



Tokio Tracing Assertions

(where I convince you to maybe use a crate I wrote)

Lucas Kent - <https://hachyderm.io/@rukai>

The project - <https://github.com/shotover/tokio-bin-process>

Hi I'm Lucas,
I'm going to give a talk on making assertions on tokio tracing
events using a crate I wrote.



A common scenario

- We are building a backend service
- It uses tokio tracing
- Uses a json subscriber in production
- Panics are piped through tracing when in json mode

Human readable log `2023-04-27T22:28:10.428088Z INFO shotover::runner: Starting Shotover 0.1.10`

Json log `{"timestamp":"2023-04-27T22:28:19.228631Z","level":"INFO","fields":{"message":"Starting Shotover 0.1.10"},"target":"shotover::runner"}`

I'm going to start by painting a best-practice scenario for a backend service.

Say we are building a backend service and make use of tokio tracing to log different events ranging from debug info to errors. When running on a developers machine we just write the logs to stdout in the standard human readable format.

However when running in production we do the right thing and write the logs as json so they can be processed programmatically by whatever service it is that does that.

Additionally, when running tracing in json mode we want to overwrite the panic hook so that all panics are logged as tracing errors instead of being displayed in the default human readable format on stdout.

The default panic handler would be no good as the log service can only process json output.

However when running tracing in human readable mode we need to keep the default panic handler as it's a lot easier to read the

default panic handler than a panic piped through a tracing error.

On the slide I have a human readable tracing log and the equivalent json format log.



A desire

We want our integration tests to:

- Assert no errors/warnings
- Assert expected logs occurred

In our many integration tests it would be really nice if we had these properties:

* first we want to assert that we are free of unexpected errors and warnings

Without this we will later find that our old tests are full of errors when we come back to them.

* we also want to assert that expected logs occur

Being able to assert that an `info` log occurs is a powerful way to assert an internal behaviour is functioning as we expect. Say for example that a faster path is taken in certain circumstances

However there isn't currently a good way to achieve these properties



The problem

There are lots of libraries around that do something similar:

- <https://github.com/dbrgn/tracing-test>
- <https://github.com/d-e-s-o/test-log>
- <https://github.com/tobz/tracing-fluent-assertions>

No good because:

- I had a bad experience with them
- I want to fail by default on an error/warning
- Cant run in a separate process

Rust tends to guide you towards statically linking your application into your integration tests.

From there the obvious solution is to make a custom tracing subscriber that directs all tracing events into a big vector of events that we can make assertions on.

There are lots of libraries around that do something similar:

- * <https://github.com/dbrgn/tracing-test>
- * <https://github.com/d-e-s-o/test-log>
- * <https://github.com/tobz/tracing-fluent-assertions>

However they have some problems:

- * For some reason these libraries never worked well for me...

Maybe it had something to do with the application I was working on creating its own subscriber.

My memory is a little fuzzy, it was a long time ago.

And when I tried to implement my own library with this approach I drowned in tracing's type hell.

Maybe it's just user error on my side but it certainly pushed me

away from this approach.

- * I also want to fail by default if an error or warning occurs!
 - + None of these libraries do this for me.
 - + I shouldn't have to manually specify for every test that errors and warnings should fail the test
 - + yes this is opinionated, but it's a good opinion
- * And finally, the application has to run in the same process as the integration test.



Why a separate process is important

- Allows running concurrent tests
- Allows asserting on separate instances
- Provides a true integration test
- Allows testing service shutdown

Now what's wrong with running the the application in the same process as the integration test.

This lack of isolation sounds minor but actually has a lot of important things tied up in it.

- + We cant run multiple tests concurrently due to tracing (reasonably) relying on global state

- + We can't assert separately on different instances of the application, again due to tracing global state. This is useful for testing clusters of services.

- + It's not a true integration test because we aren't testing against the final binary:

- At a minimum we need to inject a custom tracing subscriber.


- But it also enables other potential hacks where we inject custom logic into the application to test something instead of relying on just observing its behaviour, I know I fell into this trap.

