Help with vec of async dyn traits

help

object88 July 31, 2025, 5:20pm 1

First off, let me admit that while I have been programming for a long while, I am fairly new to Rust. I have notions of “how things work in other languages”, and that is coloring my thought process here, on top of my unfamiliarity with Rust itself. Please bear with me.

My ultimate goal is to create *something* that will manage the lifecycle of several subroutines in my program. I am using `tokio` for concurrency. There are a few things I’m trying to achieve:

- Accept an arbitrary number of subroutines that will be managed.
- Subroutines will implement a common “interface” to manage their lifespan.
- Subroutines will be cancellable, such that when the program is asked to stop, I can cleanly halt all subroutines before the program terminates.
- Subroutines are not required to do the same things: there may be two that start up different `axum` servers on different ports, and another that watches for file changes, for example.
- Subroutines may have interdependencies, and the lifecycle management must not care about that.

Toward these goals, I have created a trait:

```
use async_trait::async_trait;
use tokio_util::sync::CancellationToken;

#[async_trait]
pub trait Runnable: Send + Sync {
 async fn run(&mut self, cancel_token: CancellationToken);
}
```

Why a trait? It felt like the first / closest rust concept I know to an interface in another language, and maybe that’s an initial flaw.

Also, does it really need to be `async`? Also unclear, let’s get into that.

In terms of implementing `Runnable`, I have only gotten so far as implementing an `axum` server:

```
use async_trait::async_trait;
use axum::{routing::get, Router};
use tokio::net::TcpListener;
use tokio_util::sync::CancellationToken;

use crate::lifecycle::Runnable;

pub struct Http {
 app: Option<Router>,
 listener: Option<TcpListener>,
}

// Removed for brevity, code that creates the Http & its app and listener

#[async_trait]
impl Runnable for Http {
 async fn run(&mut self, cancel_token: CancellationToken) {
   let app = self.app.take();
   let listener = self.listener.take();
   axum::serve(listener.unwrap(), app.unwrap()).with_graceful_shutdown(async move {
     cancel_token.cancelled().await
   }).await.unwrap();
 }
}
```

So, the idea here is that when the `cancel_token` is cancelled, the `axum` service should cleanly shutdown, and the `run` method will exit.

I have a `lifecycle` mod, that will manage `Runnable`s:

```
use async_trait::async_trait;
use tokio::task::JoinSet;
use tokio_util::sync::CancellationToken;

pub async fn run(cancel_token: CancellationToken, runnables: &'static mut Vec<Box<dyn Runnable>>) {

 let mut set = JoinSet::new();

 for r in runnables.iter_mut() {
   let c0 = cancel_token.clone();
   set.spawn(async move {
     r.run(c0).await;
   });
 }

 tokio::select! {
   _ = cancel_token.cancelled() => {
     // TBD...
   }
 }

 while let Some(res) = set.join_next().await {
   match res {
     Ok(_val) => println!("Task returned."),
     Err(e) => eprintln!("Task failed: {:?}", e),
   }
 }
}
```

And here, I'm concerned that I'm going down a rabbithole. I have a `&'static mut Vec<Box<dyn Runnable>>`… a reference to a static lifespan'ed mutable vector of box'ed runnables. Digging into this…

> Why `dyn`? Because `Runnable` is a trait, I think? Google's AI, if I were to believe it, says that `dyn` makes the trait dynamically dispatchable (as compared to static dispatch), and that's required because `Runnable` may be implemented by any sort of struct. That feels like it checks out.
>
> Why boxed? Because as I understand it, I need a fat pointer (pointer + vtable) in order to the trait object. I guess I don't understand why the vtable isn't part of the trait object, but Rust docs say that's what I need to do.
>
> Why `Vec`? Because Rust doesn't have variadic parameters. Fair.
>
> Why mutable? Because in some cases (as seen in `http` above), executing `run` can alter `self`.
>
> Why `'static`? I think because the compiler cannot detect that in `lifecycle::run`, the `rs` from `runnables.iter_mut()` can't outlive the `run` func. I mean, I suppose it's *possible*, but the final `while` loop should be guarding against that. I'm a little hazy on this part.

There is also something about box'ing because a `vec` requires `Sized` elements, and dyn traits are of unknown size? I think that adds up, but maybe I'm misunderstanding that also.

Circling back to the question, why is `Runnable::run` an `async` method? I looks like `JoinSet::spawn` requires it?

