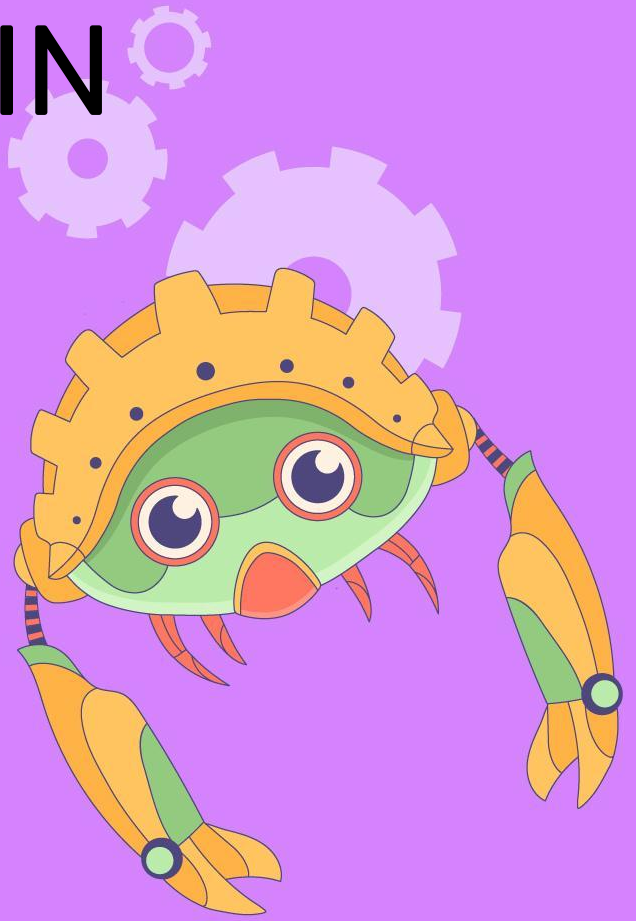
# CHRISTIAN VISINTIN

aka veeso || veeso_dev

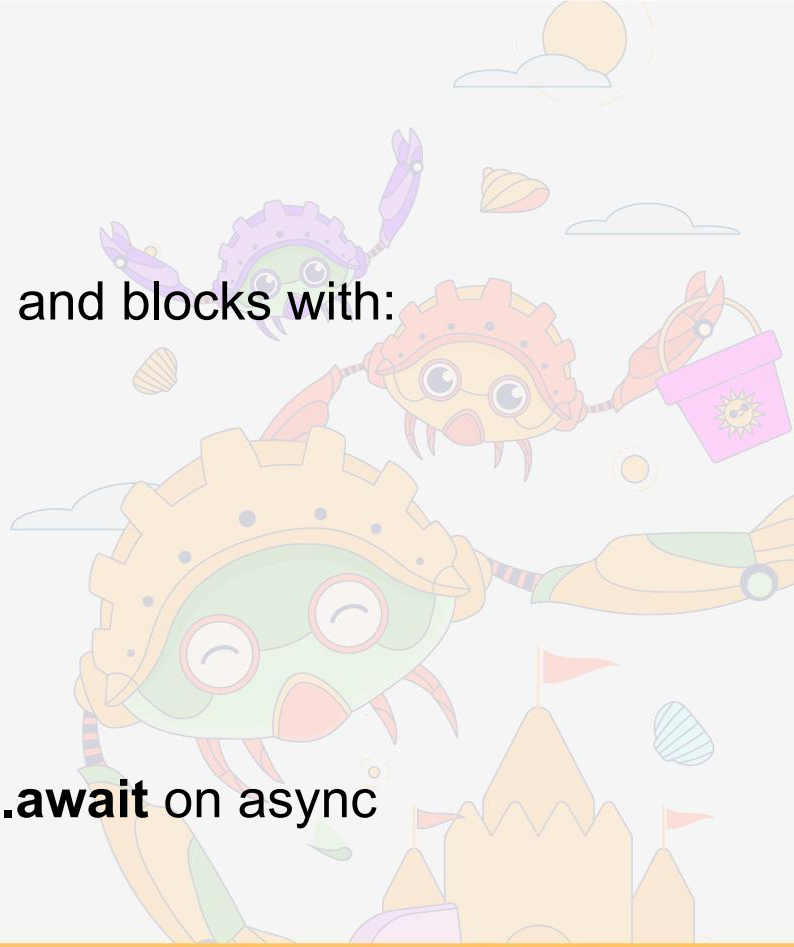
Rust Freelance Software Engineer | Rust OSS Developer

