GiGa infosystems

Generating ergonomic C++ APIs using Rustdoc, procedural macros, and Serde

Björn Wieczoreck, RustLab, 10.11.2024

A little bit about myself

It’s me, hi!

- Masters in Geoinformationscience
- Work for GiGa infosystems GmbH since 2017
- Working with Rust for over 8 years now
- SwishSwushPow@mastodon.social

Why are FFIs important?

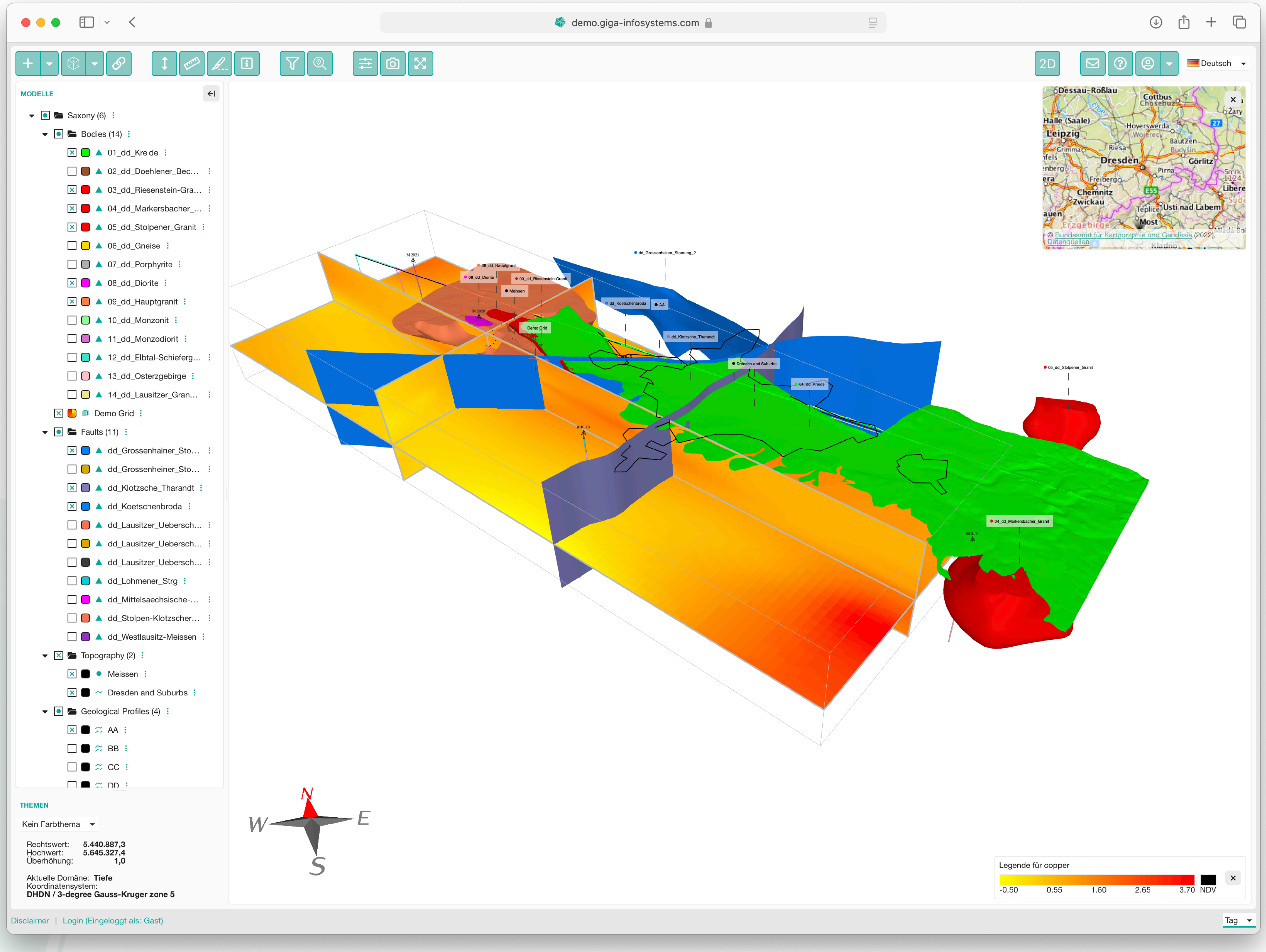
An important piece of the puzzle

- **F**oreign **F**unction **I**nterface
- Allows one language to call code written in another language
- Rust needs widespread adoption at companies of all sizes
- Existing code-bases will have to communicate with Rust code
- Ideal properties
  - Small amount of boilerplate required
  - (Almost) no negative performance impact
  - Ergonomic to use

And what experience do we bring to the table?

Fearless/foolish adoption of Rust from the start

- Our Rust journey started in 2016
- Fully replaced C++/SQL backend
- At GiGa infosystems we have:
  - DBMS for storing 3D geoscientific models, written in Rust
  - Desktop application written in C++
  - Web apps using Rust through WASM
  - Standalone Rust helper apps



Rechtswert: 5.440.887,3
Hochwert: 5.645.327,4
Überhöhung: 1,0

Aktuelle Domäne: **Tiefe**
Koordinatensystem:
DHDN / 3-degree Gauss-Kruger zone 5

Our journey so far

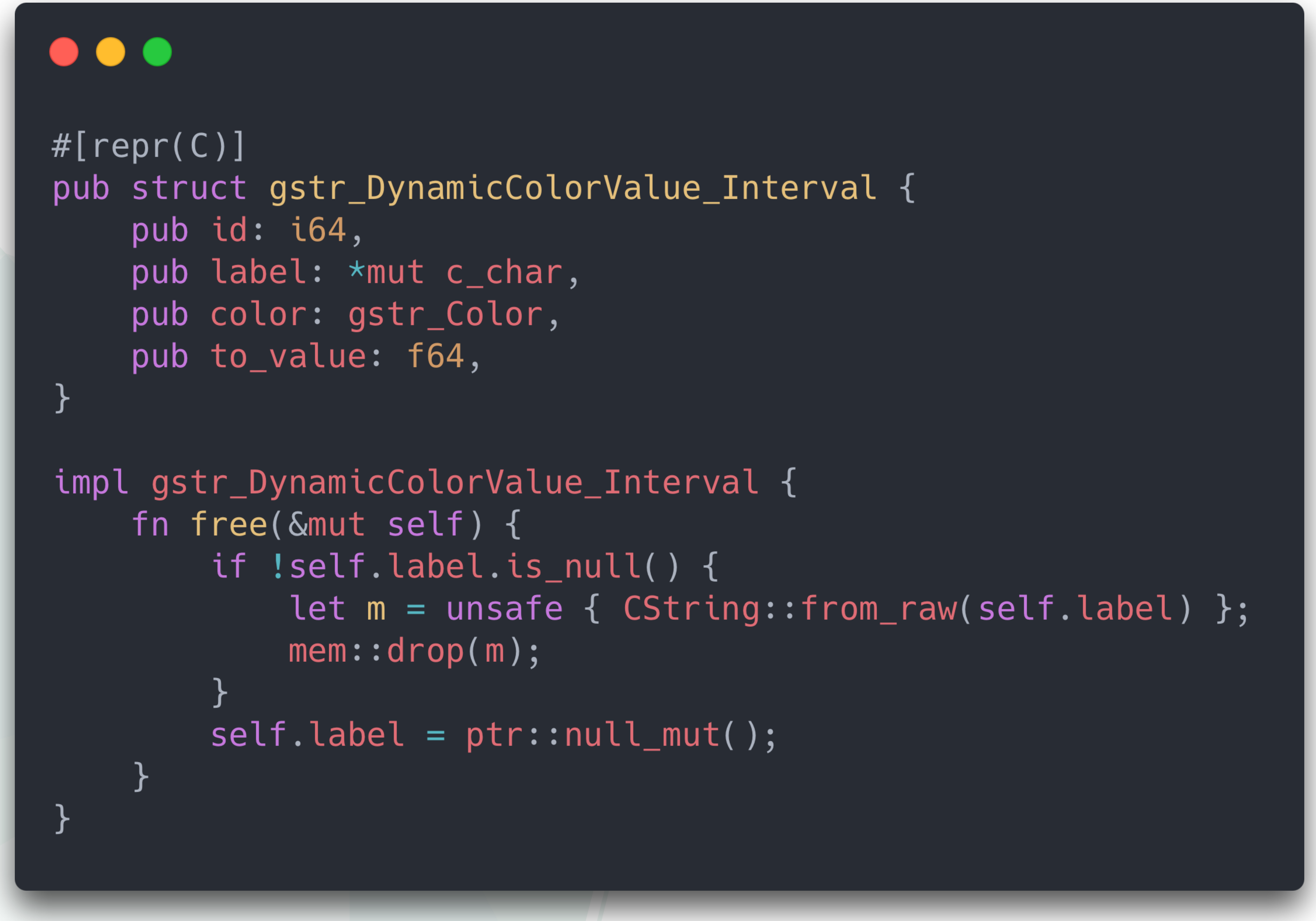
What have we used in the past?

- **cbindgen** (custom fork)
  - Generated not the best C++ code (String encoding, Windows-1252 <-> UTF-8)
  - Rust
    - String encoding
    - Type conversions
    - Manual deallocation
    - A lot of unsafe code (manual pointer handling)

- Extern „C"-functions were calling Rust code and handling all of the above

Our journey so far

Examples

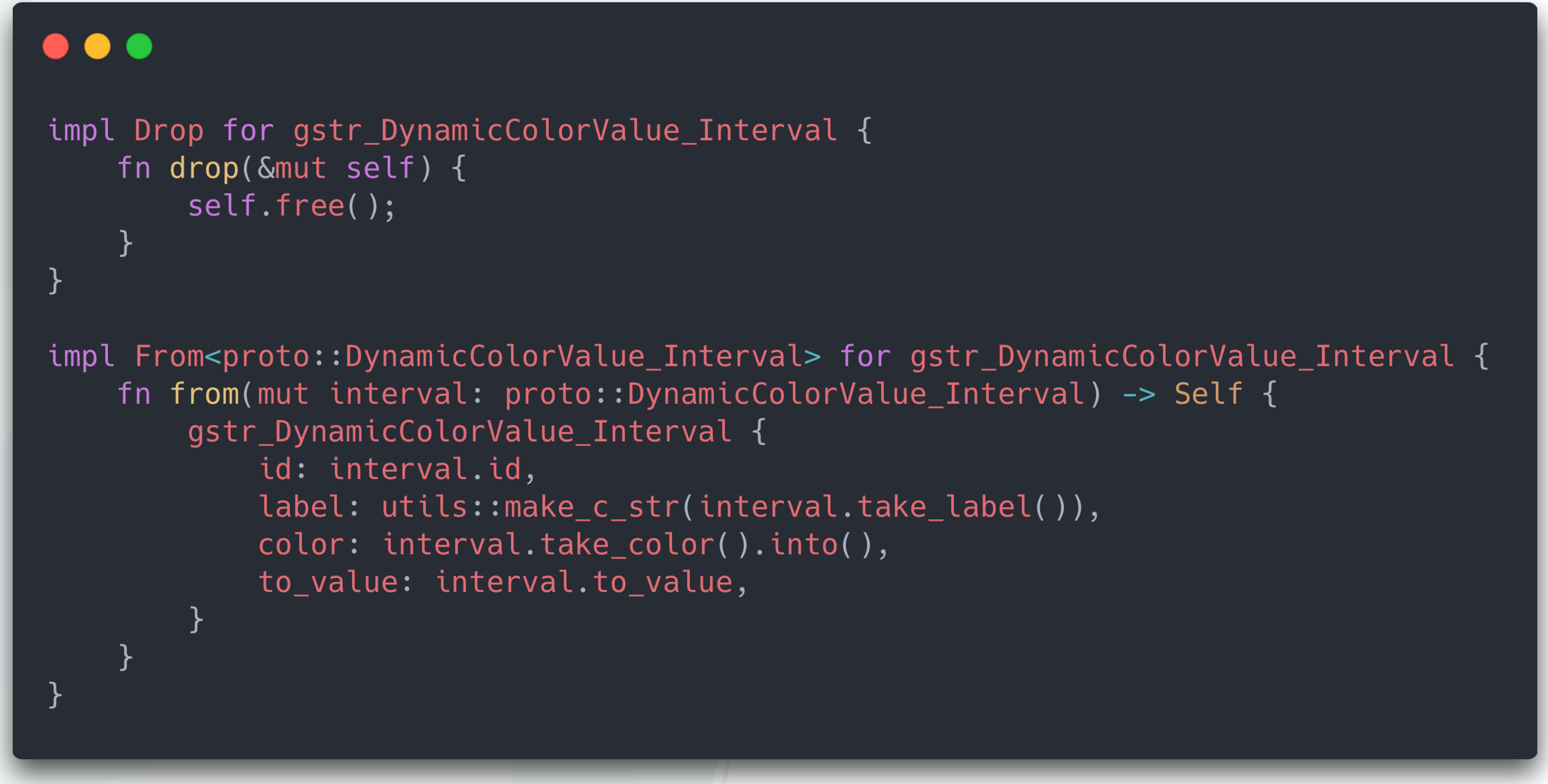


```
#[repr(C)]
pub struct gstr_DynamicColorValue_Interval {
    pub id: i64,
    pub label: *mut c_char,
    pub color: gstr_Color,
    pub to_value: f64,
}

impl gstr_DynamicColorValue_Interval {
    fn free(&mut self) {
        if !self.label.is_null() {
            let m = unsafe { CString::from_raw(self.label) };
            mem::drop(m);
        }
        self.label = ptr::null_mut();
    }
}
```

