The Talk you've been .await-ing for

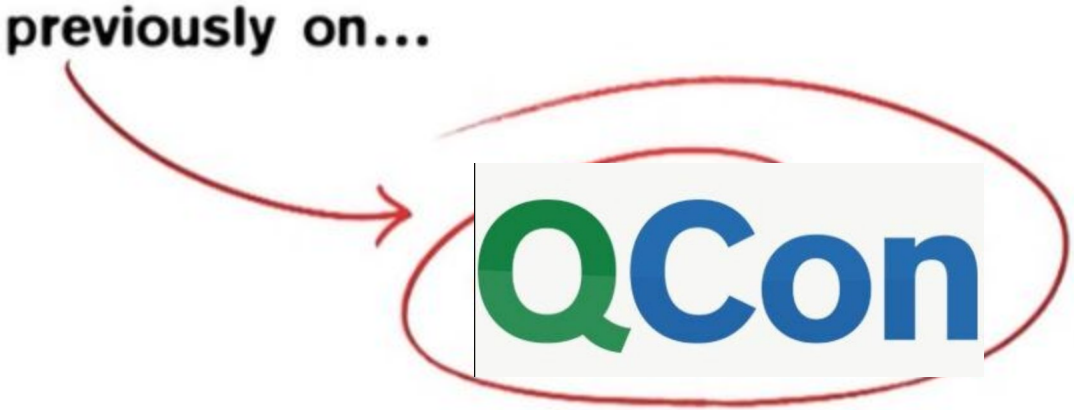
@steveklabnik

The `.await` is over, `async fn`s are here

Previously in Rust 1.36.0, we announced that the `Future` trait is here. Back then, we noted that:

> With this stabilization, we hope to give important crates, libraries, and the ecosystem time to prepare for `async` / `.await`, which we'll tell you more about in the future.

A promise made is a promise kept. So in Rust 1.39.0, we are pleased to announce that `async` / `.await` is stabilized! Concretely, this means that you can define `async` functions and blocks and `.await` them.

