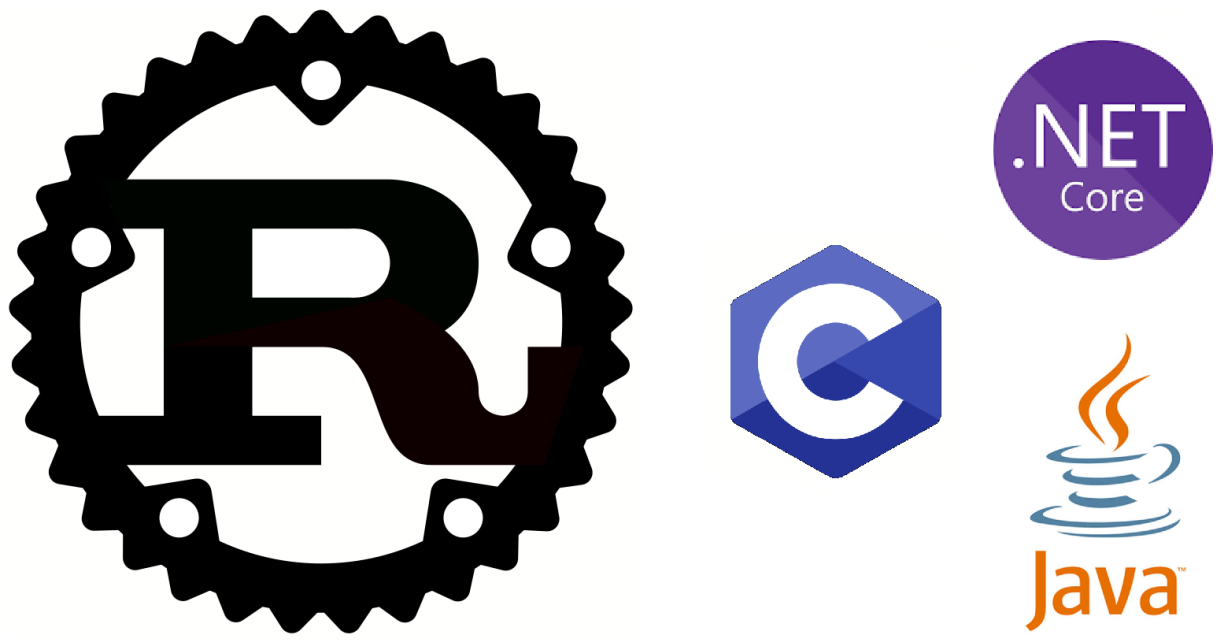
Binding Rust to other languages safely and productively

ÉMILE GRÉGOIRE AUGUST 10, 2020



When we made the decision to write our next generation of libraries in Rust, we knew we needed a solid approach for binding them to other languages. It may be some time before we have customers purchasing our libraries to use them in a Rust-only codebase. The majority of our customers will want to use the libraries in C/C++, .NET, or Java.

Writing the core implementation in Rust means more productivity, fewer errors, and certain safety guarantees compared to writing it in C++. This is incredibly valuable and worth the effort of building a binding solution.

From our clients perspective though, Rust can represent more of a hurdle. Binding any kind of native code into a higher level framework like Java or .NET is non-trivial and fraught with error. If we didn't take an extremely disciplined and maintainable approach to building such bindings, we might lose all the stability, safety, and performance benefits of writing our libraries in Rust in the first place.

Additionally, the bindings needed to feel as if they were written entirely in the target language. They needed to follow the proper conventions, naming, and patterns for that language.

Past Experience

We have extensive experience in providing bindings to native libraries and have learned some important lessons from it.

OpenDNP3, a library written in modern C++, exposes Java and .NET bindings. To ease the pain of interacting with Java Native Interface (JNI), we wrote a code generator in Java to generate error prone parts of the bindings and add type safety using the C++ type system. This code generator has proven to be extremely valuable, saving us a lot of time. However, there was still a lot of code to write by hand, and this resulted in many minor issues over the years. Also, the code generator was tightly coupled to OpenDNP3 and could only assist with the Java bindings.

The lessons we have learned from OpenDNP3 project are:

- Users are interested in using bindings, C# and Java mostly commonly.
- Writing a code generator for the repetitive and error-prone pieces of the binding code increases productivity and decreases maintenance.
- It would be nice if the code generator could model the entire external API to avoid having gaps between what's available in each target language.
- Considering the effort required to design and implement the code generator, the generator should be reusable across multiple projects to maximize our ROI.