Our journey so far

Examples

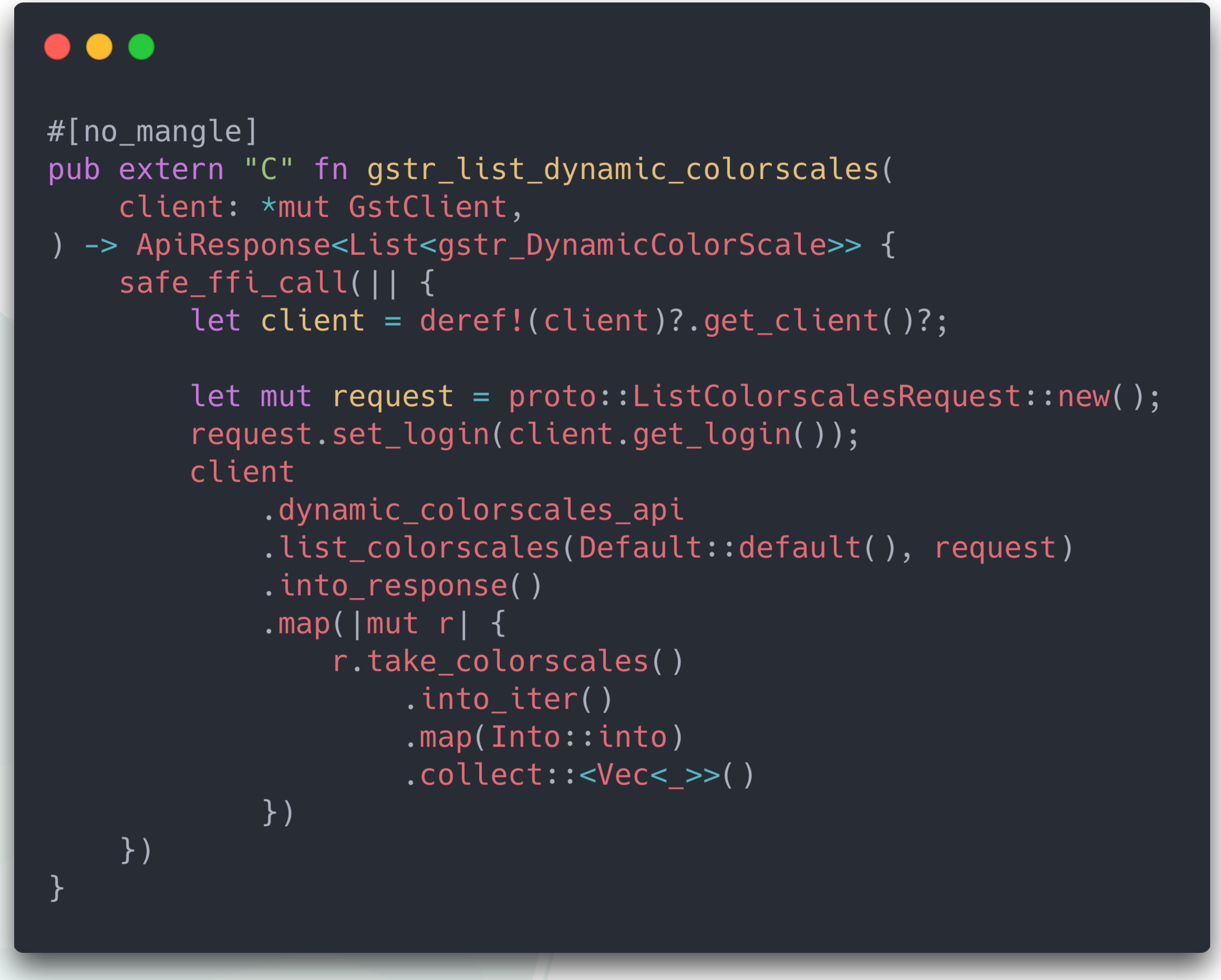


```rust
impl Drop for gstr_DynamicColorValue_Interval {
    fn drop(&mut self) {
        self.free();
    }
}

impl From<proto::DynamicColorValue_Interval> for gstr_DynamicColorValue_Interval {
    fn from(mut interval: proto::DynamicColorValue_Interval) -> Self {
        gstr_DynamicColorValue_Interval {
            id: interval.id,
            label: utils::make_c_str(interval.take_label()),
            color: interval.take_color().into(),
            to_value: interval.to_value,
        }
    }
}
```

Our journey so far

Examples



```
#[no_mangle]
pub extern "C" fn gstr_list_dynamic_colorscales(
    client: *mut GstClient,
) -> ApiResponse<List<gstr_DynamicColorScale>> {
    safe_ffi_call(|| {
        let client = deref!(client)?.get_client()?;

        let mut request = proto::ListColorscalesRequest::new();
        request.set_login(client.get_login());
        client
            .dynamic_colorscales_api
            .list_colorscales(Default::default(), request)
            .into_response()
            .map(|mut r| {
                r.take_colorscales()
                    .into_iter()
                    .map(Into::into)
                    .collect::<Vec<_>>()
            })
    })
}
```

Our journey so far

What other approaches have we considered?

- **Cxx**
  - Bad error propagation
  - Enum support not enough
- **safer_ffi**
  - Big adjustments for our C++ code required
  - We have to free things manually
- **Diplomat**
  - We would have to roll with a fork as well
  - Issues with String support in structs

Our journey so far

How has it evolved?

- Worked on error propagation and tracing
- Added Rust API „after the fact“ (extern „C“ functions were rewritten)
- Tried to handle remaining issues as well as possible
- Monitored new opportunities

- Slowly we have gained a clear picture what we would like/need

Our journey so far

How has it evolved?

- Worked on error propagation and tracing
- ~~Added Rust API „after the fact“ (extern „C“ functions were rewritten)~~
- Tried to handle remaining issues as well as possible
- Monitored new opportunities

- Slowly we have gained a clear picture what we would like/need

Generating ergonomic C++ APIs

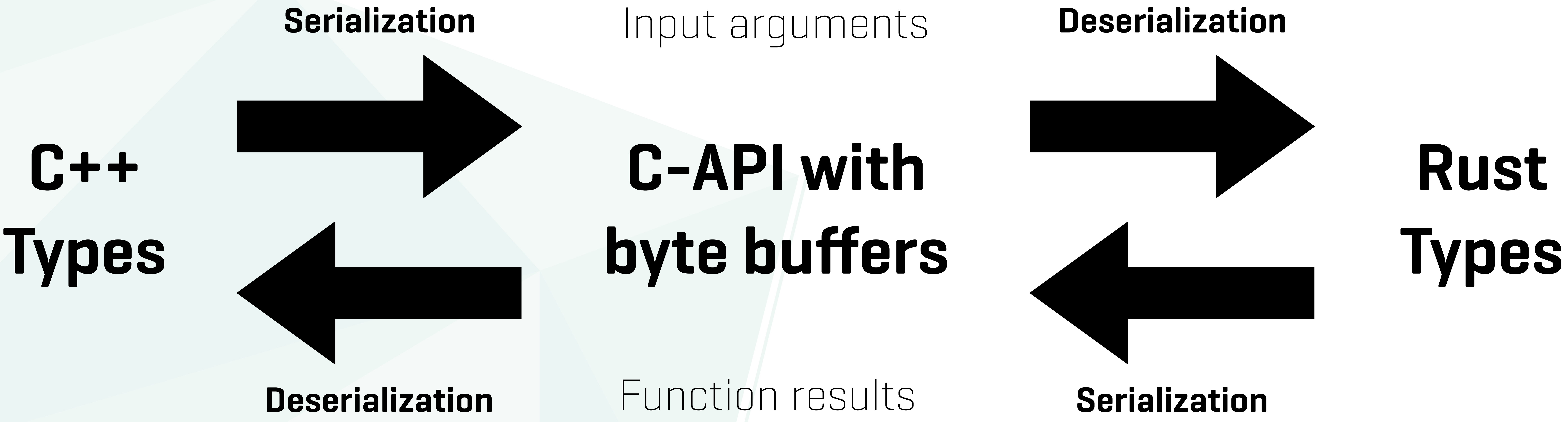
What are our goals?

- Cut down boilerplate
  - No manual conversion of types
  - No manual deallocation
  - No weird String handling
  - Contain unsafe code somewhere safe
- Make it nice to use from a C++ perspective
- Don’t lose too much performance

Generating ergonomic C++ APIs

Key idea: Replace all types with byte buffers

- Types create many headaches
- We replace (almost) all input/output types with byte buffers



Serialization

Input arguments

Deserialization

C++ Types

C-API with byte buffers

Rust Types

Deserialization

Function results

Serialization

Generating ergonomic C++ APIs

Our approach

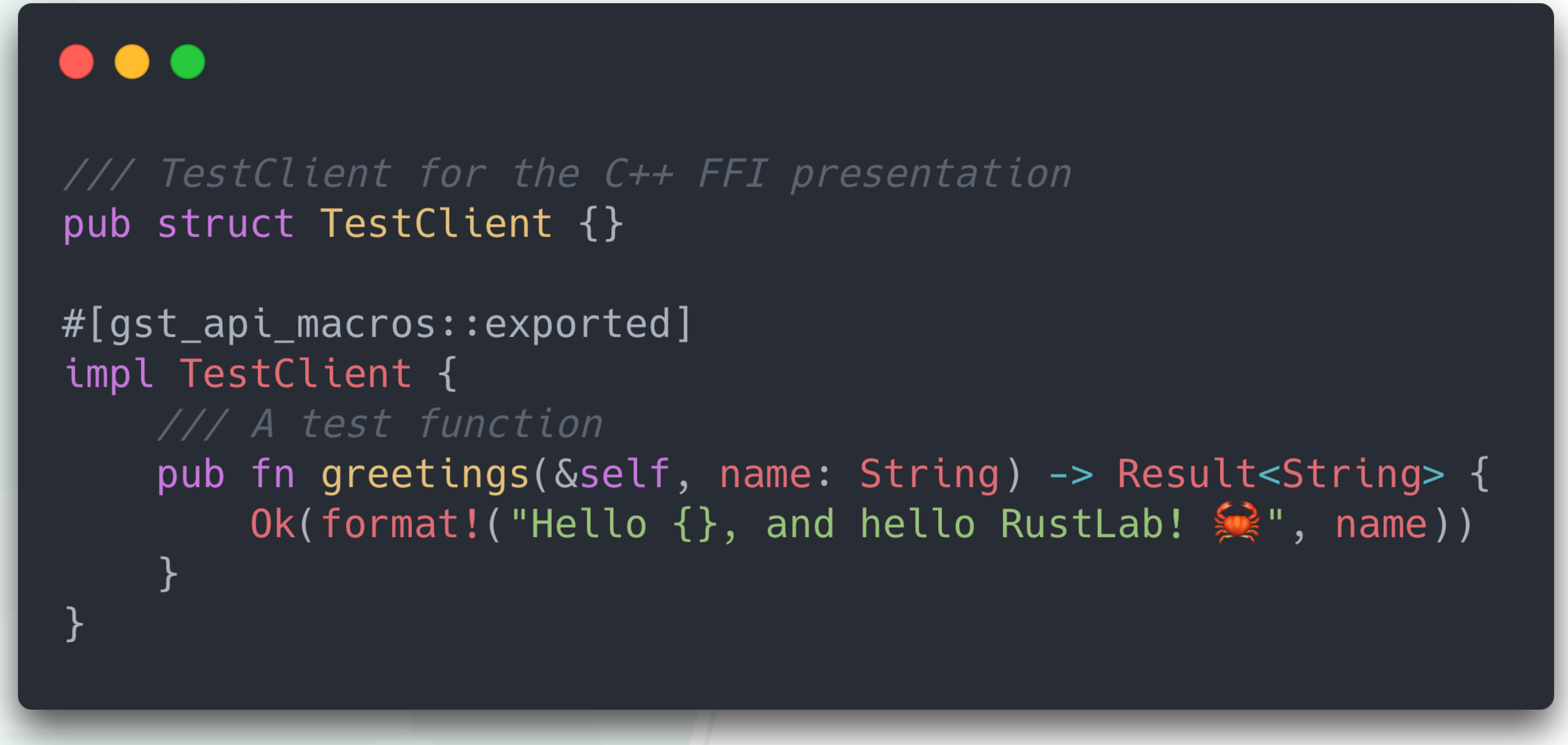
1. **Procedural macros** -> generate extern „C“ fns from Rust API
2. **Rustdoc + rustdoc-types** -> parse the generated code from above
3. **serde-reflection + serde-generate** -> use rustdoc-types input to generate C/C++ code