- + Another thing is we can't test the shutdown procedure.

In production we need to be able to send the service a sigterm, have it complete in progress tasks, then exit with an exit

code.

If we aren't running in a separate process then naturally this is all impossible to test.



So what now?

Lets parse the JSON logs of a process!

I think the existing libraries would be a good fit for unit testing a library but for integration testing a full service we can clearly do better!

So I took a step back and reevaluated this, what if we ran the application binary in a separate process with json tracing enabled? Then I get easy machine processable logs and assert on those! With the extra benefit that our integration tests are running closer to how they run in production!




The solution: tokio-bin-process

- Run the binary in json mode
- Process the json
- Make assertions
- Fail on error/warn by default

So I built the tokio-bin-process crate to solve this. It does the following:

- * run the crate binary in a separate process, set to json mode.
- * process the json events into event structs as they are emitted
- * allow the test to make assertions on the events received so far
- * when the process is shutdown assert that no error or warning events occurred, unless they are explicitly allowed



Fun hack

We need print debugging, so just reimplement tracings human formatter!

```
shotover 06:37:31.974719Z INFO shotover::runner: Starting Shotover 0.1.9
shotover 06:37:31.974780Z INFO shotover::runner: configuration=Config { main_log_level: "info,shotover_proxy=info", observability_interface: "0.0.0.0:9001" }
shotover 06:37:31.974800Z INFO shotover::runner: topology=Topology { sources: {"cassandra_prod": Cassandra(CassandraConfig { listen_addr: "127.0.0.1:9042", connection_limit: None, hard_connection_limit: None, tls: None, timeout: None })}, chain_config: {"main_chain": TransformChainConfig(DebugForceEncodeConfig { encode_requests: true, encode_responses: true }, CassandraSinkSingleConfig { address: "127.0.0.1:9043", tls: None, connect_timeout_ms: 3000, read_timeout: None })}}, source_to_chain_mapping: {"cassandra_prod": "main_chain"} }
shotover 06:37:31.974940Z INFO shotover::config::topology: Loaded chains ["main_chain"]
shotover 06:37:31.974989Z INFO shotover::sources::cassandra: Starting Cassandra source on [127.0.0.1:9042]
shotover 06:37:31.975069Z INFO shotover::config::topology: Loaded sources [{"cassandra_prod"}] and linked to chains
shotover 06:37:31.975108Z INFO shotover::server: accepting inbound connections
2023-04-14T06:37:34.917241Z WARN scylla::transport::connection: Response from the database contains a warning warning="USE <keyspace>` with prepared statements is considered to be an anti-pattern due to ambiguity in non-qualified table names. Please consider removing instances of `Session#setKeyspace(<keyspace>)` , `Session#execute(\"USE <keyspace>\")` and `cluster.newSession(<keyspace>)` from your code, and always use fully qualified table names (e.g. <keyspace>.<table>). Keyspace used: test_prepare_statements, statement keyspace: test_prepare_statements, statement id: 2418a6606d64ccd02a7805fc4416a352"
shotover 06:37:35.288697Z INFO shotover::runner: received SIGTERM
shotover 06:37:35.289277Z INFO shotover::runner: Shotover was shutdown cleanly.
test cassandra_int_tests::passthrough_encode::case_2_scylla ... ok
```

Alright, that sounds good, but an important part of logging when running tests, is being able to do print debugging.

We need tracing events to reach the user and it needs to happen in real time so that we still get clues to what went wrong in a hung test.

To achieve this we reimplement the tracing human display mode within tokio-bin-process.

Whenever it receives a JSON event from the process it writes it to stdout in the human format.

Sure it's duplicated logic, but I think its a well placed hack and I doubt the tokio tracing human format is going to change often.

In the screenshot we can see tokio-bin-process's reemitted info logs amongst a warning log created by the integration test itself.

The only thing that gives away the log was recreated via tokio-bin-process is that the date is missing from the timestamp and instead lists the name of the application there.

