Custom Test Setup for Stateful Applications

Feat. Proc Macros

The problem

What we do

- Stateful tests working directly with a database
- Many database backends

What we want

- Run the same test across different databases, with some configurability
- Readability
- Maintainability
- Concurrency

First solution

```
#[test]
fn unique_in_conjunction_with_custom_column_name_must_work() {
   test_each_connector(|sql_family, api| {
       let dm1 = r#"
           model A {
               id Int @id
               field String @unique @map("custom_field_name")
           }
       "#;
       let result = infer_and_apply(api, &dm1).sql_schema;
       let index = result
           .table_bang("A")
           .indices
           .iter()
           .find(|i| i.columns == &["custom_field_name"]);
       assert_eq!(index.is_some(), true);
       assert_eq!(index.unwrap().tpe, IndexType::Unique);
   });
}
```

First solution (1).ppt

Pseudo-code version of test_each_connector:

```
fn test_each_connector(f: F) where F: Fn(...) {
     println!(“---- Testing mysql -----”);
     let database = get_mysql_database().unwrap();
     let test_api = mysql_test_api(database).unwrap();
     test_fn(SqlFamily::Mysql, test_api);
     // … then do the same for mysql 8, postgres 10, sqlite
}
```

What works

- Easy to implement
- As custom as needed
- Can do setup before running the group of tests

What doesn’t work

- We didn’t isolate state
  - In practice we used `--test-threads=1` and a custom `test.sh`
- Hard to distinguish tests
  - One test is many tests
  - Limited concurrency
- Inflexible in practice
  - We didn’t want to build a config object for each test, so we had multiple helpers
    - Inconsistent across the codebase
    - Lots of unrelated code needed to be updated when adding a connector
- test(|_, api| { n o i s y })
  - Extra indentation!

The final straw

From:

```
#[test]
fn unique_in_conjunction_with_at_map_must_work() {
   test_each_connector(|sql_family, api| {
       todo!(“logic goes here”);
   });
}
```