# WE CAN'T LIVE WITHOUT ASYNC AND THAT'S A PROBLEM



# THE ASYNC AREA BEGINS

7 November 2019 - Rust 1.39



You can now create async function and blocks with:

- *async fn*
- *async move {}*
- *async {}*

respectively, and you can now call **.await** on async expressions

The Async era begings

# YOU USE IT, BUT DO YOU KNOW WHAT ASYNC IS?

```rust
pub trait Future {
  type Output;

  fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

**Future** is a trait which provides the **poll** function, which can return two states:

- Pending
- Ready

# We can async everything

```rust
struct AsyncSum {
  a: u64,
  b: u64,
}

impl Future for AsyncSum {
  type Output = u64;

  fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) -> Poll<Self::Output> {
    Poll::Ready(self.a + self.b)
  }
}

fn sum(a: u64, b: u64) -> impl Future<Output = u64> {
  AsyncSum { a, b }
}

// equivalent to

async fn sum(a: u64, b: u64) -> u64 {
  AsyncSum { a, b }.await
}

```

The Async era begings

# JUST A CRAZE?

std::thread provides an awesome implementation of multi-threading in Rust!

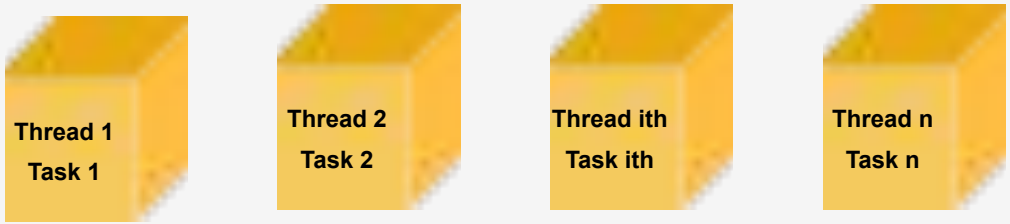
We've got:

- mpsc channels to communicate between threads
- Safe Mutex and RwLocks
- Arc for safe shared data

The Async era begings

# WHAT DO WE WANT TO SOLVE WITH ASYNC?

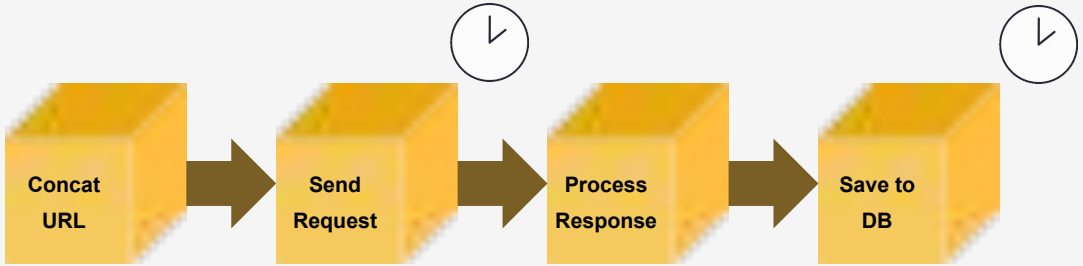
When we work with multi-theading we spawn a thread for each task we want to run



The Async era begings

BUT MANY TASKS ARE LAZY

Some steps of a task depend on external resources which won't "resolve" immediately.




Eventually, we run a lot of threads, which spend most of their lifetime sleeping 😴

# What are the **external resources**?

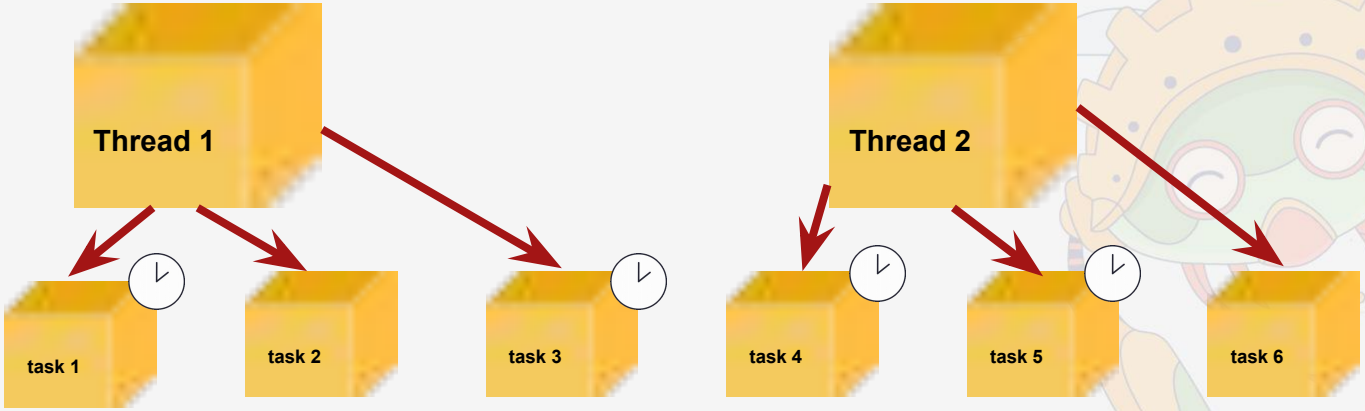
Basically, we need to await for tasks that depend on external factors

Mainly 3 cases:

- **I/O**: Reading from and writing to the FS depends on the filesystem itself. Actually, not all the FS are async (for instance, Linux is not unless you use uring).
- **Network**: Any interaction with the network creates a lot of delay, and we have many intermediate steps between the request and the response.
- **Time**: Time is async. It doesn't depend on our machine, but it depends on the universe. We can't control time, and the humankind is unhappy because of that