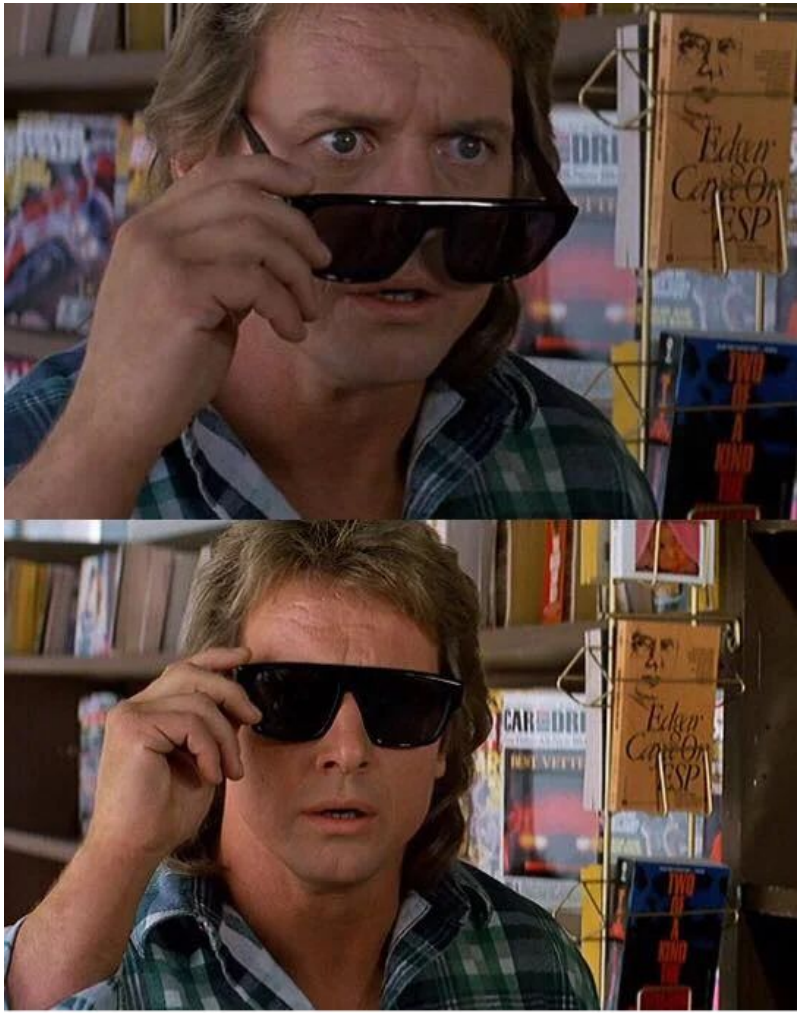

previously on...
QCon

#Rust #Async #ProgrammingLanguage

Rust's Journey to Async/Await

944 views • Nov 11, 2019

54 0 SHARE SAVE




```
async fn foo(s: String) -> i32 {
    // …
}
```

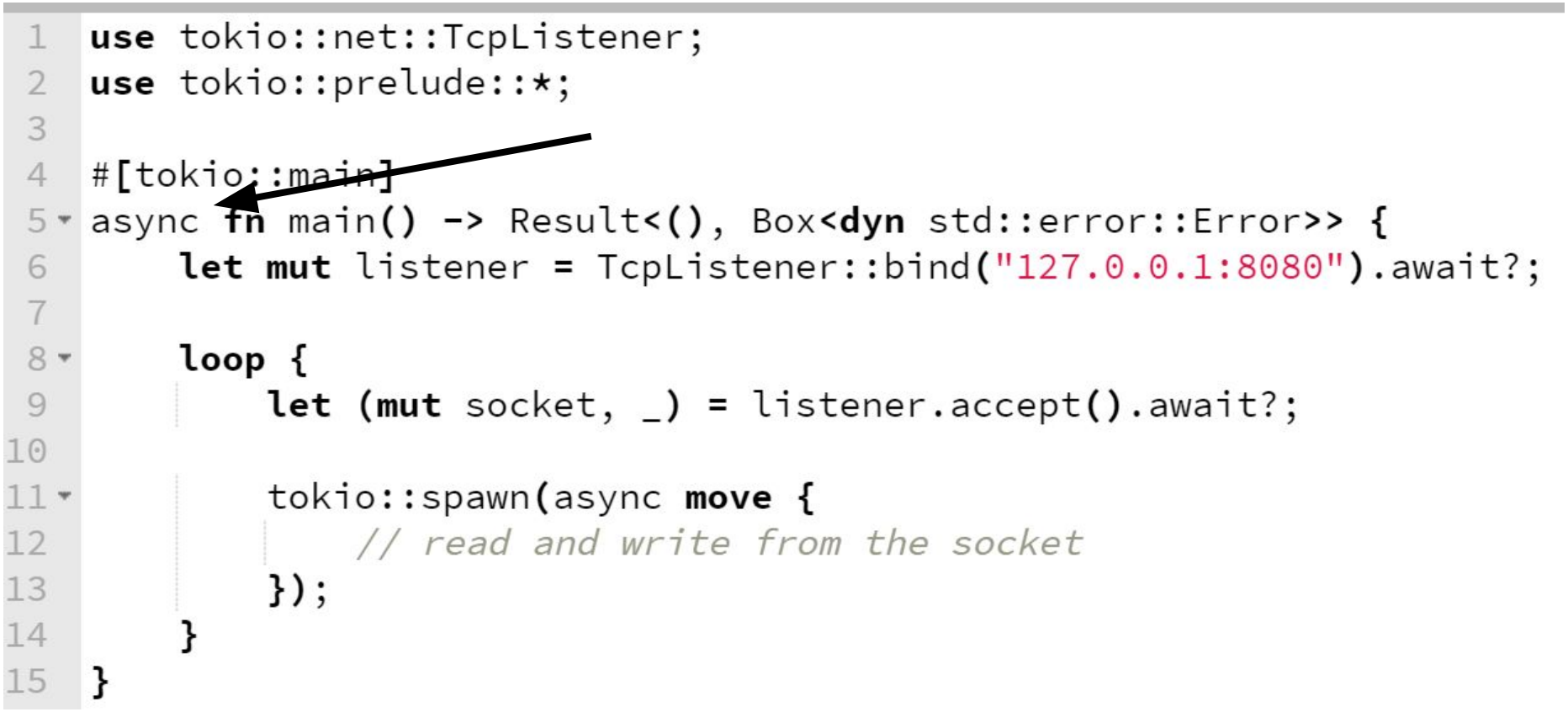
```
fn foo(s: String) -> impl
Future<Output=i32>  {
    // …
}
```