The final straw

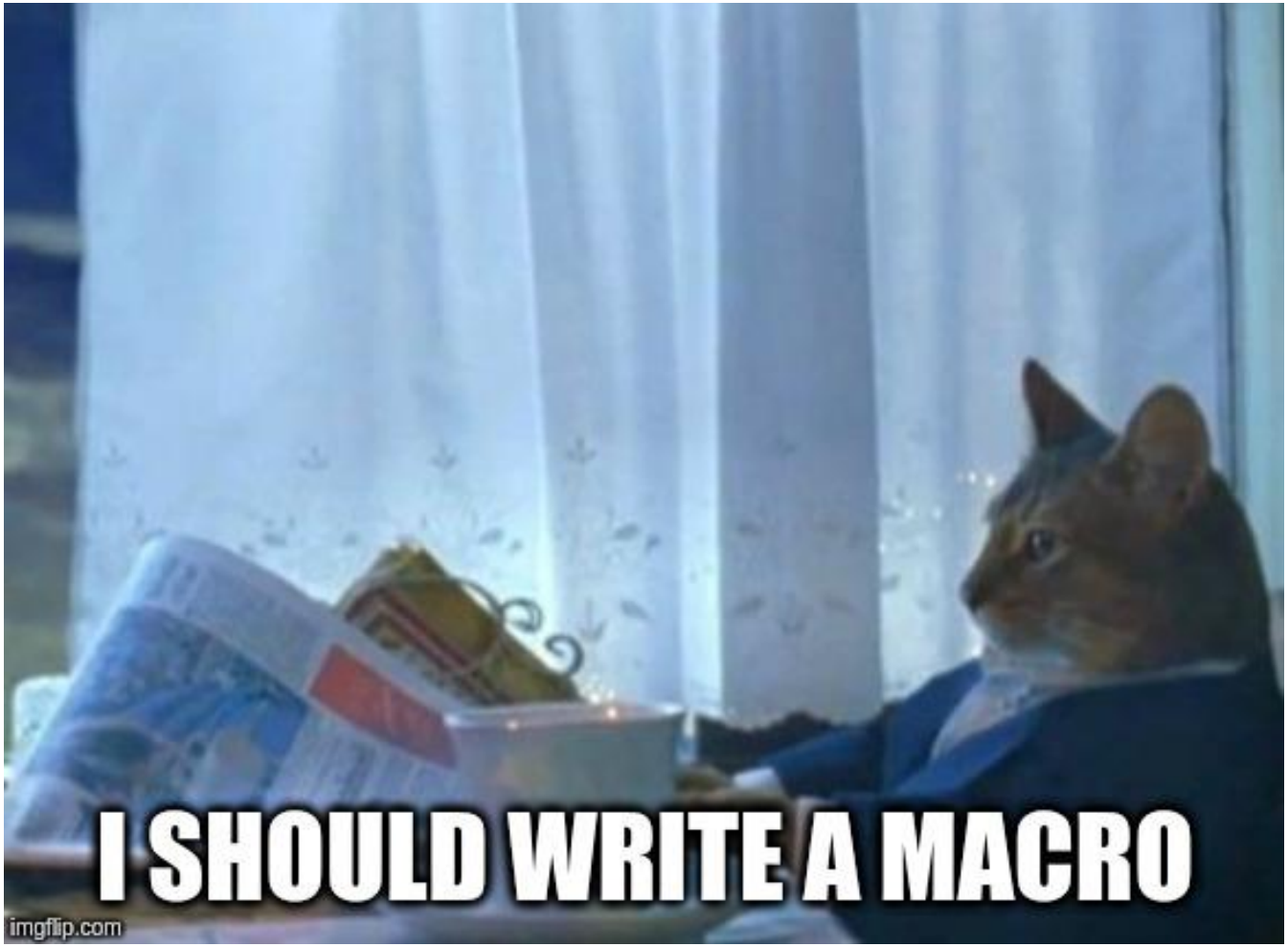
To:

```
#[test]
fn unique_in_conjunction_with_at_map_must_work() {
   test_each_connector(|sql_family, api| async {
       todo!(“logic goes here”);
   }.boxed());
}
```

The final straw

To:

```
#[test]
fn unique_in_conjunction_with_at_map_must_work() {
   test_each_connector(|sql_family, api| async {
       todo!("logic goes here");
   }.boxed());
}
```


I SHOULD WRITE A MACRO
imgflip.com

What we did next

- We knew we wanted a proc macro attribute for tests
  - like async_macros or `tokio::test`
- We couldn’t use these
  - Repetitive test setup + multiple tests per function + async
  - -> fuuuuuusion

Quick primer on procedural macros

- fn(TokenStream) -> TokenStream
- Attributes macro can rewrite their items

Important crates

- syn: parse TokenStreams into ASTs
- quote: generate rust code (`quote!(pub fn left_pad() { todo!() })`)
- darling: serde-derive for rust attributes
- cargo-expand: see what the macros expand to