# How does async solve it for us?

The **Async Runtime** will spawn threads only when needed and assign tasks to existing threads based on load.
When a task is waiting, the next task is executed on the same/another thread.



The Async era begings

# LET'S DEPRECATE THREADS AND ASYNC EVERYTHING

It may be easy to think that tasks are always preferrable over threads

But, in opposition to what many devs think, it's not always true

The Async era begings

# WHY ASYNC TOOK OVER THREADS?

- We copy/paste code
- Easier to start with
- More boiler plates
- We think it works better and it’s safer, because the runtime handles things for us; aka we don’t know what it is

## What should we go for?

Ask yourself:

Is the amount of tasks/workers **deterministic** (*and also relatively small*) for my use case?

# Async, sync threads and... hybrid systems

## HTTP Server

### Async

We've got potentially thousands of requests at the same time. The host must optimise the requests among the machine cores. Also, requests may access the database or request data from other services, which can cause internal delays.

## Make

### Multi thread

The user specifies how many parallel threads they want to use to build the source code. Each thread dequeues the object to compile until the queue is empty.

## Solana Validator

### Hybrid

There is a deterministic number of internal services, each one running in a separate thread. There is also an interface for RPC and one using QUIC to communicate with other validators. These are NOT deterministic, and so, they run in an async pool.

# Sync/Async interop

## MY FIRST ~~APPROACH~~ CONFLICT WITH ASYNC RUST

Since 2021, I have been maintaining **suppaftp**, which has become Rust's main FTP client library.

It was derived from rust-ftp, which was unmaintained since Rust 1.2x.
One of the first requests I've received was to migrate to **Async Rust**.

What was the issue?

- I only needed the sync client on termscp!
- The sync version was still widely used and required in fully-sync apps.

# How to implement sync/async libraries?

We need to implement **two parallel versions**

```rust
fn open_file(p: &Path) -> std::io::Result<File>
{ std::fs::File::open(p)
}
```

```rust
async fn open_file(p: &Path) -> std::io::Result<File>
{ tokio::fs::File::open(p).await
}
```

...Which means a lot of **code duplication**

# Why don’t you just block_on?