Stuff we're going to talk about

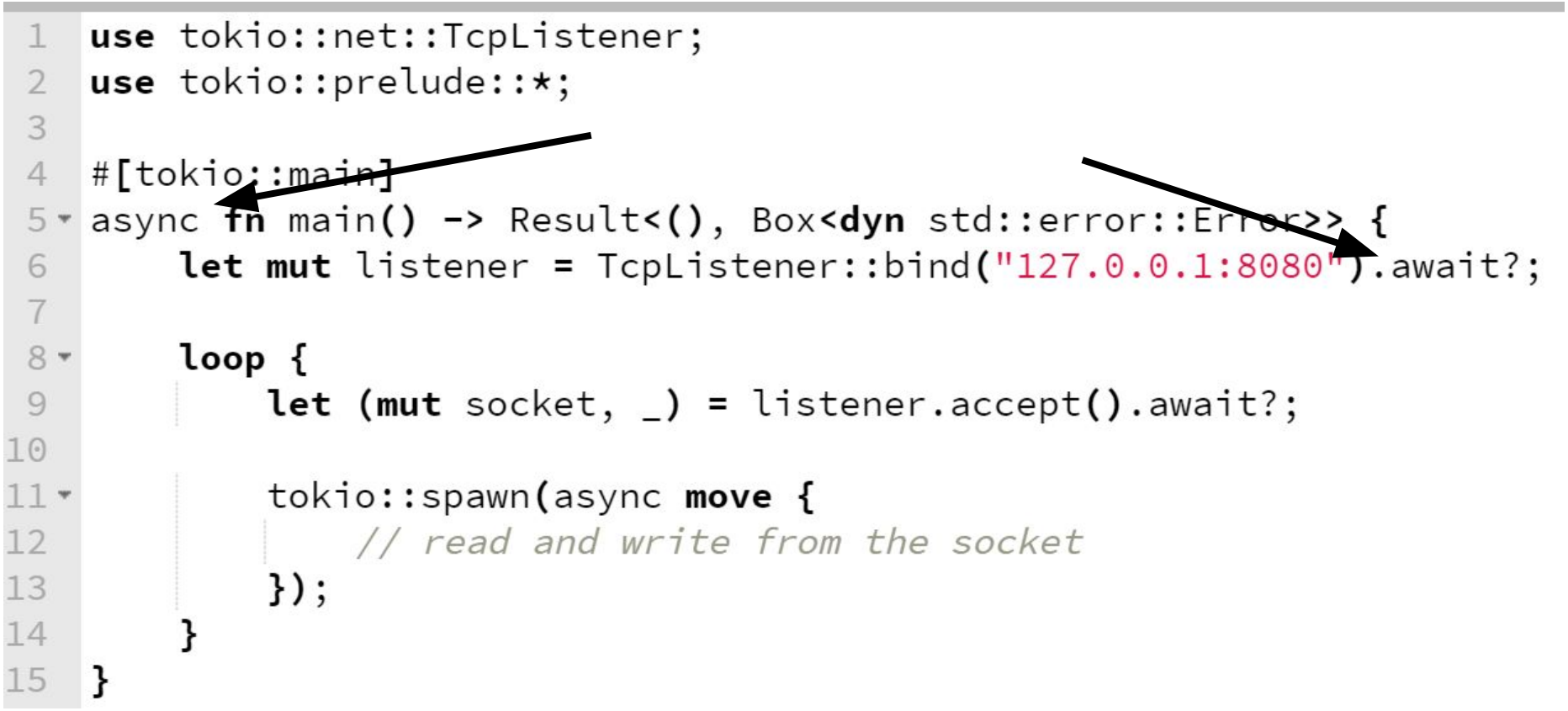
- async/await and Futures
- Generators: the secret sauce
- Tasks, Executors, & Reactors, oh my!
- … maybe async fn in traits