- **Serde/Bincode** to de-/serialize input/output into byte buffers to not worry about types

Generating ergonomic C++ APIs

Procedural macros -> extern „C“ fn

- Our extern „C“ functions dealt a lot with types and their conversion
- Using Serde/Bincode makes these functions very similar
- Procedural macro allows us to cut down boilerplate



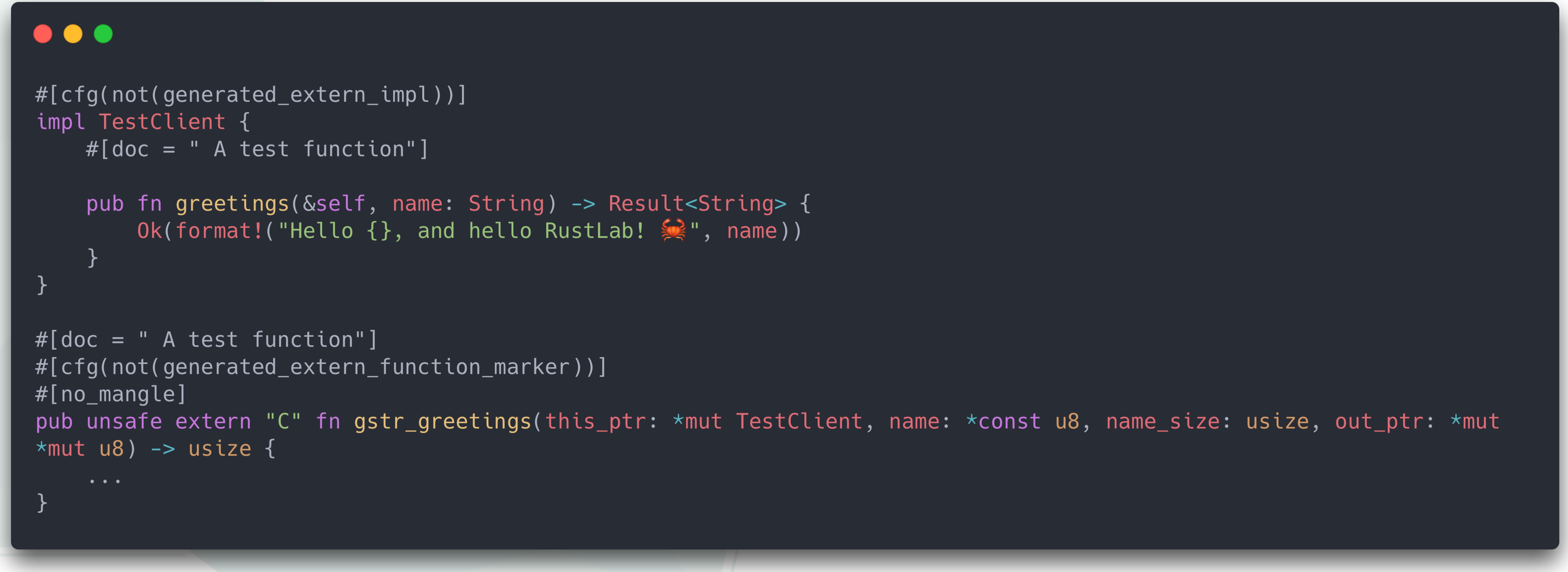
```rust
/// TestClient for the C++ FFI presentation
pub struct TestClient {}

#[gst_api_macros::exported]
impl TestClient {
    /// A test function
    pub fn greetings(&self, name: String) -> Result<String> {
        Ok(format!("Hello {}, and hello RustLab! 🦀", name))
    }
}
```

Generating ergonomic C++ APIs

Procedural macros -> extern „C“ fn

- Result of macro expansion (single step)
- Input and output types are turned into byte buffers



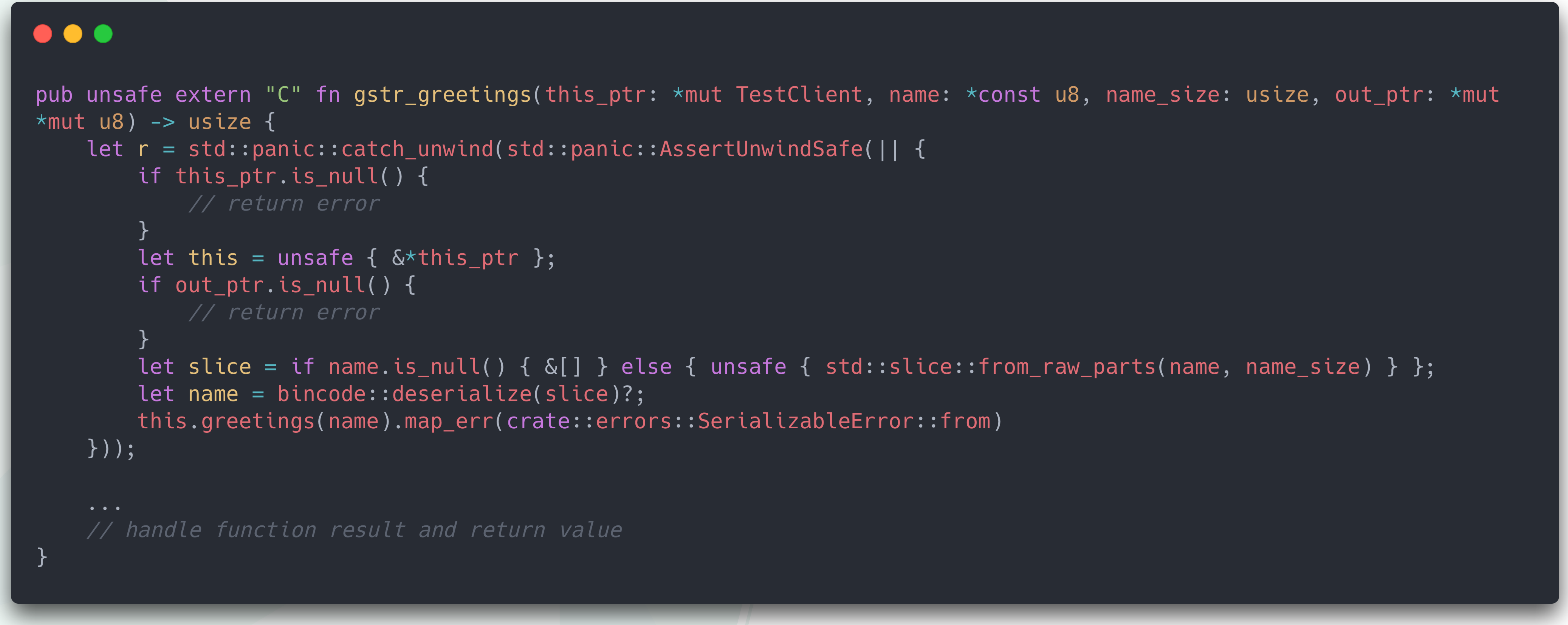
```
#[cfg(not(generated_extern_impl))]
impl TestClient {
    #[doc = " A test function"]

    pub fn greetings(&self, name: String) -> Result<String> {
        Ok(format!("Hello {}, and hello RustLab! 🦀", name))
    }
}

#[doc = " A test function"]
#[cfg(not(generated_extern_function_marker))]
#[no_mangle]
pub unsafe extern "C" fn gstr_greetings(this_ptr: *mut TestClient, name: *const u8, name_size: usize, out_ptr: *mut
*mut u8) -> usize {
    ...
}
```

Generating ergonomic C++ APIs

Procedural macros -> extern „C“ fn

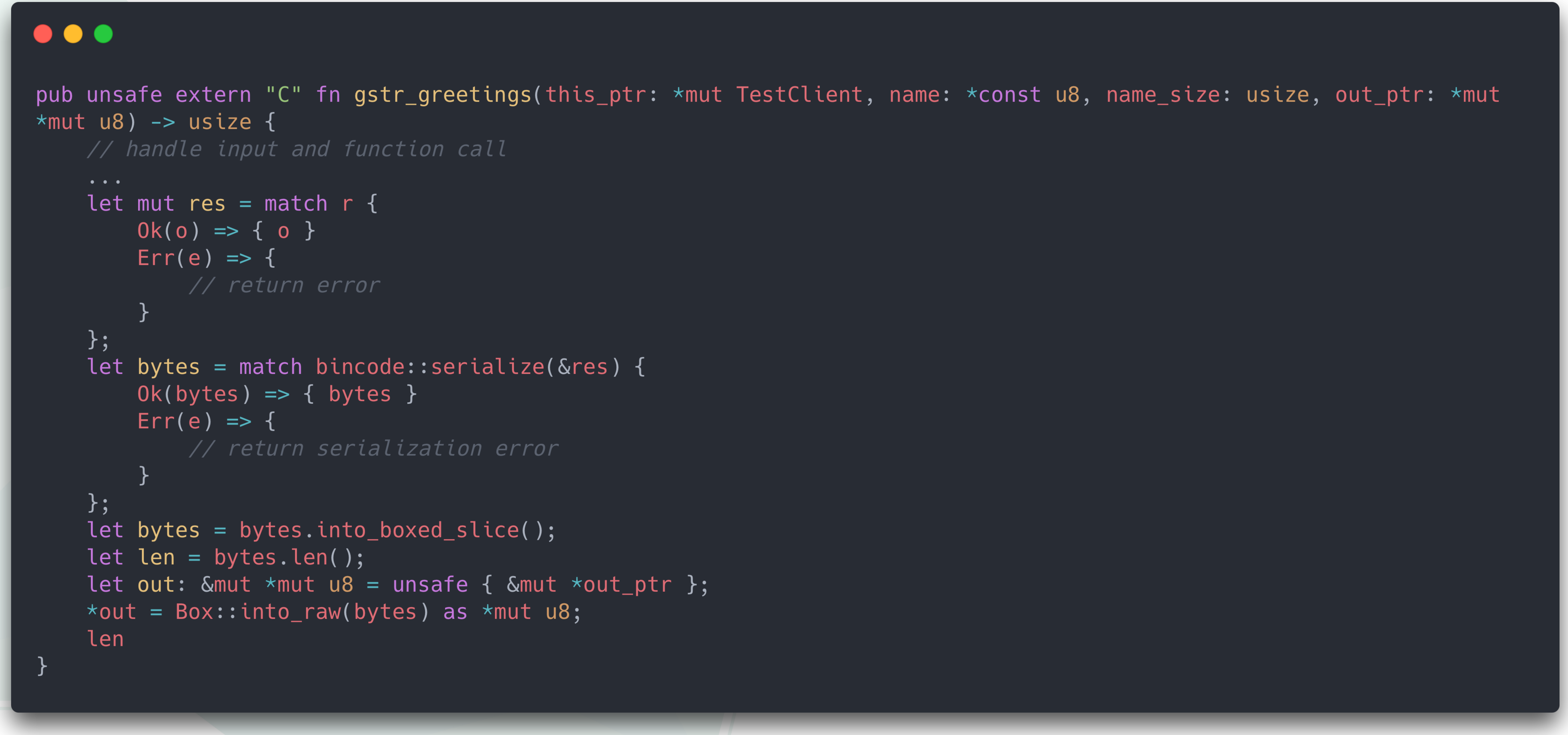


```
pub unsafe extern "C" fn gstr_greetings(this_ptr: *mut TestClient, name: *const u8, name_size: usize, out_ptr: *mut
*mut u8) -> usize {
    let r = std::panic::catch_unwind(std::panic::AssertUnwindSafe(|| {
        if this_ptr.is_null() {
            // return error
        }
        let this = unsafe { &*this_ptr };
        if out_ptr.is_null() {
            // return error
        }
        let slice = if name.is_null() { &[] } else { unsafe { std::slice::from_raw_parts(name, name_size) } };
        let name = bincode::deserialize(slice)?;
        this.greetings(name).map_err(crate::errors::SerializableError::from)
    }));

    ...
    // handle function result and return value
}
```