So, how is this intended to come together? In my main:

```
# kctxd is this crate / project...
use kctxd::{http, lifecycle::{self, Runnable}};
use tokio::signal;
use tokio_util::sync::CancellationToken;

#[tokio::main]
async fn main() {
       let token = CancellationToken::new();
       let lifecycle_token = token.clone();

       let http = ... // a new `http` instance
       let status_http = ... // another new `http` instance

       let mut v: Vec<Box<dyn Runnable + 'static>> = vec![Box::new(http), Box::new(status_http)];

       // Start lifecycle
       let task_handle = tokio::spawn(async move {
               lifecycle::run(lifecycle_token, &mut v)
       });

       tokio::select! {
               _ = signal::ctrl_c() => {
                       token.cancel();
               },
       }

       task_handle.await.unwrap();
}
```

Only… now I’m getting an error from the compiler, saying that v doesn’t live long enough:

```
error[E0597]: `v` does not live long enough
 --> kctxd/src/bin/main.rs:25:35
  |
21 |     let mut v: Vec<Box<dyn Runnable + 'static>> = vec![Box::new(http), Box::new(status_http)];
  |        ----- binding `v` declared here
...
25 |         lifecycle::run(lifecycle_token, &mut v)
  |        --------------------------------^^^^^^-
  |        |                               |
  |        |                               borrowed value does not live long enough
  |        argument requires that `v` is borrowed for `'static`
26 |     });
  |    - `v` dropped here while still borrowed
```

And that I have very little idea what to do with. I don’t think that the compiler is wrong; I just don’t know how to make my code right.

But also, I feel like I am headed down a very wrong path. Given my primary objective, are there other implementation patterns that I should be considering?

Thanks for reading this far!

quinedot July 31, 2025, 6:23pm 2

As an attempt to head off some confusion, I think something worth pointing out from the start is that Rust lifetimes are generally about the *duration of borrows*, and not about the *liveness of some value*. A Rust lifetime does not represent the duration from value creation to value destruction. They are not unrelated, as running a destructor can conflict with being borrowed, but they are not the same thing.

Rust lifetimes are also erased during compilation, and the borrow checker is a pass-or-fail test that does not alter the semantics of any compiling program.

So when you talk about...

object88:

> create *something* that will manage the lifecycle of several subroutines
> [...]
> Subroutines will implement a common “interface” to manage their lifespan
> [...]
> Subroutines may have interdependencies, and the lifecycle management must not care about that2.

...Rust lifetimes are not the mechanism you need.

---

object88:

> 5. Why `’static`?

For `tokio` specifically, it is a multi-threaded work-stealing async runtime which requires tasks to meet a `'static` bound. That means their type cannot contain any lifetimes other than `'static`. In practical terms, the tasks cannot contain borrows.

object88:

> ```
> pub async fn run(cancel_token: CancellationToken, runnables: &'static mut Vec<Box<dyn Runnable>>) {
> ```

You would have to leak the Vec to get a `&'static mut Vec<_>`. But a more proper fix is to take things by value.

```
 pub async fn run(
    cancel_token: CancellationToken,
-    runnables: &'static Vec<Box<dyn Runnable>>,
+    // Or even better some `I: IntoIterator<..>` but this is fine to start
+    runnables: Vec<Box<dyn Runnable + Send>>,
) {
```

```
    let mut set = JoinSet::new();
-    for r in runnables.iter_mut() {
+    for mut r in runnables {
```

Your notes about `dyn` and boxing are all pretty much accurate: If you want to support different tasks, you need to type erase them with `dyn`. The `dyn` is the original value, which does not include a vtable and which may have different sizes (based on what was type erased). Thus `dyn` is dynamically sized -- does not have a single size known at compile time -- is not `Sized`. To hold the vtable and deal with the dynamic size, you need some sort of wide pointer, and `Box<dyn ...>` is the canonical owning pointer.

object88:

> Given my primary objective, are there other implementation patterns that I should be considering?

I'm afraid I don't have any feedback on your overall goals or design, though.

3 Likes

firebits.io July 31, 2025, 8:20pm 3

Besides what quinedot mentioned, you might benefit from learning about `Futures` in Rust. They are the abstraction over concurrency in the language, and they are cancellable. To me, it sounds like you're trying to re-implement futures.

object88 October 9, 2025, 2:13pm 4

Just wanted to circle back and say thanks!

In the time since I asked the question, I actually lost the code... who does that in 2025? Apparently I do. I got stuck re-implementing, and your previous responses here got me past my block.

1 Like

Related topics

Topic	Replies	Views	Activity
Issue with boxed dyn trait, async and lifetimes help	3	532	May 1, 2023
Moving from struct to trait with tokio::spawn caused has an anonymous lifetime `'_` but it needs to satisfy a `'static` lifetime requirement help	10	1574	August 5, 2022
Satisfying tokio::spawn 'static lifetime requirement help	11	3853	October 21, 2022
Tokio 0.2 and lifetimes help	4	597	August 10, 2020
Struct with async methods and tokio ownership help	5	1632	June 28, 2022