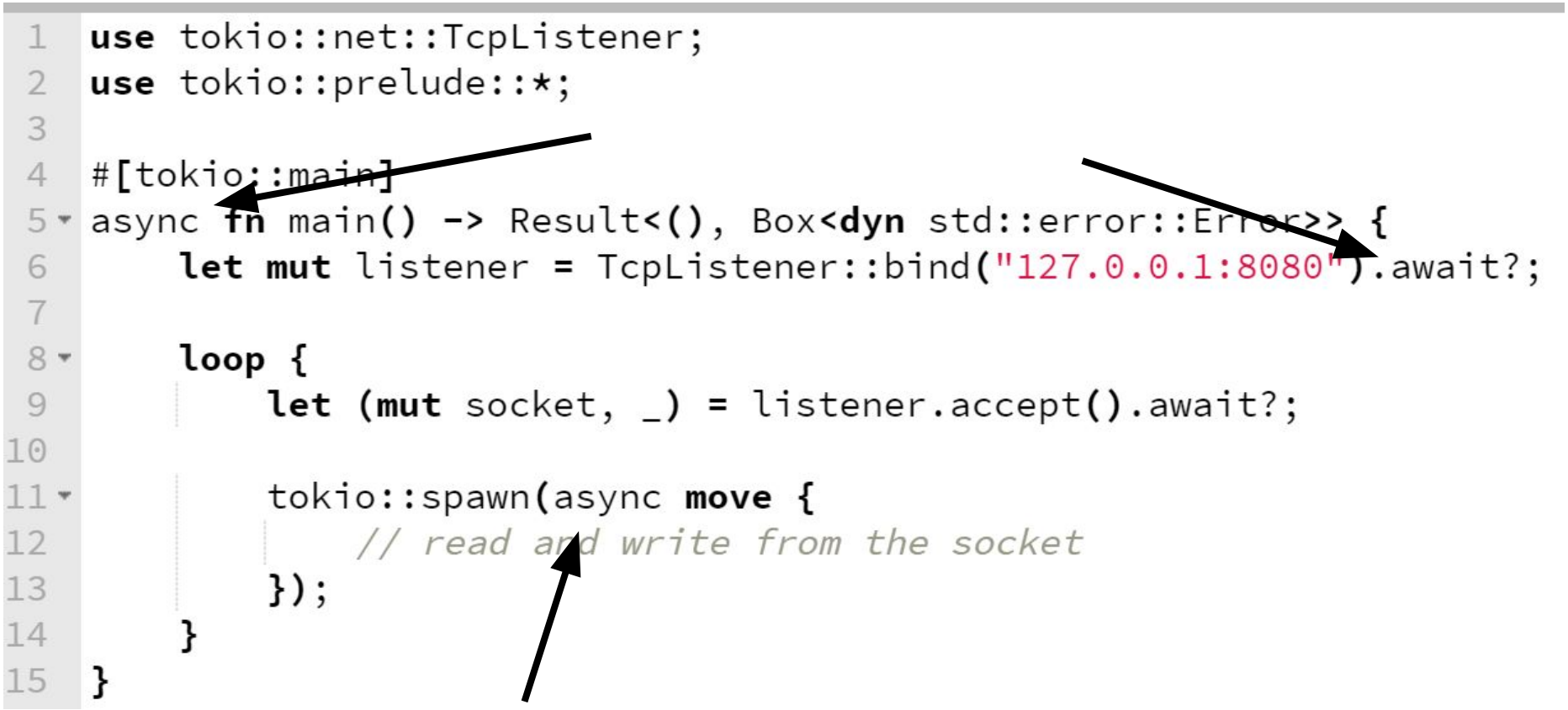
async/await and Futures

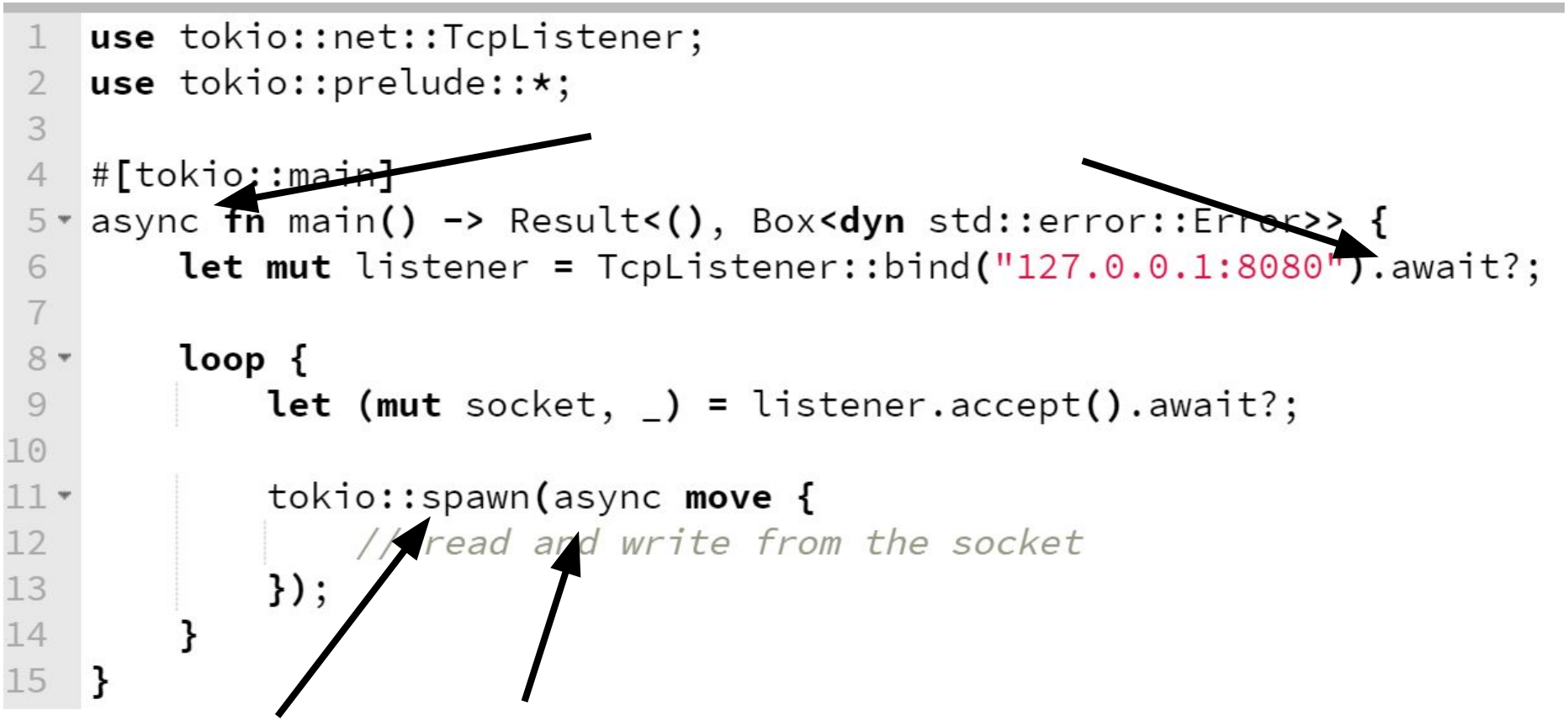
```rust
use tokio::net::TcpListener;
use tokio::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            // read and write from the socket
        });
    }
}
```



```rust
use tokio::net::TcpListener;
use tokio::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            // read and write from the socket
        });
    }
}
```


use tokio::net::TcpListener;
use tokio::prelude::*;
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
let mut listener = TcpListener::bind("127.0.0.1:8080").await?;
loop {
let (mut socket, _) = listener.accept().await?;
tokio::spawn(async move {
// read and write from the socket
});
}
}


use tokio::net::TcpListener;
use tokio::prelude::*;
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
let mut listener = TcpListener::bind("127.0.0.1:8080").await?;
loop {
let (mut socket, _) = listener.accept().await?;
tokio::spawn(async move {
// read and write from the socket
});
}
}


use tokio::net::TcpListener;
use tokio::prelude::*;
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
let mut listener = TcpListener::bind("127.0.0.1:8080").await?;
loop {
let (mut socket, _) = listener.accept().await?;
tokio::spawn(async move {
// read and write from the socket
});
}
}

Async/await is simpler syntax for Futures

Async/await is simpler syntax for Futures*

A Future represents a value that will exist sometime in the future

Let’s build a future!

A timer future

- Mutex around a boolean
- Spins up a new thread that sleeps for some amount of time
- When the thread wakes up, it sets the boolean to true and ‘wakes up’ the future
- Calls to poll check the boolean to see if we’re done

```rust
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

```rust
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

```rust
pub struct TimerFuture {
    shared_state: Arc<Mutex<SharedState>>,
}

struct SharedState {
    /// Whether or not the sleep time has elapsed
    completed: bool,

    /// the "waker" to wake up the future
    waker: Option<Waker>,
}
```

```rust
impl Future for TimerFuture {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let mut shared_state = self.shared_state.lock().unwrap();

        if shared_state.completed {
            Poll::Ready(())
        } else {
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

```rust
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();

            shared_state.completed = true;