```rust
use std::pin::Pin;
use std::task::{Context, Poll, Waker};
use std::time::Duration;

/// A simple runtime to execute async code
pub struct DumbRuntime;

impl DumbRuntime {
    pub fn block_on<F>(mut f: F) -> F::Output
    where
        F: Future,
    {
        let mut f = unsafe { Pin::new_unchecked(&mut f) };

        let mut ctx = Context::from_waker(Waker::noop());

        loop {
            println!("polling future");
            match f.as_mut().poll(&mut ctx) {
                Poll::Ready(val) =>
                    return val;
                }
                Poll::Pending => {
                    std::thread::sleep(Duration::from_micros(10));
                }
            }
        }
    }
}
```

Writing a simple **Future executor** is very easy

**BUT...**

We can’t always work around like that.
Some tokio types **require explicitly the tokio runtime** to be running.

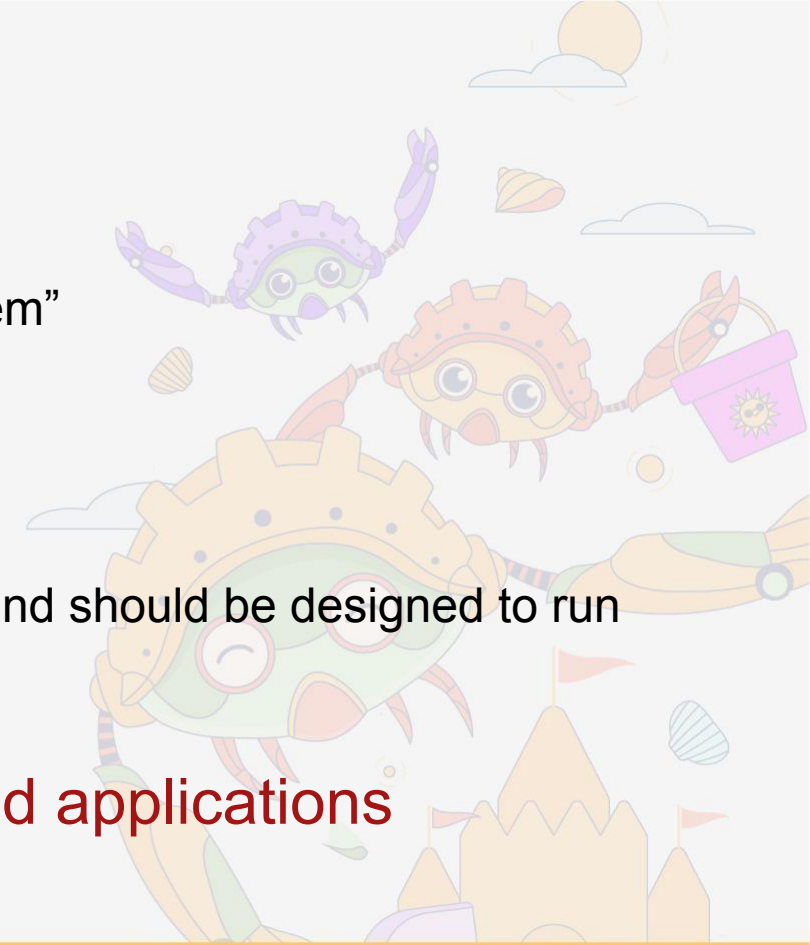
## Just use tokio

I've been told: "Use tokio and solve the problem"

But why should I?

Rust is about **performance and efficiency**, and should be designed to run anywhere with the lowest possible cost.

Async is forcing us to make bloated applications



# Reusing async/sync code

Can’t we just reuse all the logic with some feature gating?

- Not really. We often need the two systems to co-exist because of hybrid architectures

One could import the same library twice with different features though

- Correct, but we still have to face compatibility issues

# WHY IS ASYNC/SYNC INTER OPERABILITY AN ISSUE

There are numerous different patterns that cannot be achieved in the same way between asynchronous and synchronous code.

Sometimes it’s just too different and hard to handle.

# Boxed Functions