Example

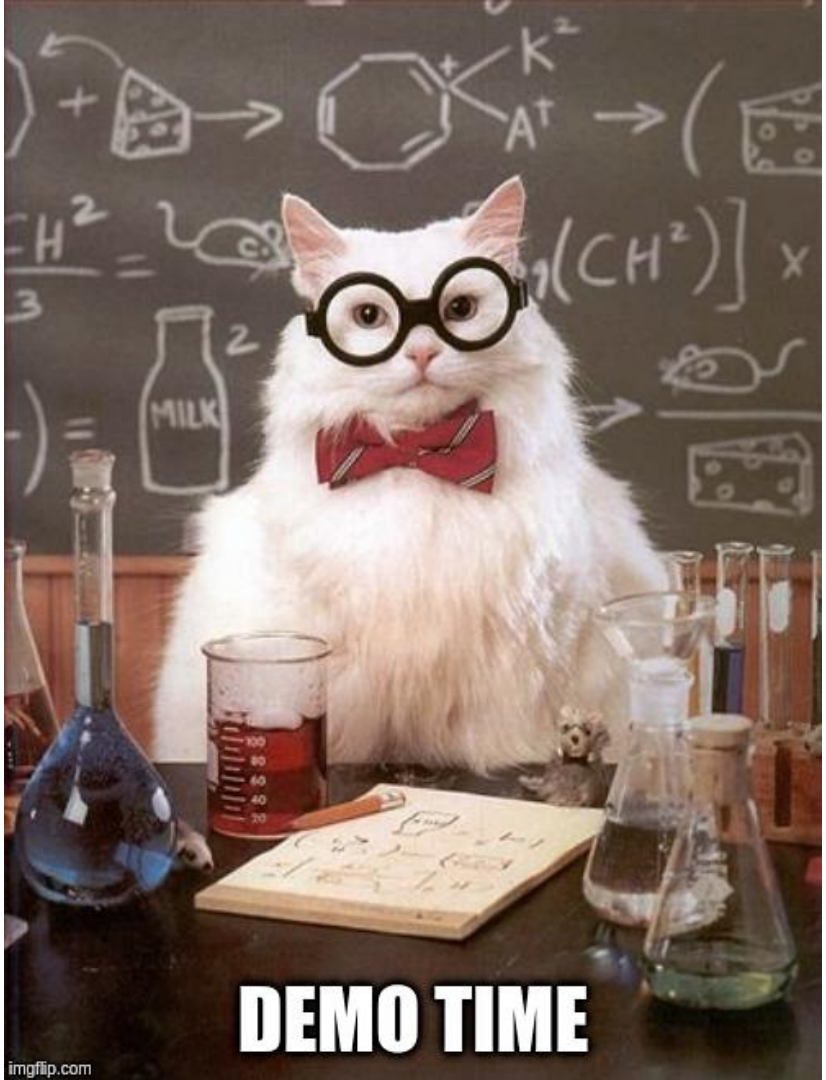
```
extern crate proc_macro;

use proc_macro::TokenStream;
use syn::ItemStruct;

#[proc_macro_attribute]
pub fn make_public(attr: TokenStream, item: TokenStream) -> TokenStream {
   let item = syn::parse_macro_input!(item as ItemStruct);
   quote::quote!( pub #item ).into()
}
```

Example (1).jpg

```
use make_public::make_public;

#[make_public]
struct Private;

// expands to:

pub struct Private;
```

Sketch of the transformation

- You write an async function that takes the test setup, slap the #[test_each_connector] macro attribute on it.
- For each connector, the macro will:
  - Produce a regular #[test] function that sets up the database, the async runtime and blocks on the function you wrote.


MILK
DEMO TIME
imgflip.com

Good sides

- The tests fail in isolation
- The tests are easy to filter
  - `cargo test postgres`
- Visually much cleaner

Good sides

- Now runs with just `cargo test`
- Very flexible/decoupled/customizable
  - The macro does not care about TestApi implementation
  - You only need a TestApi constructor for each connector you want to support
  - New TestApi structs are fast to build (e.g TestApi simulating multiple users)
-
- The macro knows a lot about your code and it can generate boilerplate for you
  - Example: optional return type, automatic unique database name, etc.

Bad sides

- Macros are harder to implement and understand
  - Rust metaprogramming is *very* powerful
  - -> depend on regular crates inside your macros, it’s doable and good, actually
- The macro attributes are not as discoverable as functions in a `test_setup` module (document them!)
- No teardown

Learnings

- Shared-nothing test setup is often the easiest to implement
  - No premature reuse of test setup. Rust is fast.
- Proc Macros are good
- Use with moderation
- In the future: custom test frameworks

Links

- The proc macro chapter in the Rust Book https://doc.rust-lang.org/book/ch19-06-macros.html?highlight=procedural#procedural-macros-for-generating-code-from-attributes
- ERFC for custom test frameworks https://github.com/rust-lang/rfcs/pull/2318
- Blog post on how the built-in #[test] attribute works https://blog.jrenner.net/rust/testing/2018/07/19/test-in-2018.html
- serial_test crate https://crates.io/crates/serial_test