Generating ergonomic C++ APIs

Procedural macros -> extern „C“ fn



```rust
pub unsafe extern "C" fn gstr_greetings(this_ptr: *mut TestClient, name: *const u8, name_size: usize, out_ptr: *mut
*mut u8) -> usize {
    // handle input and function call
    ...
    let mut res = match r {
        Ok(o) => { o }
        Err(e) => {
            // return error
        }
    };
    let bytes = match bincode::serialize(&res) {
        Ok(bytes) => { bytes }
        Err(e) => {
            // return serialization error
        }
    };
    let bytes = bytes.into_boxed_slice();
    let len = bytes.len();
    let out: &mut *mut u8 = unsafe { &mut *out_ptr };
    *out = Box::into_raw(bytes) as *mut u8;
    len
}
```

No more proc macros!

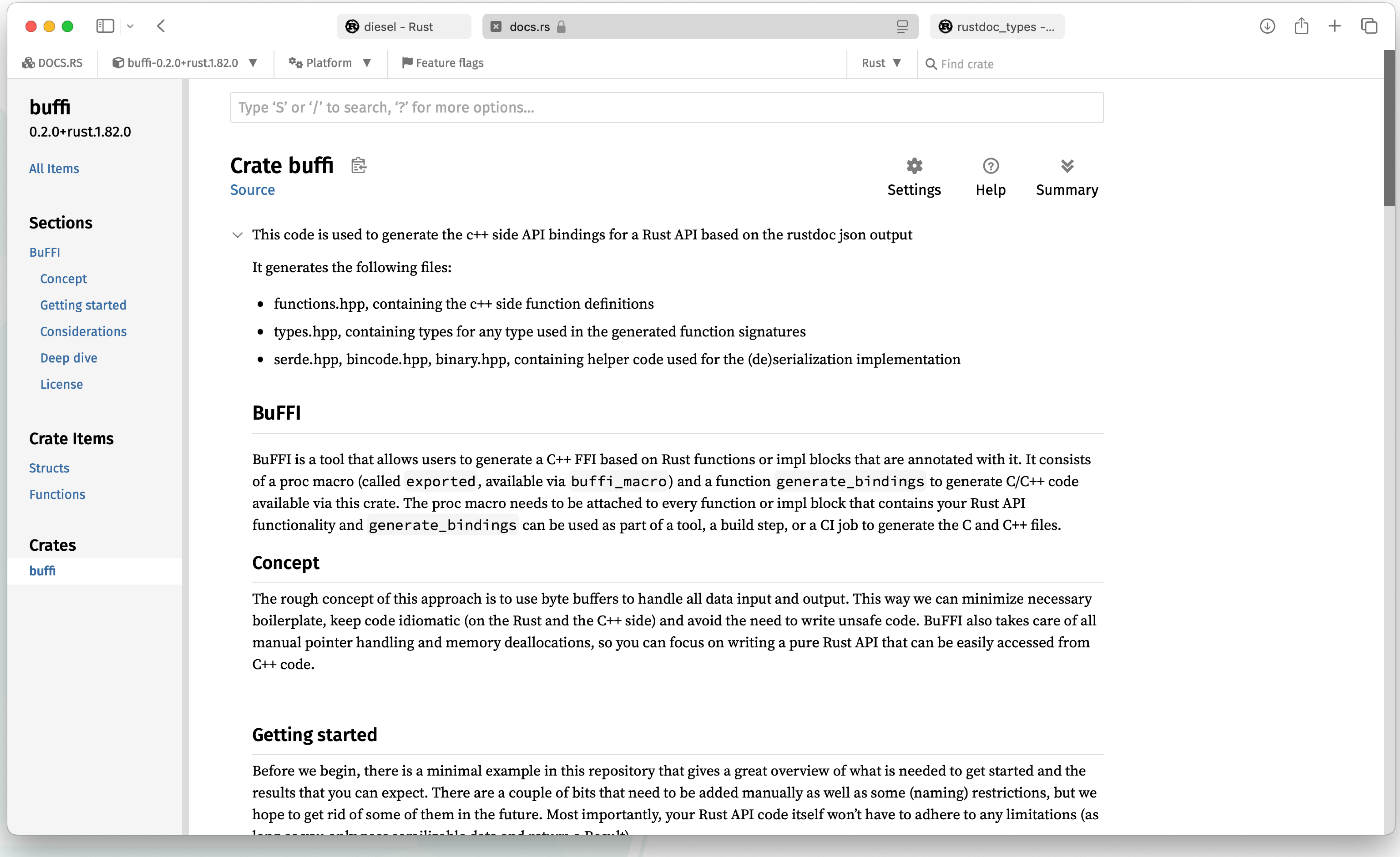
Generating ergonomic C++ APIs

Rustdoc, JSON, and rustdoc-types

- Now we have to generate the C/C++ side of things
- We need an understanding of
  - The functions we have added
  - The types we used
  - Whether a function is part of an „impl“ block or not

Generating ergonomic C++ APIs

Rustdoc, JSON, and rustdoc-types



Crate buffi

Source

This code is used to generate the c++ side API bindings for a Rust API based on the rustdoc json output

It generates the following files:

- functions.hpp, containing the c++ side function definitions
- types.hpp, containing types for any type used in the generated function signatures
- serde.hpp, bincode.hpp, binary.hpp, containing helper code used for the (de)serialization implementation

BuFFI

BuFFI is a tool that allows users to generate a C++ FFI based on Rust functions or impl blocks that are annotated with it. It consists of a proc macro (called `exported`, available via `buffi_macro`) and a function `generate_bindings` to generate C/C++ code available via this crate. The proc macro needs to be attached to every function or impl block that contains your Rust API functionality and `generate_bindings` can be used as part of a tool, a build step, or a CI job to generate the C and C++ files.

Concept

The rough concept of this approach is to use byte buffers to handle all data input and output. This way we can minimize necessary boilerplate, keep code idiomatic (on the Rust and the C++ side) and avoid the need to write unsafe code. BuFFI also takes care of all manual pointer handling and memory deallocations, so you can focus on writing a pure Rust API that can be easily accessed from C++ code.

Getting started

Before we begin, there is a minimal example in this repository that gives a great overview of what is needed to get started and the results that you can expect. There are a couple of bits that need to be added manually as well as some (naming) restrictions, but we hope to get rid of some of them in the future. Most importantly, your Rust API code itself won't have to adhere to any limitations (as long as you only pass serilizable data and return a Result)