```rust
pub struct SyncStruct {
    callback: Box<dyn Fn(u64) -> Result<u64, String> + Send + Sync>,
}

pub struct AsyncStruct {
    callback: Box<
        dyn Fn(u64) -> Pin<Box<dyn Future<Output = Result<u64, String>> + Send +
Sync>>      + Send
            + Sync,
    >,
}
```

# Input/Output

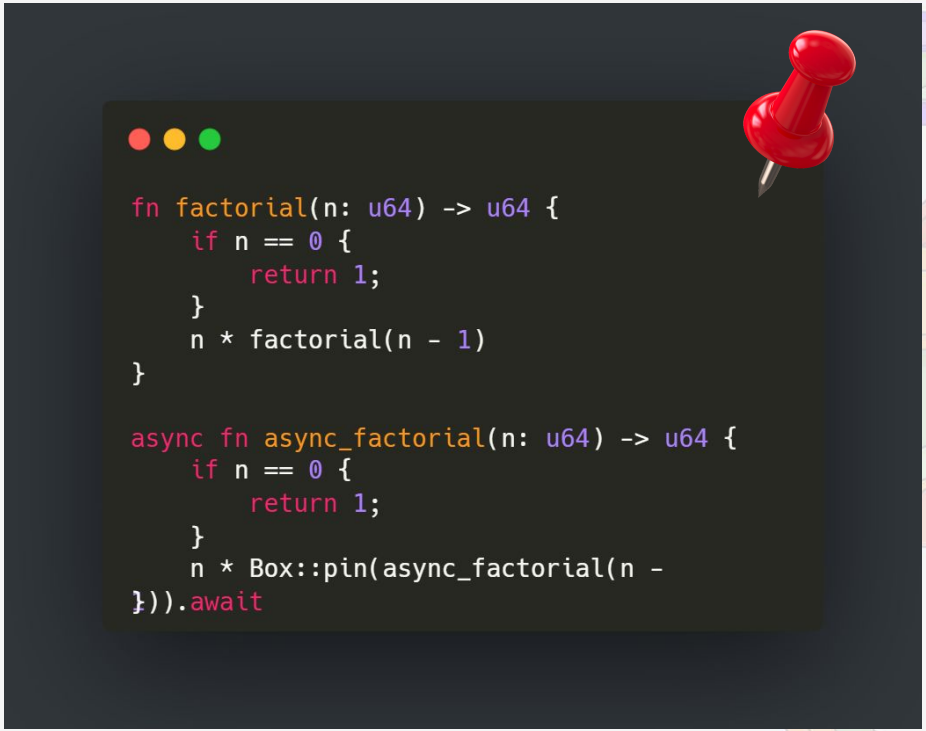
Basically, everything related to I/O is incompatible since it relies on different types and **async-specific traits**.

- Read, Write and Seek Vs. AsyncRead, AsyncWrite, AsyncSeek

# Same, but different

