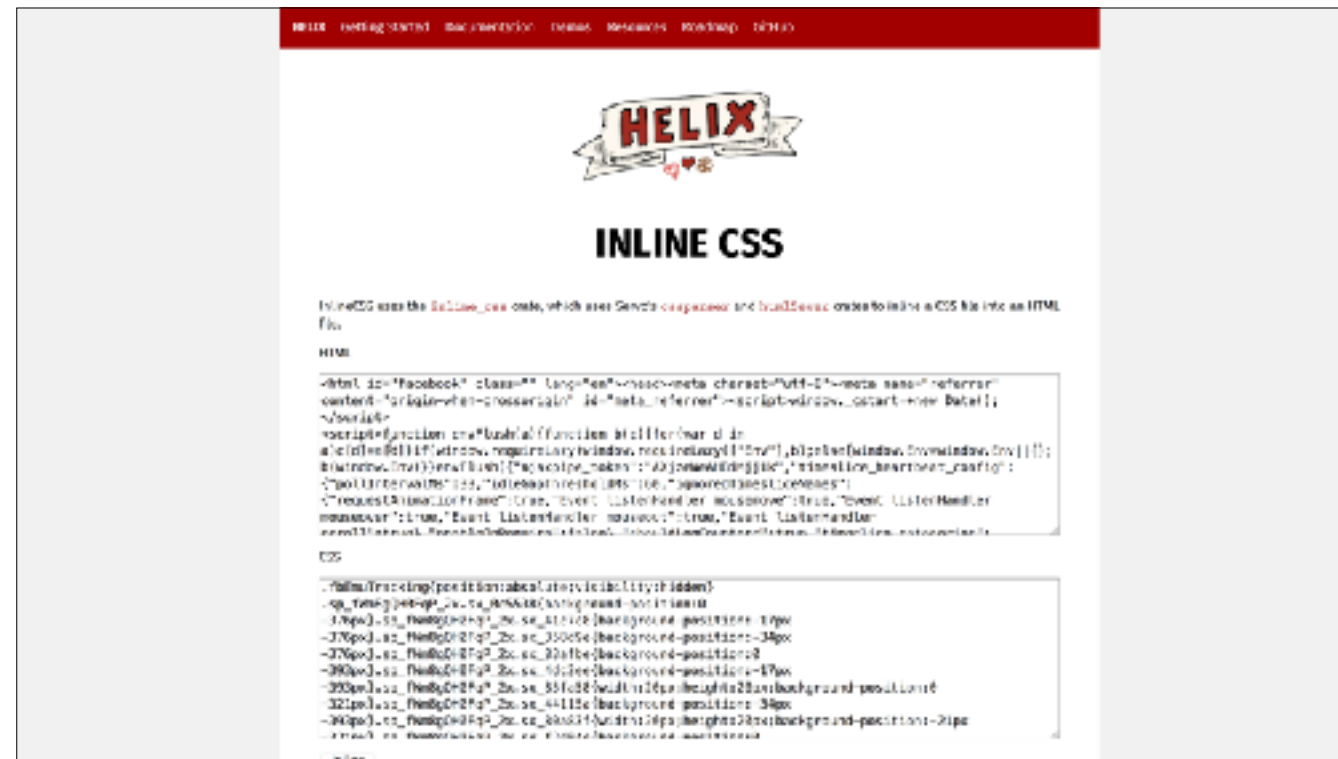
If this all is too tedious for you then you can also call the generator to do all the work for you.

# 04 WHEN TO USE IT

But when do we use Rust in our Ruby projects?



One good example is data processing. Since Rust's speed is comparable to C it can do data analysis a lot quicker than Ruby. Here we search for the number of occurrences of the word "a" in all of Shakespeare's works. As you can see, it took Rust 60ms to go through all of Shakespeare's works and count 12k occurrences of the word "a"



Here is another good use-case. Inlining CSS. Mozilla, the organisation behind Rust, recently put out a lot of open-source projects for HTML and CSS parsing written in Rust. These projects are used in the latest builds of Firefox and they power Firefox's Quantum rendering engine.

I copy-pasted my Facebook feed here. There's 200KB of HTML and about 5MB of CSS and JS. This Rust code inlines all of it in just a second.

(for the observant people, some CSS classes are still present because there is already inlined CSS at the bottom of the HTML file which I didn't notice at the time of recording)

No KV or optional arguments  
Can't write Modules  
No cross-compilation

Though there are some downsides to using Helix.

[read list]

If you need these features and still want to use Rust then I would recommend you take a look at RuRu, which is a wrapper around Ruby's C library. It doesn't provide any kind of niceties as Helix but it has feature parity with the latest release of Ruby's C library.

## Non-standard compiler\*

The other downside is that not all machines come with Rust pre-installed. Starting from October this year all builds of Debian, RedHat Linux, Fedora and OpenSuse come with Rust pre-installed, but that's not even close to the number of distributions that support GCC. This could potentially be solved when Rust gets ported to GCC or rust gets more adoption, but still, it's not such a big issue to install Rust since it's available on all package managers.