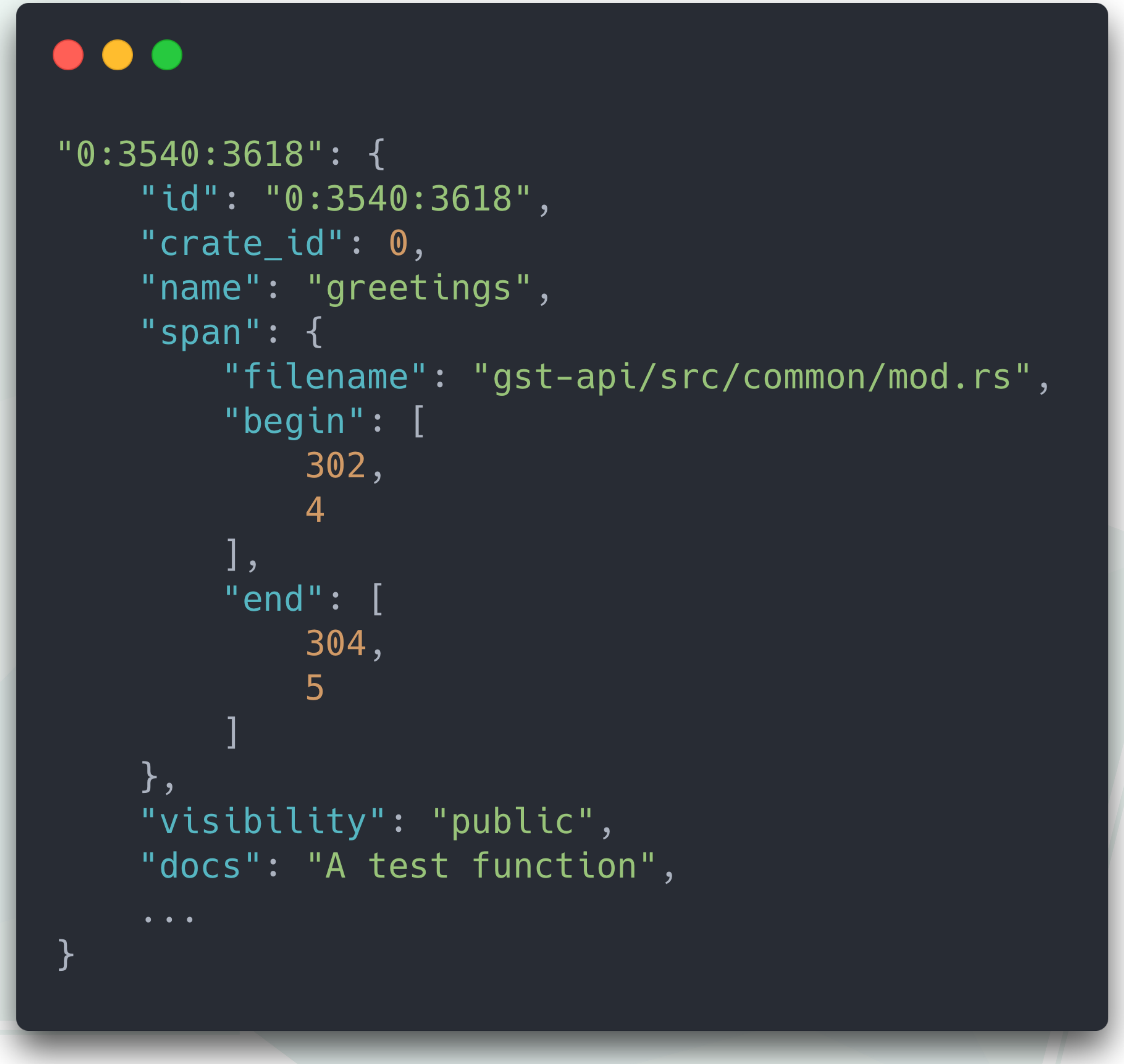
Generating ergonomic C++ APIs

Rustdoc, JSON, and rustdoc-types

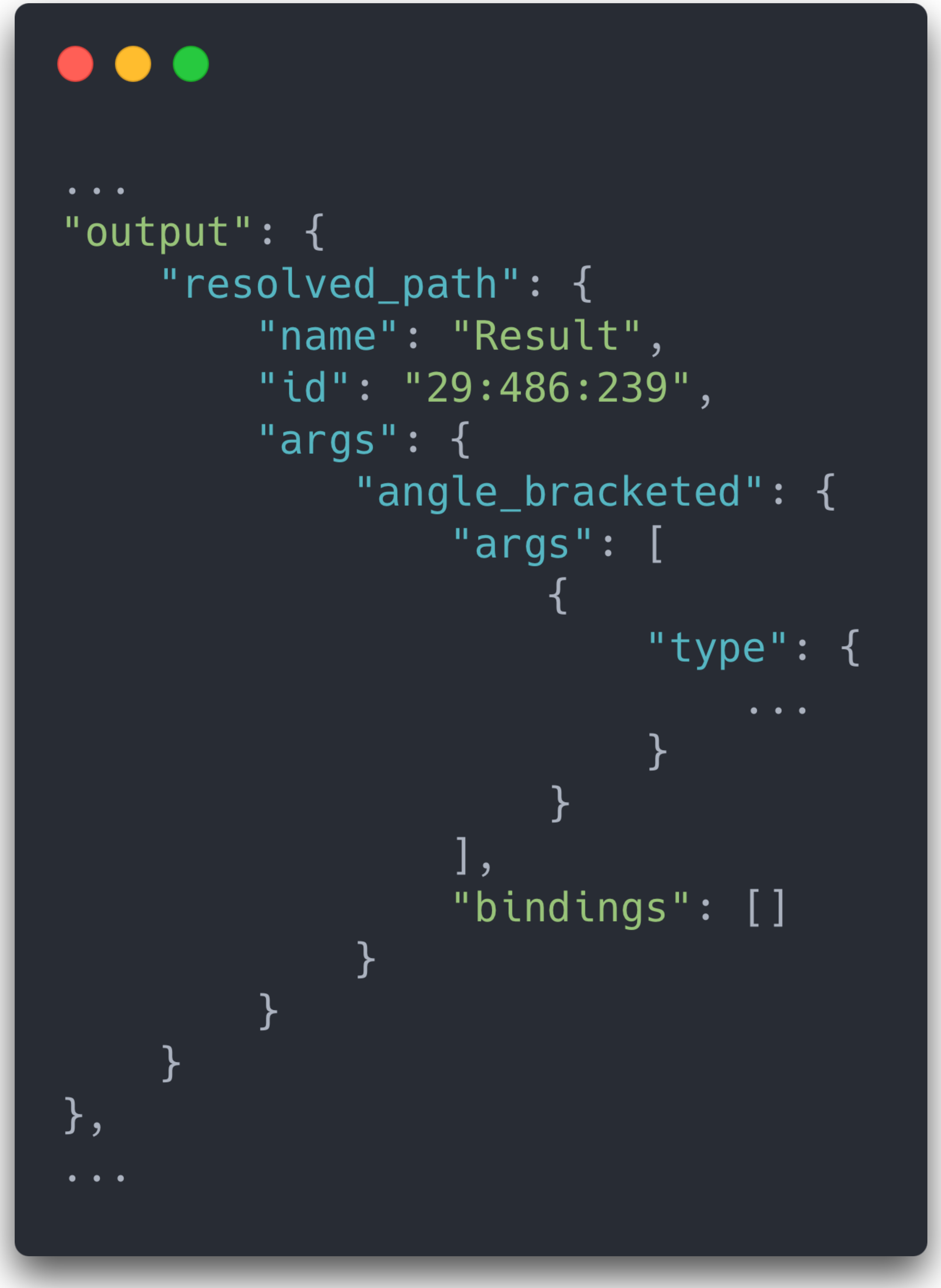
- Usually **Rustdoc** generates HTML output (as seen on docs.rs)
- Rustdoc also has unstable JSON output format
- **rustdoc-types** can read this (with serde)
- We generate this JSON for our own crates and relevant external dependencies
- ???
- Profit

Generating ergonomic C++ APIs

Rustdoc, JSON, and rustdoc-types



```
"0:3540:3618": {
    "id": "0:3540:3618",
    "crate_id": 0,
    "name": "greetings",
    "span": {
        "filename": "gst-api/src/common/mod.rs",
        "begin": [
            302,
            4
        ],
        "end": [
            304,
            5
        ]
    },
    "visibility": "public",
    "docs": "A test function",
    ...
}
```



```
...
"output": {
    "resolved_path": {
        "name": "Result",
        "id": "29:486:239",
        "args": {
            "angle_bracketed": {
                "args": [
                    {
                        "type": {
                            ...
                        }
                    }
                ],
                "bindings": []
            }
        }
    }
},
...
```

Generating ergonomic C++ APIs

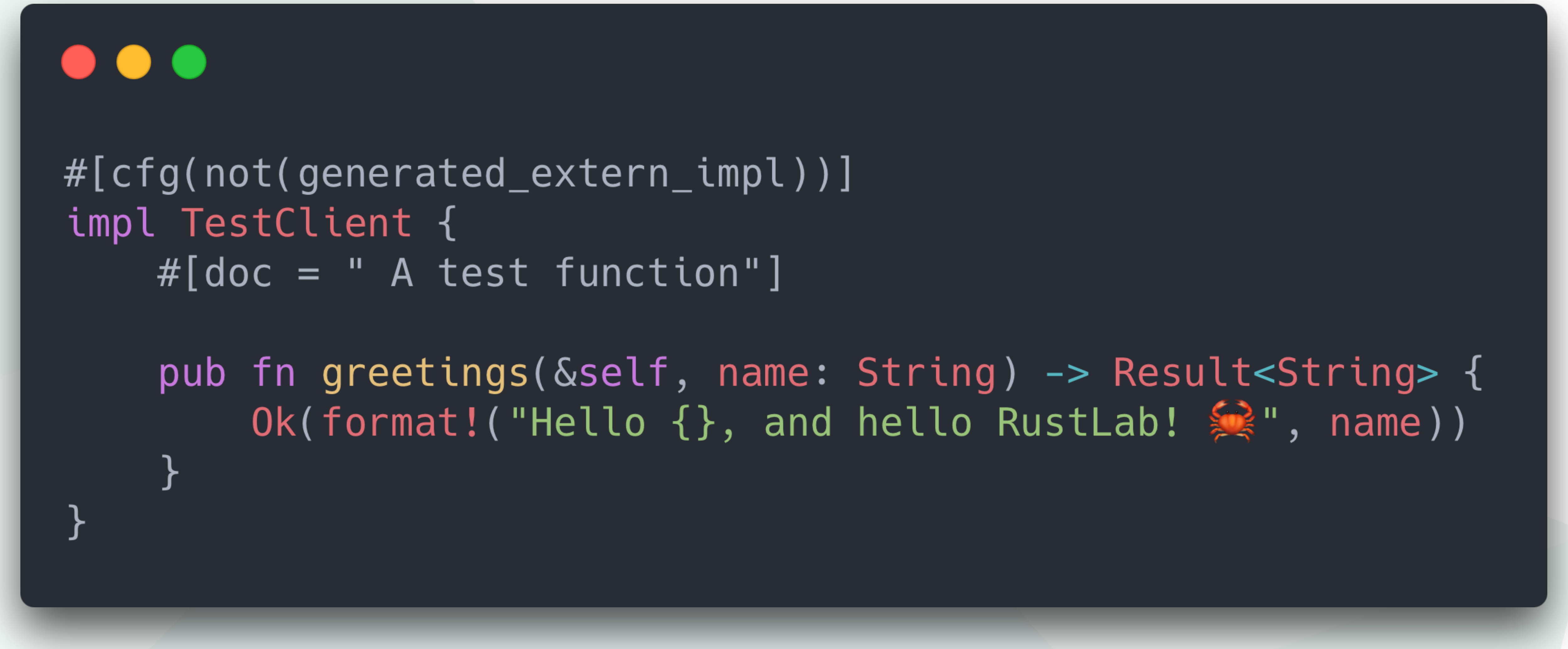
Rustdoc, JSON, and rustdoc-types

- Now we only have to find the relevant functions and types ...

Generating ergonomic C++ APIs

Rustdoc, JSON, and rustdoc-types

- Now we only have to find the relevant functions and types ...



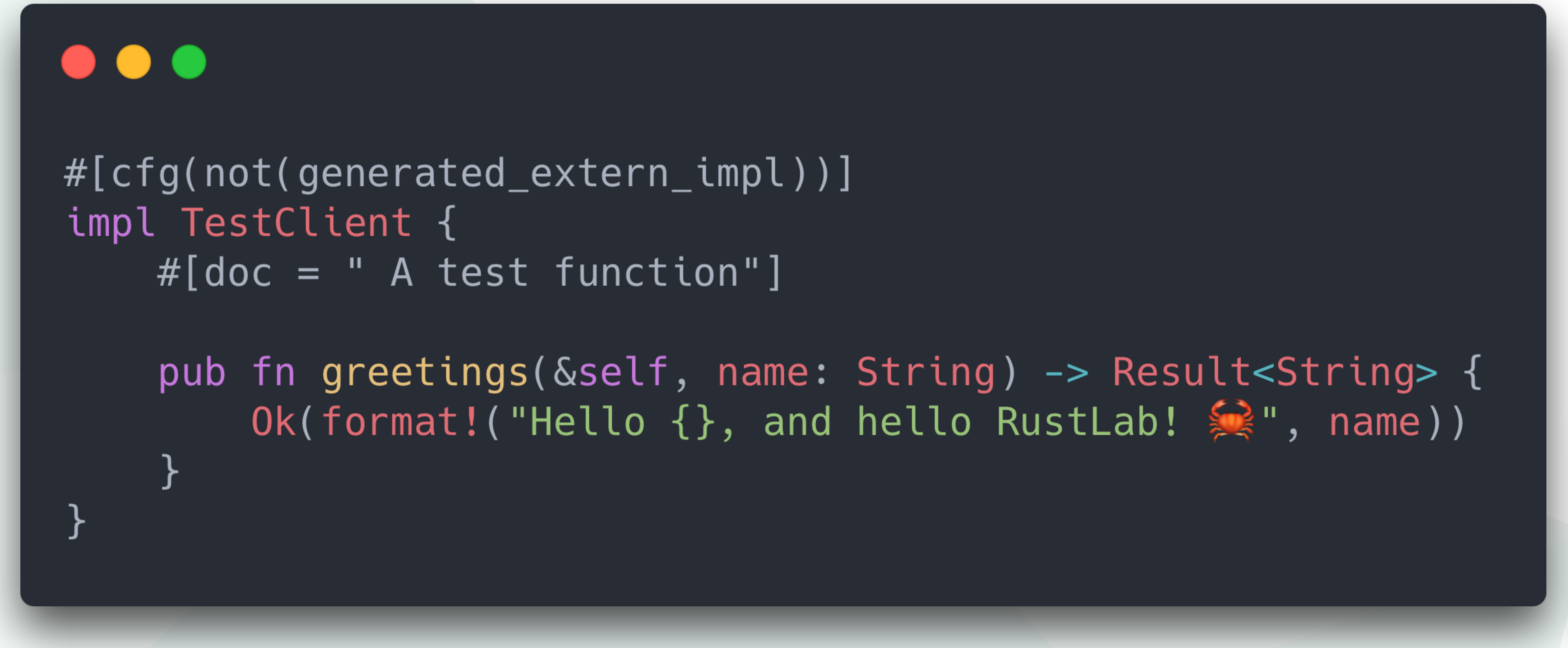
```rust
#[cfg(not(generated_extern_impl))]
impl TestClient {
    #[doc = " A test function"]

    pub fn greetings(&self, name: String) -> Result<String> {
        Ok(format!("Hello {}, and hello RustLab! 🦀", name))
    }
}
```