```rust
let sync_mutex = std::sync::Arc::new(std::sync::Mutex::new(5u64));
let async_mutex =
std::sync::Arc::new(tokio::sync::Mutex::new(5u64));
let value = sync_mutex.lock().unwrap();
let async_value = async_mutex.lock().await;
```

# Recursion



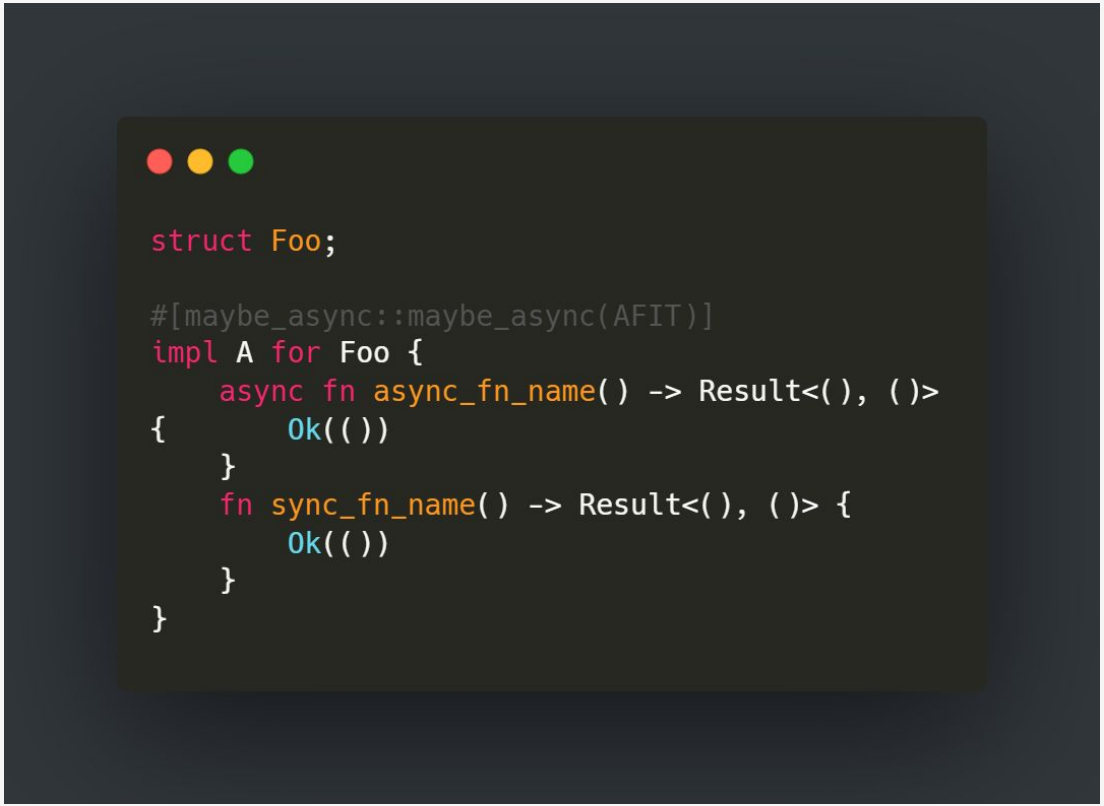
```rust
fn factorial(n: u64) -> u64 {
    if n == 0 {
        return 1;
    }
    n * factorial(n - 1)
}

async fn async_factorial(n: u64) -> u64 {
    if n == 0 {
        return 1;
    }
    n * Box::pin(async_factorial(n -
})).await
```

# Sync/Async interop

WE NEED AN INTEROP LAYER

MAYBE MAYBE-ASYNC



```rust
struct Foo;

#[maybe_async::maybe_async(AFIT)]
impl A for Foo {
    async fn async_fn_name() -> Result<(), ()>
{       Ok(())
    }
    fn sync_fn_name() -> Result<(), ()> {
        Ok(())
    }
}
```

- Removing async/await is not enough

Because again

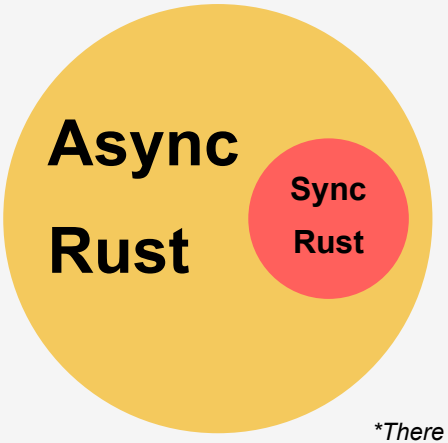
**Compatibility issues**

## Sync/Async interop

# A MATTER OF PERSPECTIVE

Async code is not compatible with Sync Rust

But, mostly* all Sync code is compatible with Async Rust




*There are exceptions though, such as recursion

maybe-fut

# INTRODUCING MAYBE-FUT

A Rust Library to export sync/async API from the **same code**

**Async over Sync**

Async and Sync API can **co-exist** (Yippee)

95% of the code is **proc macros** 🪄

# A wrapper around sync and async impls

```rust
#[derive(Debug, Unwrap)]
/// A reference to an open file on the filesystem.
pub struct File(FileInner);

/// Inner pointer to sync or async file.
#[derive(Debug)]
enum FileInner {
    /// Std variant of file <https://docs.rs/rustc-std-workspace-std/latest/std/fs/struct.File.html>
    Std(std::fs::File),
    #[cfg(tokio_fs)]
    #[cfg_attr(docsrs, doc(cfg(feature = "tokio-fs")))]
    /// Tokio variant of file <https://docs.rs/tokio/latest/tokio/fs/struct.File.html>
    Tokio(tokio::fs::File),
}
```

# We dispatch the type on the go