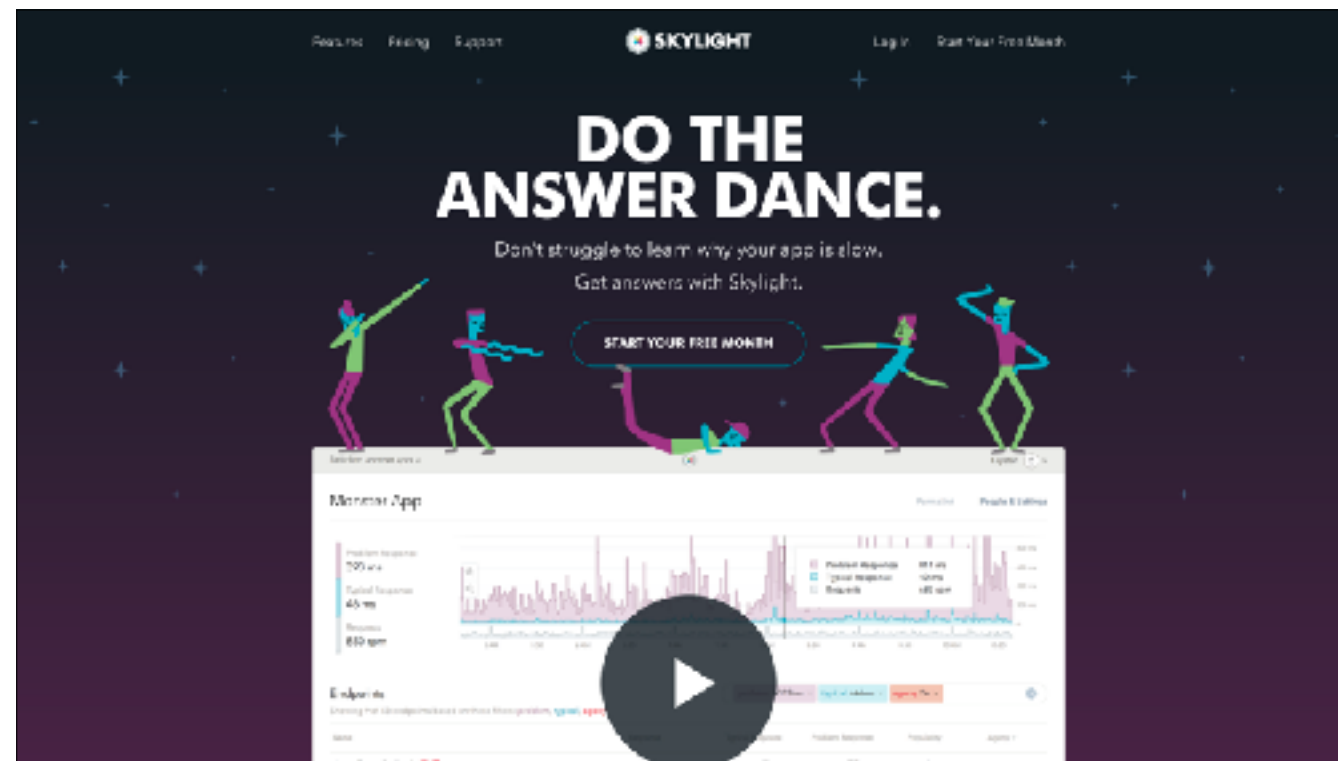
# Safety and speed

I would always recommend Rust over C or Go for any kind of application. It's as performant as C, has no garbage collector, has a great ecosystem and documentation, and most of all it's safe. You can be sure that your extension won't bring the whole VM down because of a typo.





Here is a number of companies that run and trust Rust in production.



Lastly, if you use Skylight to monitor your app, then you are already running Rust code through Helix in your application.

# QUESTIONS


 [github.com/stankek](https://github.com/stankek)

 [@monorkin](https://twitter.com/monorkin)

 [hi@stanko.io](mailto:hi@stanko.io)

Thank you for your patience. If anybody has questions I'll accept them now.  
If you are shy, you can also contact me via Twitter or Email.



```
1 
2 fn sign_in(username: String, password: String) → Result<User, String> {
3     if username == "user".to_string() && password == "password".to_string() {
4         return Ok(
5             User::new(username, password)
6         );
7     }
8     return Err("No such user found");
9 }
10 }
```

Username and password are deallocated

And to briefly explain lifetimes. Basically the compiler notices when a variable isn't used anymore, and implicitly deallocates it after the line it's last used

```
1 fn main {  
2     let name = "petar".to_string();  
3     let amne = shuffle(name); // ← The function shuffle became the owner of  
4                               // the name variable  
5  
6     println!("Original: {}", name); // ← Errors during compilation  
7                                     // We tried to use a variable we don't own  
8     println!("Shuffled: {}", amne);  
9 }
```

Ownership and borrowing aren't that easy to fully explain, but here it is in a nutshell. You can think of it as if you are borrowing a book. Once you give a book to a friend you can't use it any more until they give it back to you. The same applies to this example. Once we gave the variable called "name" to the "shuffle" function we don't own it anymore, and therefore we can't use it. The compiler won't let us. This resolves all data race conditions. I won't go into lifetimes here, but believe me that they are pure magic that abolishes the need for a garbage collector.