Generating ergonomic C++ APIs

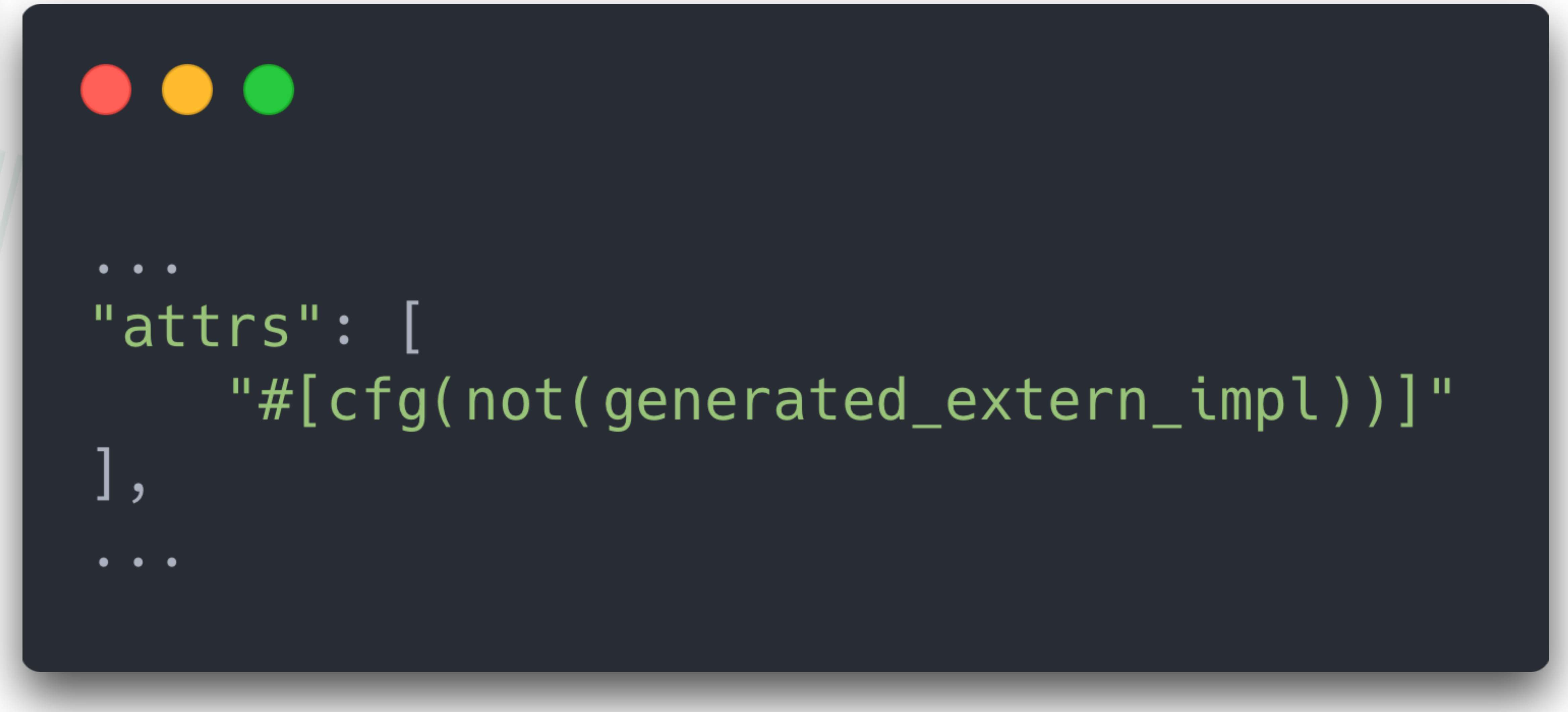
Rustdoc, JSON, and rustdoc-types

- Now we only have to find the relevant functions and types ...



```rust
#[cfg(not(generated_extern_impl))]
impl TestClient {
    #[doc = " A test function"]

    pub fn greetings(&self, name: String) -> Result<String> {
        Ok(format!("Hello {}, and hello RustLab! 🦀", name))
    }
}
```



```
...
"attrs": [
     "#[cfg(not(generated_extern_impl))]"
],
...
```

Generating ergonomic C++ APIs

rustdoc-types -> serde-reflect/serde-generate

- Challenge is to work through the tree and find the right types
- Include external dependencies if necessary

- For the **types**
  - Convert them into **serde-reflection** types
  - Put the result into **serde-generate**
- For the **functions**
  - Not so „easy“, but only dealing with byte buffers helps a lot
  - Put together the C declarations and C++ functions manually

Generating ergonomic C++ APIs

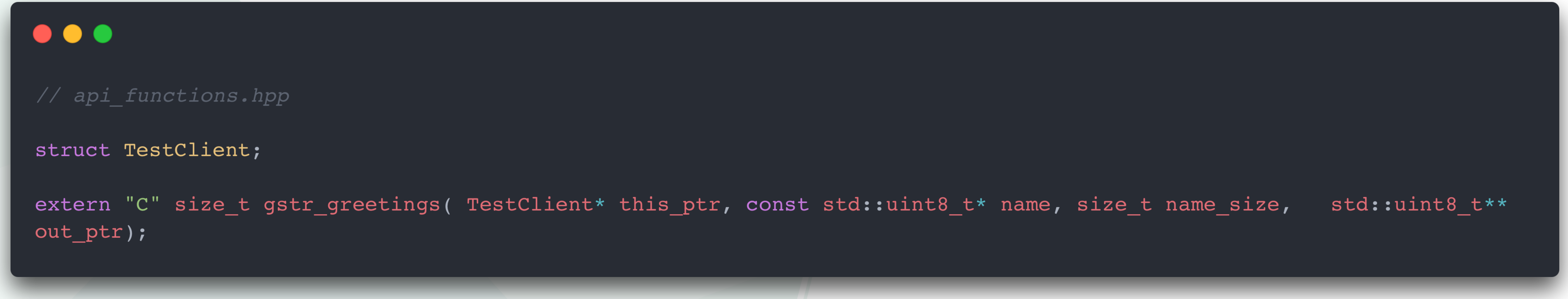
rustdoc-types -> serde-reflect/serde-generate

- In summary we write these files
  - **binary.hpp** and **bincode.hpp** [for Bincode]
  - **serde.hpp** [for Serde]

  - **api_functions.hpp** [C-API with byte buffers]
  - **types.hpp** [includes all the types]
  - **testclient.hpp** [C++ functions with actual input/output types and de-/serialization]
  - **free_standing_functions.hpp** [C++ functions not from an „impl“ block]

Generating ergonomic C++ APIs

api_functions.hpp



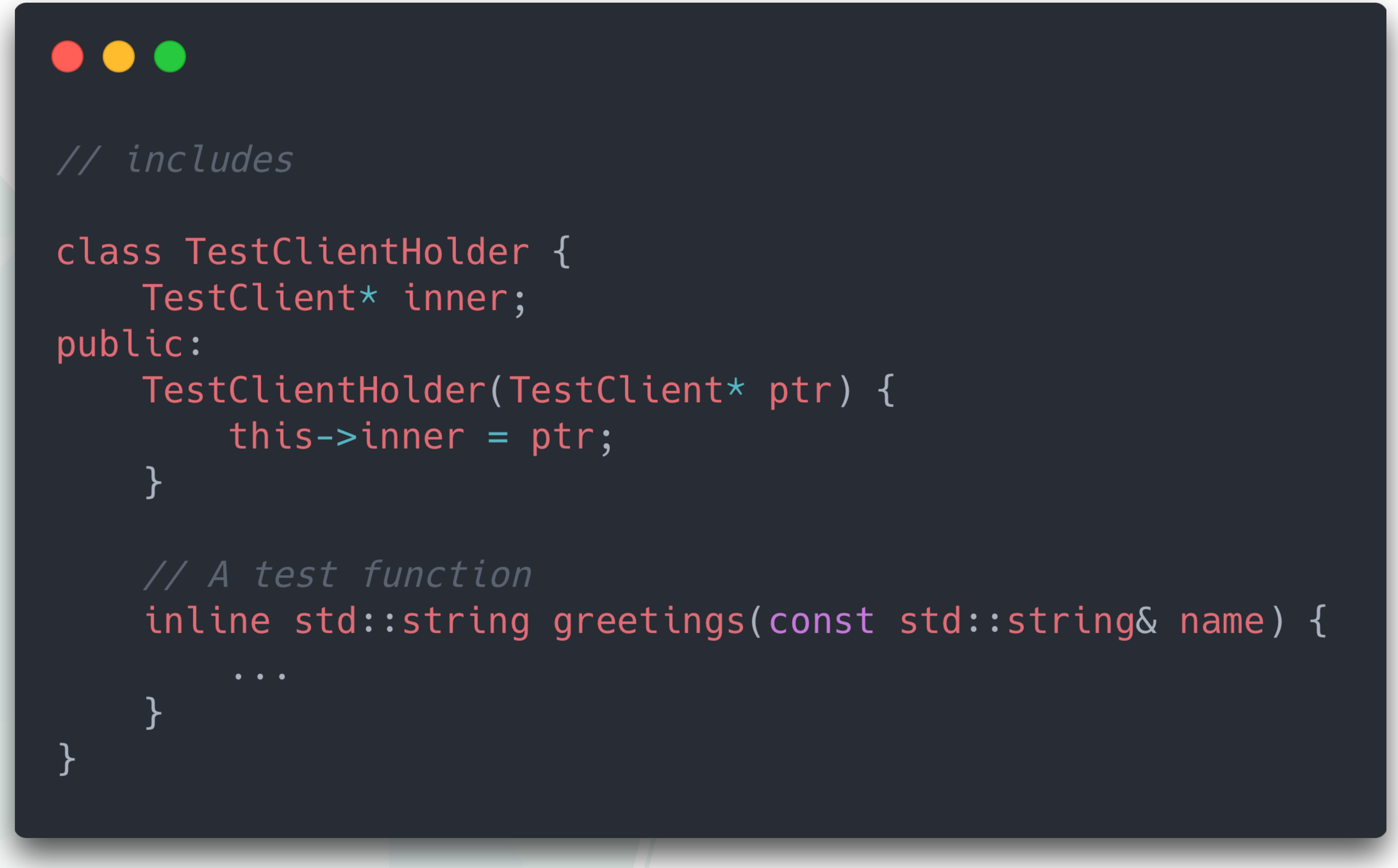
```
// api_functions.hpp

struct TestClient;

extern "C" size_t gstr_greetings( TestClient* this_ptr, const std::uint8_t* name, size_t name_size,   std::uint8_t**
out_ptr);
```

Generating ergonomic C++ APIs

testclient.hpp

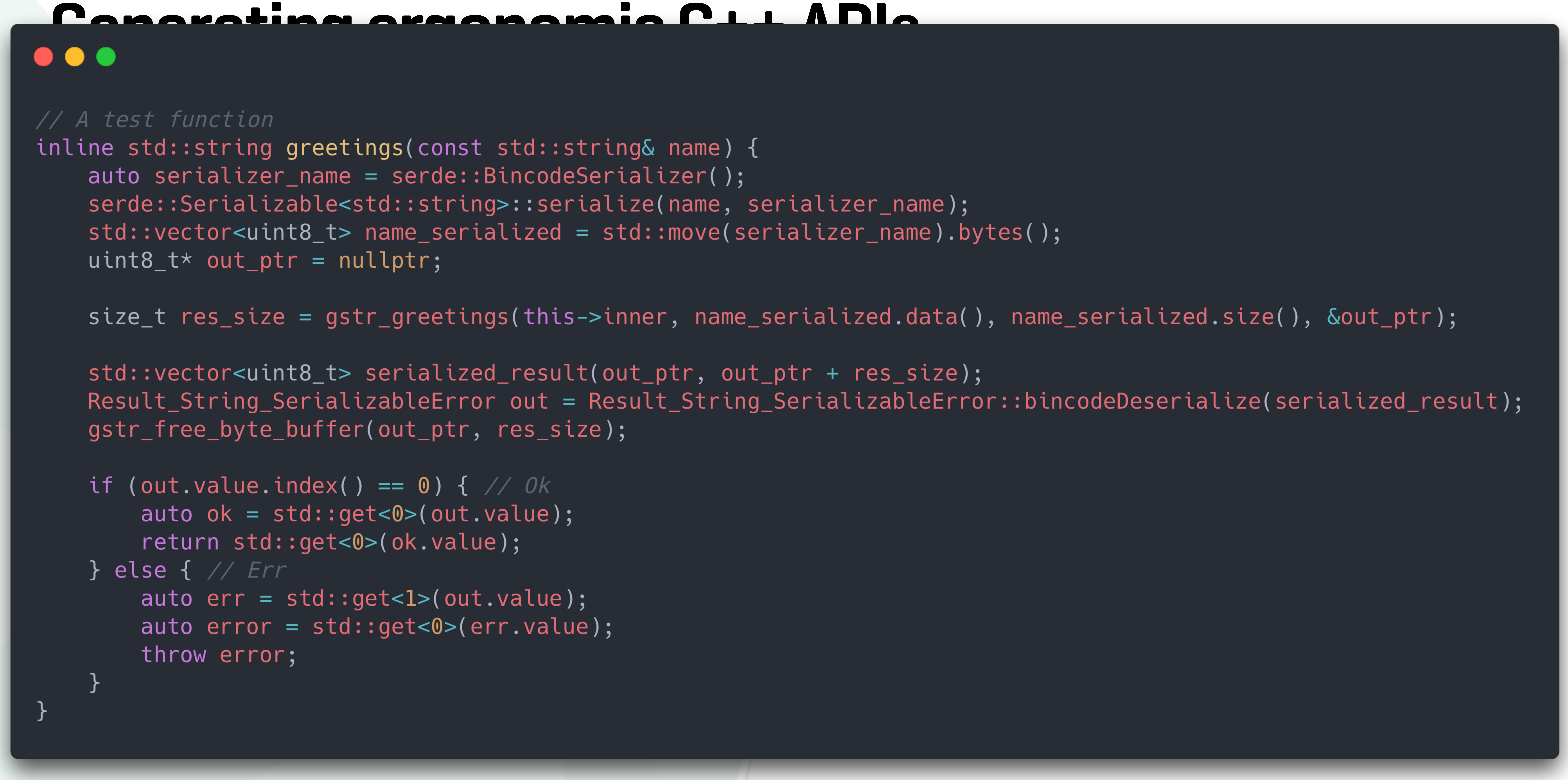


```
// includes

class TestClientHolder {
    TestClient* inner;
public:
    TestClientHolder(TestClient* ptr) {
        this->inner = ptr;
    }

    // A test function
    inline std::string greetings(const std::string& name) {
        ...
    }
}
```