```rust
pub async fn open(path: impl AsRef<Path>) -> std::io::Result<Self> {
    #[cfg(feature = "tokio-fs")]
    {
        if crate::is_async_context() {
            tokio::fs::File::open(path).await.map(Self::from)
        } else {
            std::fs::File::open(path).map(Self::from)
        }
    }
    #[cfg(not(feature = "tokio-fs"))]
    {
        std::fs::File::open(path).map(Self::from)
    }
}
```

- And then we do the same for all methods

```rust
pub async fn metadata(&self) -> std::io::Result<std::fs::Metadata> {
    match &self.0 {
        FileInner::Std(inner) => inner.metadata(),
        #[cfg(feature = "tokio-fs")]
        FileInner::Tokio(inner) => inner.metadata().await,
    }
}
```

- But we can use macros to speed up the process

```
maybe_fut_method!(
    /// Queries metadata about the underlying file.
    metadata() -> std::io::Result<std::fs::Metadata>,
    FileInner::Std,
    FileInner::Tokio,
    tokio_fs
);
```

- Which is implemented as

```rust
/// A macro to create a method that can be used in both async and sync contexts.
#[macro_export]
macro_rules! maybe_fut_method {
    ($(#[$meta:meta])*
        $name:ident
        (
            $( $arg_name:ident : $arg_type:ty ),* $(,)?
        )
        -> $ret:ty,
        $sync_inner_type:path,
        $async_inner_type:path,
        $feature:ident
    ) => {
            $(#[$meta])*
            pub async fn $name( &self, $( $arg_name : $arg_type ),* ) -> $ret {
                match &self.0 {
                    $sync_inner_type(inner) => inner.$name( $( $arg_name ),* ),
                    #[cfg($feature)]
                    $async_inner_type(inner) => inner.$name( $( $arg_name ),*
).await,        }
            }
        };
}
```

maybe-fut

# HOW DOES THE TYPE DISPATCHER WORK?

- We can get to know wheter we're async with this simple snippet
- We must query it every time; because (again) of hybrid systems!

```rust
#[inline]
pub fn is_async_context() -> bool {
    #[cfg(tokio)]
    {
        tokio::runtime::Handle::try_current().is_ok()
    }
    #[cfg(not(tokio))]
    {
        false
    }
}
```

- Did you re-export the entire tokio API?

# Of course I did

- But how is the API exported then?

With the power of **Proc Macros!!!**

```rust
use std::path::{Path, PathBuf};

use maybe_fut::fs::File;

struct FsClient {
    path: PathBuf,
}
```

```rust
#[maybe_fut::maybe_fut(
    sync = SyncFsClient,
    tokio = TokioFsClient,
    tokio_feature = "tokio"
)]
impl FsClient {
    /// Creates a new `FsClient` instance.
    pub fn new(path: impl AsRef<Path>) -> Self {
        Self {
            path: path.as_ref().to_path_buf(),
        }
    }

    /// Creates a new file at the specified path.
    pub async fn create(&self) -> std::io::Result<()> {
        // Create a new file at the specified path.
        let file = File::create(&self.path).await?;
        file.sync_all().await?;

        Ok(())
    }
}
```

- Which allows us to use the exported types

```rust
fn sync_main(path: &Path) -> Result<(), Box<dyn std::error::Error>> {
    println!("Running in sync mode");

    let client = SyncFsClient::new(path);
    client.create()?;

    Ok(())
}

#[cfg(feature = "tokio")]
async fn tokio_main(path: &Path) -> Result<(), Box<dyn std::error::Error>>
{   println!("Running in async mode");

    let client = TokioFsClient::new(path);
    client.create().await?;

    Ok(())
}
```

# How is the sync code generated from async?