Existing Solutions

There are already some solutions available to generate bindings in Rust. Unfortunately, each solution targets only a single language or has other serious drawbacks. For example, cbindgen generates C and not-so-natural C++ headers by parsing Rust code. Dotnet-bindgen generates C# bindings using P/Invoke. Each solution requires its own annotations and process to generate the code. This translates to many potential sources of inconsistency and errors, not to mention the time required to maintain each binding technology.

There are some projects to help writing the C FFI binding by eliminating some of the required glue-code. One interesting project is safer_ffi. This only solves part of the problem as it doesn't generate the bindings in languages other than C.

Then of course there's the venerable SWIG project. Swig parses your C headers along with additional metadata from an "interface file". It can generate bindings in many target languages. The downside is that there is no integration with Rust and the generated bindings are not idiomatic. We would end up with a product where the developer feels like they are programming C in a higher-level language. SWIG doesn't always make the best decisions, and we wanted tight control over the binding code that is emitted.

Goals

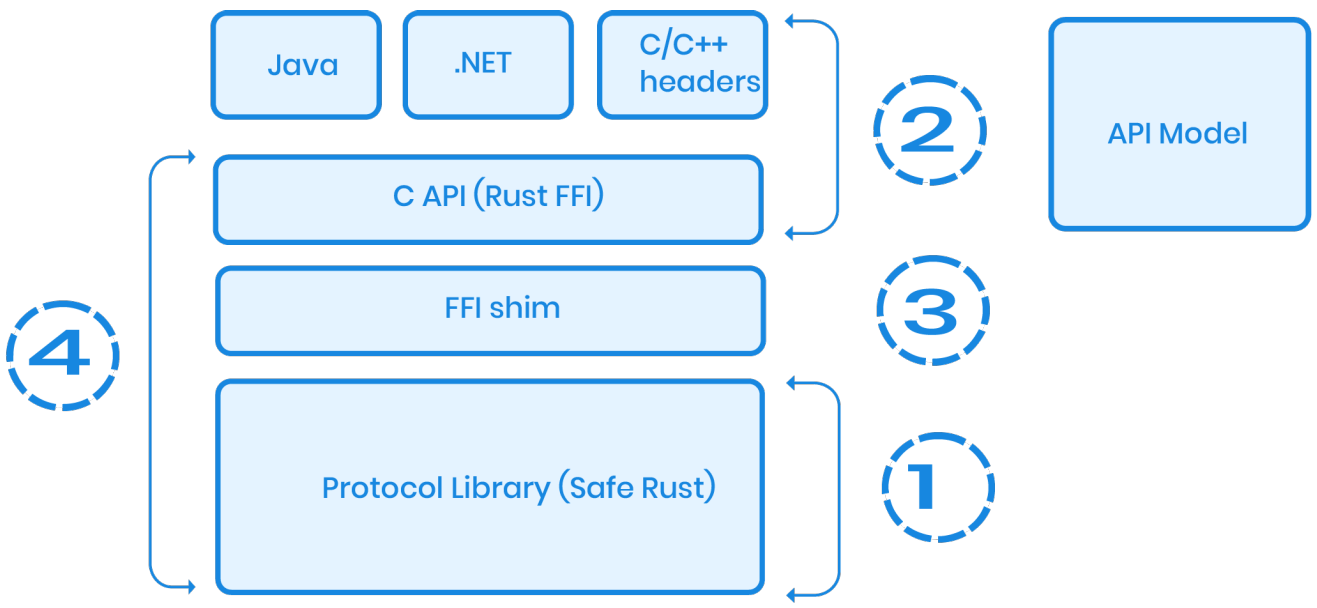
For our new projects, we decided to create a unified solution that we could reuse across multiple protocol stacks. With a single tool, we wanted to be able to generate bindings for multiple projects for multiple target languages.

The goals for this project were the following:

- Provide safe and idiomatic bindings to C and common object-oriented

languages.
- Minimize the amount of manually written code required to create bindings.
- Maximize the ROI of the generator by using the same code generator in every library we develop.

The only stable ABI that exists between Rust and any other language is the C ABI. We needed an approach that would allow us to simultaneously describe the C ABI, and how it maps to an abstract OO API.