The date isnt useful in an integration test since it ideally completes

within a few minutes at most.

And including the name there allows us to tell where the log is coming from without throwing off the monospaced text alignment.



First, a helper

Now I'll run through what using the library is like.
Fair warning, if you dont like seeing code in slides it's time to close
your eyes.
click it again ahhhhhh



First, a helper


```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    )
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

For this example say we have a database that we are building called cooldb.

We create a test file for it in cooldb slash test slash test dot rs.



First, a helper

```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    )
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

Actually creating the BinProcess takes some boilerplate.
We don't want to repeat this boilerplate in every test, so we create a helper method to create it called `cooldb_process`

First, a helper

```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    )
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

In the implementation of that method we see the `BinProcess::start_binary` constructor takes the path to the binary we want to run.

We pass it `bin_path!("cooldb")`, this macro gets the path to the named binary in the current package.

Cargo conveniently provides the path to any binary in the same package as the running test or bench, and the `bin_path` macro makes use of that to find the path.

So we give it “cooldb” the name of the crate we are testing.




First, a helper

```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    )
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

The next important thing is a list of args we want to give our process.




First, a helper

```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    ),
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

At a minimum we need to tell our process to enable its json logging format however we decide to do that.



First, a helper

```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    )
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

But we can also provide any extra flags that we want to use in our test.

First, a helper


```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    )
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

Then we use BinProcess's waiting functionality to block asynchronously until the database is ready to accept connections. This is super important to do otherwise we might try to connect to our database before its ready, giving us intermittent failures :/ This is defined by an EventMatcher which is used all over the BinProcess API.

The EventMatcher is created via an additive Builder API, the more "with underscore" statements we add the more specific its matches will be.




First, a helper

```
/// you'll want a helper like this as you'll be creating this in every integration test.
async fn cooldb_process() -> BinProcess {
    // start the process
    let mut process = BinProcess::start_binary(
        // Locate the path to the cooldb binary from an integration test or benchmark
        bin_path!("cooldb"),
        "cooldb", // The name that BinProcess should prepend its forwarded logs with
        &[
            // provide any custom CLI args required
            "--foo", "bar",
            // tokio-bin-process relies on reading tracing json's output,
            // so configure the application to produce that.
            "--log-format", "json"
        ],
    )
    .await;

    // block asynchronously until the application gives an event indicating that its ready
    tokio::time::timeout(
        Duration::from_secs(30),
        process.wait_for(
            &EventManager::new()
                .with_level(Level::Info)
                .with_target("cooldb")
                .with_message("accepting inbound connections"),
        ),
    )
    .await
    .unwrap();
    process
}
```

cooldb/test/test.rs

Then once this is all complete we return the process handle so the test can keep it alive for as long as it needs.



Now,
The test

```
#[tokio::test]
async fn test_some_functionality() {
    // start the db
    let cooldb = cooldb_process().await;


    // connect to the db, do something and assert we get the expected result
    perform_test();

    // Shutdown the DB, asserting that no warnings or errors occurred,
    // but allow and expect a certain warning.
    // A drop bomb ensures that the test will fail if we forget to call this method.
    cooldb
        .shutdown_and_then_consume_events(&[
            EventMatcher::new()
                .with_level(Level::Warn)
                .with_target("cooldb::internal")
                .with_message("The user did something silly that we want to warn about")
        ])
        .await;
}
```

cooldb/test/test.rs

Now we can use our helper to write a test!

First we initialize our database by calling the cooldb helper we just wrote.



Now,
The test

```
#[tokio::test]
async fn test_some_functionality() {
    // start the db
    let cooldb = cooldb_process().await;

    // connect to the db, do something and assert we get the expected result
    perform_test();