Generating ergonomic C++ APIs



```cpp
// A test function
inline std::string greetings(const std::string& name) {
    auto serializer_name = serde::BincodeSerializer();
    serde::Serializable<std::string>::serialize(name, serializer_name);
    std::vector<uint8_t> name_serialized = std::move(serializer_name).bytes();
    uint8_t* out_ptr = nullptr;

    size_t res_size = gstr_greetings(this->inner, name_serialized.data(), name_serialized.size(), &out_ptr);

    std::vector<uint8_t> serialized_result(out_ptr, out_ptr + res_size);
    Result_String_SerializableError out = Result_String_SerializableError::bincodeDeserialize(serialized_result);
    gstr_free_byte_buffer(out_ptr, res_size);

    if (out.value.index() == 0) { // Ok
        auto ok = std::get<0>(out.value);
        return std::get<0>(ok.value);
    } else { // Err
        auto err = std::get<1>(out.value);
        auto error = std::get<0>(err.value);
        throw error;
    }
}
```

Generating ergonomic C++ APIs

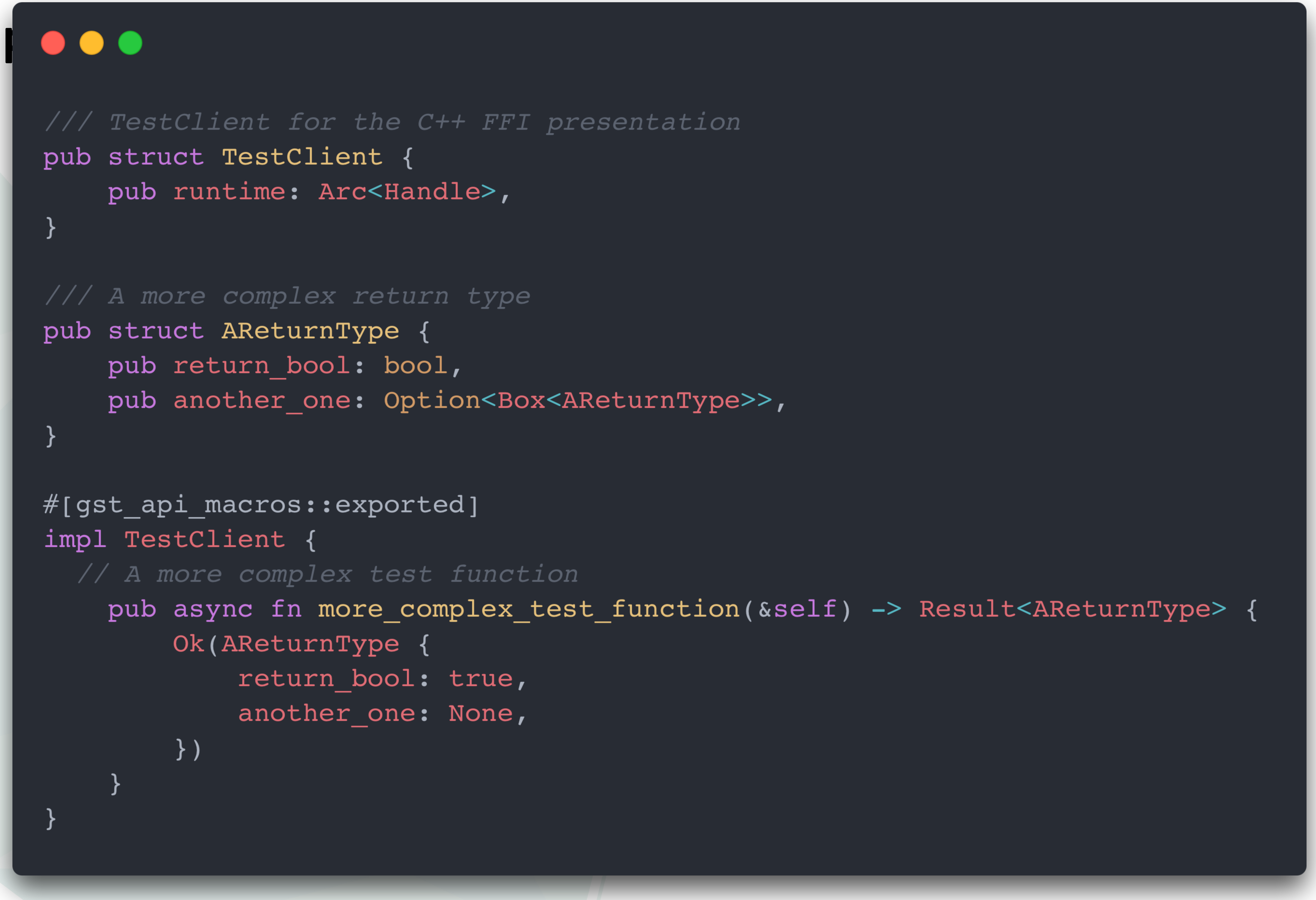
Custom error and result types

- Use custom types for Result and Errors
- Result -> **Result_String_SerializableError**
  - Holds **Ok** and **Err** variants, they implement the same De-/Serialization as other types
- Error -> **SerializableError**
  - Holds additional info such as tracing
  - Error type should be replaceable in the future

More complex example

Generating ergonomic C++ APIs

More comp



```rust
/// TestClient for the C++ FFI presentation
pub struct TestClient {
    pub runtime: Arc<Handle>,
}

/// A more complex return type
pub struct AReturnType {
    pub return_bool: bool,
    pub another_one: Option<Box<AReturnType>>,
}

#[gst_api_macros::exported]
impl TestClient {
  // A more complex test function
    pub async fn more_complex_test_function(&self) -> Result<AReturnType> {
        Ok(AReturnType {
            return_bool: true,
            another_one: None,
        })
    }
}
```

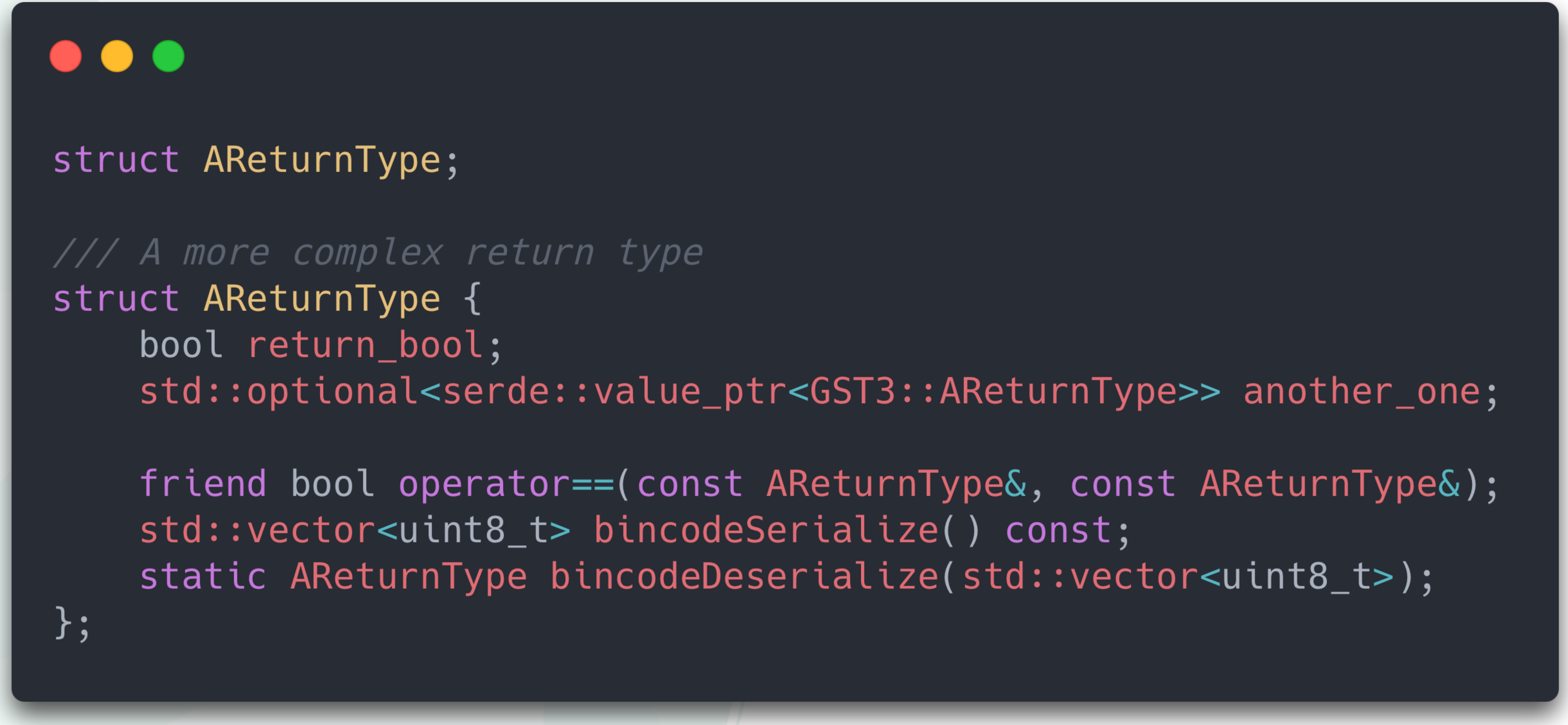
Generating ergonomic C++ APIs

More complex example (proc macro expansion)

```
pub unsafe extern "C" fn gstr_more_complex_test_function(this_ptr: *mut TestClient, out_ptr: *mut *mut u8) -> usize
{
    let r = std::panic::catch_unwind(std::panic::AssertUnwindSafe(|| {
        if this_ptr.is_null() {
            // return error
        }
        let this = unsafe { &*this_ptr };
        if out_ptr.is_null() {
            // return error
        }
        let runtime = std::sync::Arc::clone(&this.runtime);
        let fut = async move { this.more_complex_test_function()
          .await
          .map_err(crate::errors::SerializableError::from) };
        runtime.block_on(fut)
    }));
    ...
    // handle function result and return value
}
```