The workflow

The expected workflow is the following:

1. The core domain library (e.g. a protocol stack) is written in idiomatic safe Rust with no consideration for how this API will map to C.
2. You then write an abstract model of an ideal C API for this library using a tool that we call oo_bindgen. This model also includes how the C API maps to an abstract object-oriented language. This model is then used to generate the C API (in Rust), C headers, and bindings for Java, .NET Core, etc.
3. You write the Rust code that binds the C API to the core library.
4. The core library, the FFI shim, and the C API are complied into a single shared library consumable from any language that understands the C ABI.

Each target language has its own backend generator that can emit all the required artifacts:

- For C, only headers are required, typically a single header file, e.g. “library.h”
- For .NET, this means emitting .cs files which internally use PInvoke to call the C ABI.
- For Java, this means emitting .java files which internally use JNI to call the C ABI.

From our perspective, this approach helps us focus on the really important part: the actual protocol stack. The effort required to write the schema and the glue-code is small compared to development of the underlying library. The advantages are numerous:

- We can support additional backend languages for ALL of our libraries by simply creating a new backend generator. If a client wants to be able to use our libraries in Go or Python at some point, there’s only a single thing we need to write to make that possible.
- We don’t repeat ourselves. We can even generate documentation stubs in the model that are then rendered using the appropriate tags for the target language!

- We ensure consistency between the actual C ABI, and our how our bindings call into the ABI.
- We ensure that our bindings always expose identical feature sets since they are rendered from the same model.

We have taken a O(N*M) problem where N is the number of libraries we have and M is the number of binding languages, and decoupled it into two problems of O(N) and O(M) instead.

Idiomatic Bindings

From the user perspective, the library looks and feels like any other library written in their language of choice. This proved to be one of the more challenging aspects of this endeavor, and is harder to achieve than one would first think.

For simple libraries, the existing code generators might be sufficient. However, our protocol stacks revolve around lots of asynchronicity. How one handles asynchronous operations varies greatly from on language to another.

For example, in C, asynchronous operation usually involves a function pointer with a context argument (i.e. `void*`) that gets called whenever the operation is complete. The following code sample is from the C API for our DNP3 library. It starts a read request and receives the result asynchronously via a callback.

```
// Callback method
void on_read_complete(read_result_t result, void* ctx)
{
    printf("ReadResult: %s\n", ReadResult_to_string(result));
}

// Create the request
request_t* request = request_new();
request_add_all_objects_header(request, Variation_Group10Var0);
request_add_all_objects_header(request, Variation_Group40Var0);

// Create the callback
read_task_callback_t cb =
{
    .on_complete = &on_read_complete,
    .ctx = NULL,
};

// Send the request, and receive the result asynchronously via the
callback
association_read(association, request, cb);

// Delete the request, it's copied internally
request_destroy(request);
```

However, in C#, best practice nowadays is to use C#'s excellent `async`/`await` functionality which revolves around functions returning values of type `Task<T>`. Our code generator for C# transforms the C callback API into .NET Tasks which can then be awaited.

```
// Create the request
var request = new Request();
request.AddAllObjectsHeader(Variation.Group10Var0);
request.AddAllObjectsHeader(Variation.Group40Var0);

// Send the request asynchronously
var result = await association.Read(request);

// Print the result
Console.WriteLine($"Result: {result}");
```

In Java, asynchronous operations are idiomatically represented by a `CompletionStage` that can be composed to chain asynchronous computations together and schedule them on a thread pool. The same example in Java would look like this:

```
// Create the request
Request request = new Request();
request.addAllObjectsHeader(Variation.GROUP10_VAR0);
request.addAllObjectsHeader(Variation.GROUP40_VAR0);

// Send the request asynchronously and print the result when it completes
association.read(request).thenAccept(
  result -> System.out.println("Result: " + result)
);
```

A development team can be working on either end of the wire and our products will be easy to integrate. For example, your team might be integrating a new protocol into your supervisory system running on Microsoft .NET or be writing the firmware of a resource-constrained field device in C, and they both can efficiently use the same underlying library with all the safety and performance benefits Rust has to offer.

Binding with Confidence

Because the code generator does all the hard work and is shared across all our projects, we expect it to lower the risk of bugs between the target language and the core library.

To further make sure that our binding code is safe and robust, we develop an extensive test suite that gets executed in all target languages on as many target platforms as possible.

RUST BINDINGS

Reliable and secure software for critical infrastructure

stepfunc

stepfunction_io

Address

395 SW Bluff Drive

Suite 10

Bend, OR 97702

info@stepfunc.io

+1-919-428-1002

Navigation

Products

Services

Blog

About Us

Contact