    // Shutdown the DB, asserting that no warnings or errors occurred,
    // but allow and expect a certain warning.
    // A drop bomb ensures that the test will fail if we forget to call this method.
    cooldb
        .shutdown_and_then_consume_events(&[
            EventMatcher::new()
                .with_level(Level::Warn)
                .with_target("cooldb::internal")
                .with_message("The user did something silly that we want to warn about")
        ])
        .await;
}
```

cooldb/test/test.rs

Then we do the test logic, for cooldb this would mean creating a connection to the database and then checking that some specific functionality works.

Usually this does not involve BinProcess at all, because the database can be tested purely by sending and receiving messages. Sometimes though you will want to assert on an info log, which can be done via an EventMatcher.



Now,
The test

```
#[tokio::test]
async fn test_some_functionality() {
    // start the db
    let cooldb = cooldb_process().await;

    // connect to the db, do something and assert we get the expected result
    perform_test();

    // Shutdown the DB, asserting that no warnings or errors occurred,
    // but allow and expect a certain warning.
    // A drop bomb ensures that the test will fail if we forget to call this method.
    cooldb
        .shutdown_and_then_consume_events(&[
            EventMatcher::new()
                .with_level(Level::Warn)
                .with_target("cooldb::internal")
                .with_message("The user did something silly that we want to warn about")
        ])
        .await;
}
```

cooldb/test/test.rs

Then we shutdown the database by calling
`shutdown_and_then_consume_events`.

It takes a list of matchers which allow us to specify any errors or warnings that are expected.

If an error or warning occurs that doesn't match any of our matchers then a panic occurs to fail the test.

In this example we have allowed a specific warning that we expect to occur in this test.

Additionally `BinProcess` utilizes a drop bomb to enforce that `shutdown_and_then_consume_events` is always called.



Requirements on the binary

- Provide a way to set tracing to output in json mode
- Handle panics as ``tracing::error`s` in json mode.
- Ensure SIGTERM cleanly shutdown the application and exits with a return code

There are a few hard requirements that the application must meet to use tokio-bin-process:

- * It needs to provide a way to set tracing to output in json mode
- * It needs to handle panics as ``tracing::error`s` in json mode
- * It needs to ensure SIGTERM cleanly shutdown the application and exits with a return code

But they all happen to be best practices anyway so I think enforcing these is good actually.

Some complications

- `BinProcess::start_binary` does not work in some scenarios.
 - Running *in* examples/bins
 - Running other crates in same workspace
- `BinProcess::start_crate_name` can work around those cases but has many downsides.
 - Slower
 - Profile weirdness

```
let mut process = BinProcess::start_binary(  
    // Locate the path to the cooldb binary from an inte  
    bin_path!("cooldb"),  
    "cooldb", // The name that BinProcess should prepend  
    &[  
        // provide any custom CLI args required  
        "--foo", "bar",  
        // tokio-bin-process relies on reading tracing :  
        // so configure the application to produce that  
        "--log-format", "json"  
    ],  
)  
.await;
```

To finish up I think it's important to make clear some complications that arise in the use of `tokio-bin-process` due to some cargo design details.

Remember back in the code sample, how we used the `bin_path` macro to find where cargo built our binary?
Unfortunately there are some cases where that doesn't work.

Whenever cargo compiles a benchmark or integration test any binary crates in the same package will also be compiled.
Any cases outside of that means we cannot access the binary.
This includes running `tokio-bin-process` from within an example or bin.

And also using `tokio-bin-process` to run crates from another package within the same cargo workspace.

To work around these shortcomings `BinProcess` has an alternate constructor `BinProcess::start_crate_name``.
This constructor will invoke cargo to build the binary and locate

where it is.

Unfortunately this has a lot of downsides, it is both expensive to do and custom cargo profiles are reduced to their derived profile.

For example a custom profile derived from release is just built in the release profile.

So while it allows tokio-bin-process to be used in the cases I previously described, you should always prefer

`BinProcess::start_binary` where possible.



tokio-bin-process

- <https://github.com/shotover/tokio-bin-process>
- <https://crates.io/crates/tokio-bin-process>

And thats it.

Tokio-bin-process was developed to meet the needs of my project at work, but its published to github and crates.io under the apache license for anyone else who shares these needs.