Generating ergonomic C++ APIs

types.hpp - Struct



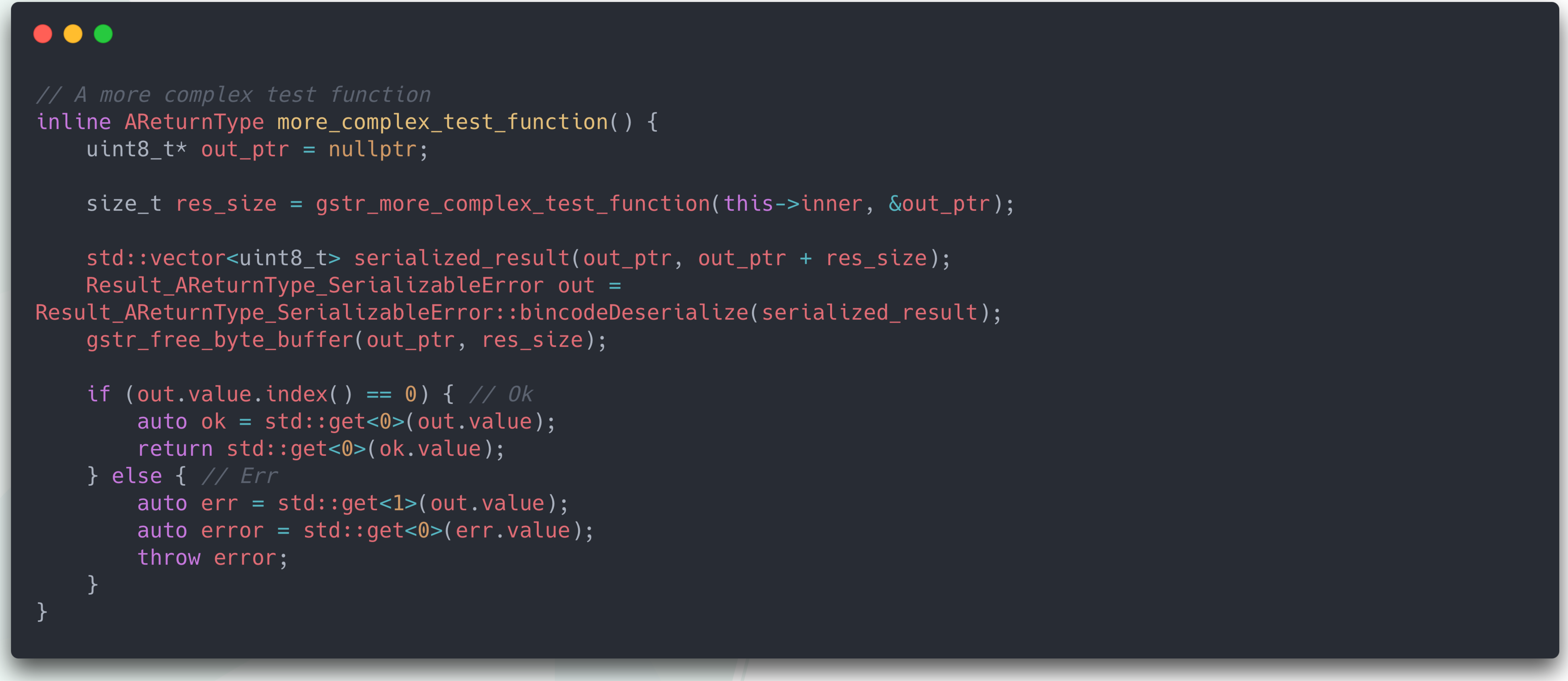
```
struct AReturnType;

/// A more complex return type
struct AReturnType {
    bool return_bool;
    std::optional<serde::value_ptr<GST3::AReturnType>> another_one;

    friend bool operator==(const AReturnType&, const AReturnType&);
    std::vector<uint8_t> bincodeSerialize() const;
    static AReturnType bincodeDeserialize(std::vector<uint8_t>);
};
```

Generating ergonomic C++ APIs

testclient.hpp



```
// A more complex test function
inline AReturnType more_complex_test_function() {
    uint8_t* out_ptr = nullptr;

    size_t res_size = gstr_more_complex_test_function(this->inner, &out_ptr);

    std::vector<uint8_t> serialized_result(out_ptr, out_ptr + res_size);
    Result_AReturnType_SerializableError out =
Result_AReturnType_SerializableError::bincodeDeserialize(serialized_result);
    gstr_free_byte_buffer(out_ptr, res_size);

    if (out.value.index() == 0) { // Ok
        auto ok = std::get<0>(out.value);
        return std::get<0>(ok.value);
    } else { // Err
        auto err = std::get<1>(out.value);
        auto error = std::get<0>(err.value);
        throw error;
    }
}
```

Generating ergonomic C++ APIs

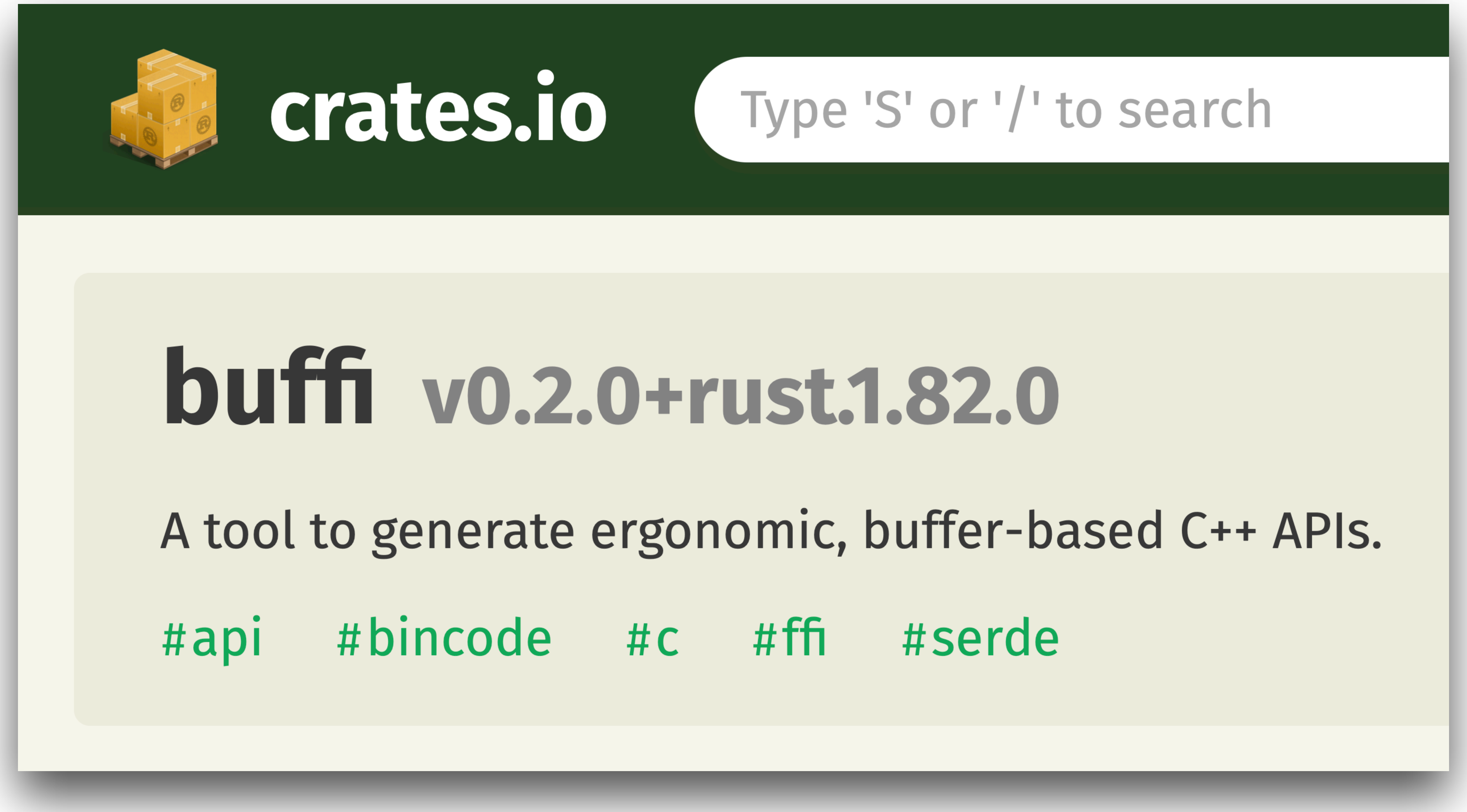
Up- and downsides

- **Upsides**
  - No explicit type conversions
  - No exposed unsafe code
  - No pointer handling
  - No explicit deallocations
- **Downsides**
  - Lose a bit of performance
  - No directly „useable“ C-API

You can give this a try today!

BuFFI is now available on crates.io

- Just released this week
- „**buffi**“ and „**buffi_macro**“



crates.io

Type 'S' or '/' to search

buffi v0.2.0+rust.1.82.0

A tool to generate ergonomic, buffer-based C++ APIs.

#api #bincode #c #ffi #serde

- Rustdoc type resolving has grown organically
- Work together to make this more universally applicable
- Recommended for production?
  - Stabilization of Rustdoc JSON output would be huge!
  - Otherwise **RUSTC_BOOTSTRAP** or a (specific) **nightly toolchain** has to be used

Follow along!

And don’t miss anything

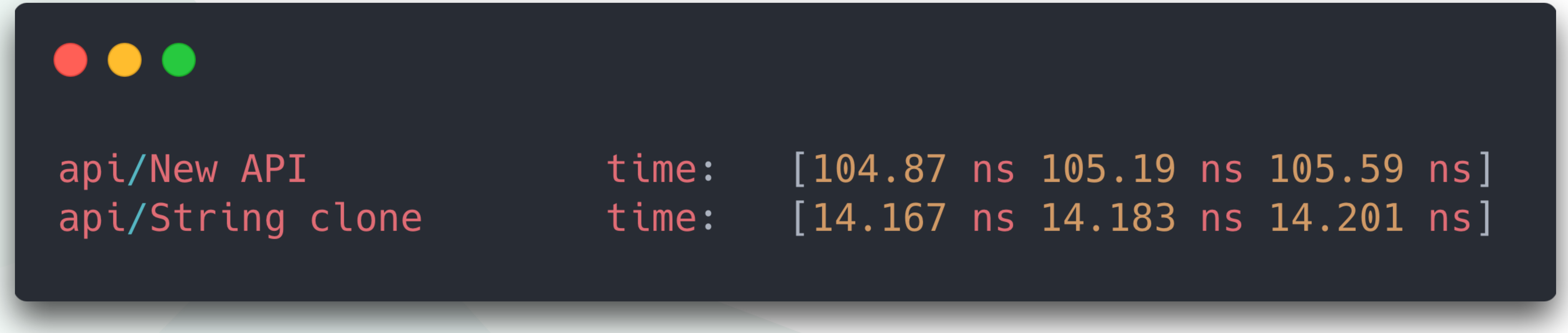
- „buffi“ and `buffi_macro` on crates.io
- Mastodon
  - SwishSwushPow@mastodon.social
  - weiznich@social.weiznich.de
- GitHub: https://github.com/GiGainfosystems/buffi
- Email: bjoern.wieczoreck@giga-infosystems.com

- Or just approach us during the conference!

Generating ergonomic C++ APIs

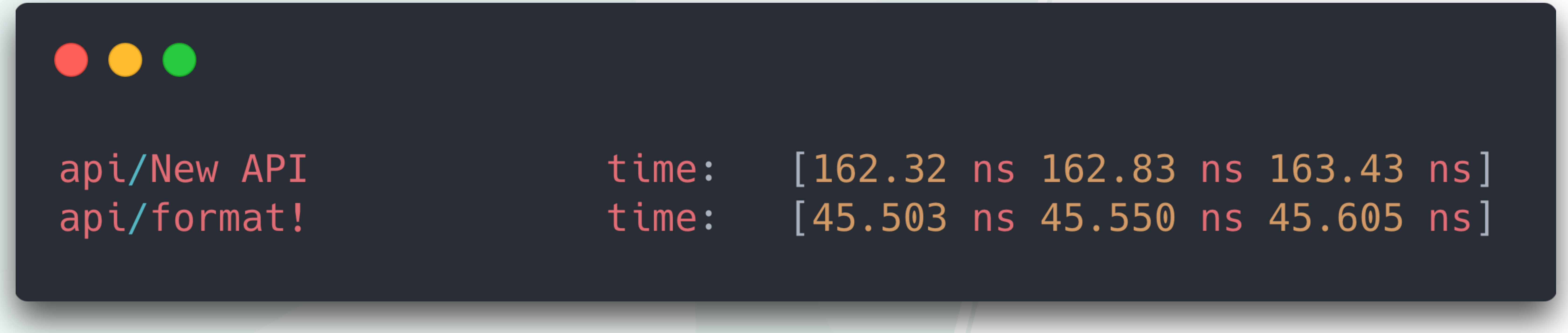
A bit of benchmarking

- **„String::clone“** Benchmark (String goes in and is returned)



```
api/New API                 time:   [104.87 ns 105.19 ns 105.59 ns]
api/String clone            time:   [14.167 ns 14.183 ns 14.201 ns]
```

- **„format!“** Benchmark (String goes in and is used in format! call, combined String is returned)



```
api/New API                 time:   [162.32 ns 162.83 ns 163.43 ns]
api/format!                 time:   [45.503 ns 45.550 ns 45.605 ns]
```