```
if is_async && !async_methods {
    quote! {
        #(#attrs)*
        #visibility #constness fn #method_name(#args) #ret_type {
            ::maybe_fut::SyncRuntime::block_on(
                #fn_body
            )
        }
    }
} else {
    quote! {
        #(#attrs)*
        #visibility #constness #asyncness fn #method_name(#args) #ret_type {
            #fn_body
        }
    }
}
```

maybe-fut

# HOW DID I DEAL WITH IO?

As we've seen before IO is one of those incompatible things between sync/async

So I had to reimplement here also the Read/Seek/Write traits, but with Async, and all the io types have been re-implemented

# What about performance?

- I was quite concerned about querying every time **is_async_context**

```
is_async_context        time:   [3.3511 ns 3.3567 ns 3.3621
Found 4 outliers among 100 measurements (4.00%)
  3 (3.00%) high mild
  1 (1.00%) high severe
```

- What about the **overhead**?

```
tokio_create_file       time:   [11.529 µs 11.620 µs 11.715
maybe_fut_create_file   time:   [11.603 µs 11.696 µs 11.786
```

WANT TO KNOW MORE?

**maybe-fut**

https://github.com/veeso/maybe-fut

## Conclusions

# DOES THIS FIX ALL THE PROBLEMS?

## No, but...

- It should at least raise the concern that library maintainers have when providing both synchronous and asynchronous code.
- Should we find a std solution for async interop?
- Awaitable std io library

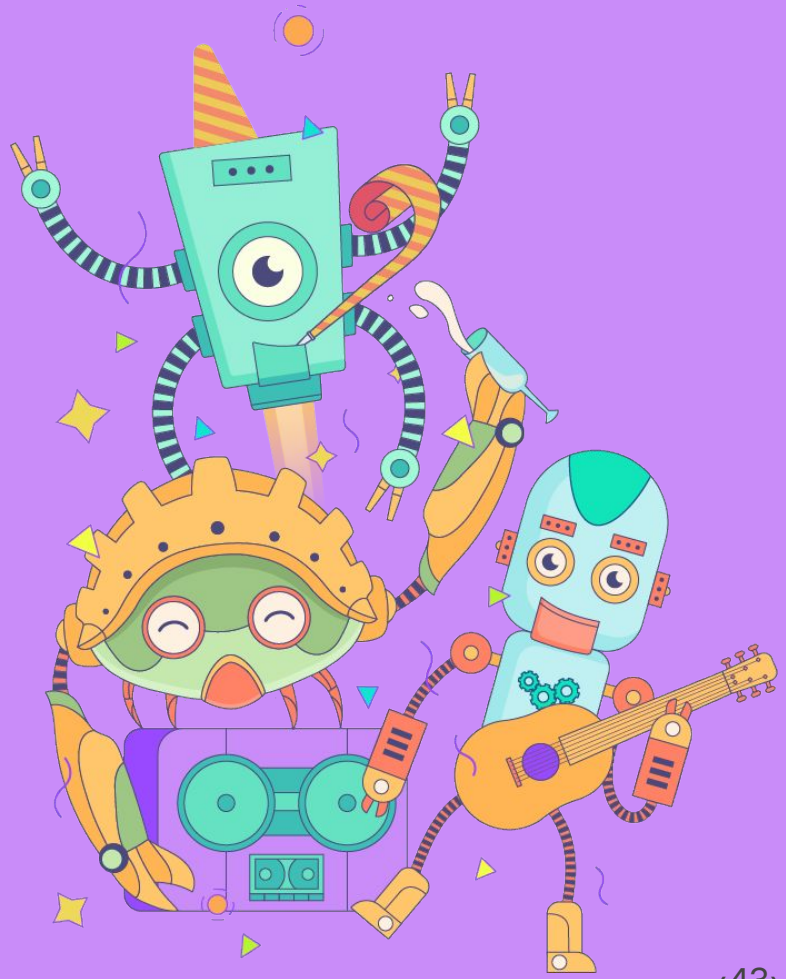
## Conclusions

# SHOULD WE USE MAYBE-FUT IN THE MEANTIME

Probably not. While most of the cases are covered by the library, there are mainly two issues:

- Missing support for third party libraries
- It’s more an experiment than something serious.

# Thank you for your attention!



CHRISTIAN VISINTIN

christian.visintin@veeso.dev

@veeso_dev@hachyderm.io

github.com/veeso

blog.veeso.dev