            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Four rules

For using async/await

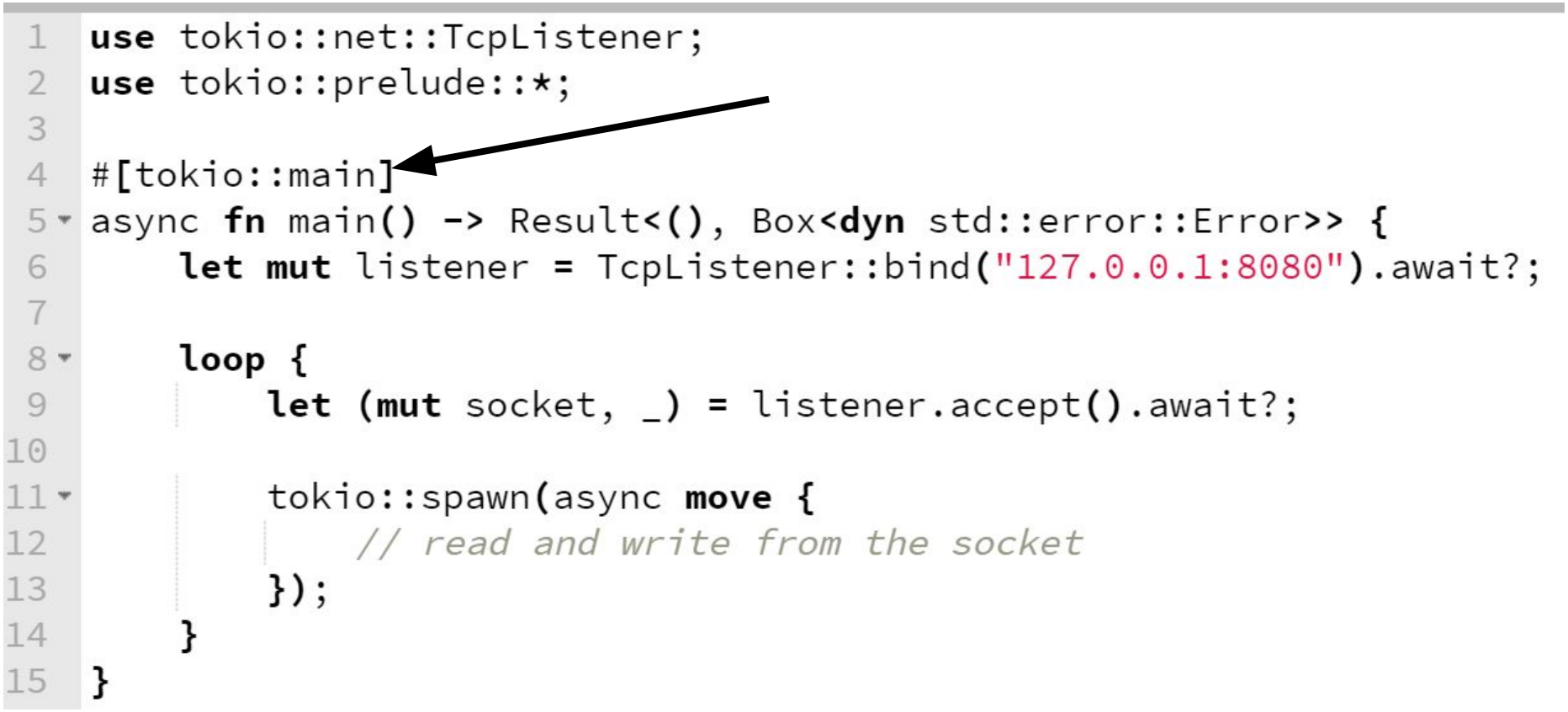
```
async fn foo(s: String) -> i32 {
    // …
}
```

```
fn foo(s: String) -> impl Future<Output=i32>  {
    // …
}
```

If you have a `Future<Output=i32>` and you want an `i32`, use `.await` on it

You can only .await inside of an async fn or block

To start executing a `Future`, you pass it to an executor



```rust
use tokio::net::TcpListener;
use tokio::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            // read and write from the socket
        });
    }
}
```

```
async fn i_sleep() {
    Delay::new(Duration::from_secs(5)).await;
}

async fn how_long() {
    let x = i_sleep();
    let y = i_sleep();

    x.await;
    y.await;
}
```

```rust
async fn i_sleep() {
    Delay::new(Duration::from_secs(5)).await;
}

async fn how_long() {
    let x = i_sleep();
    let y = i_sleep();

    future::join(x, y).await;
}
```

Generators aka stackless coroutines

Generators are *not stable*

... yet

```rust
let mut gen = || {
    let xs = vec![1, 2, 3];

    let mut sum = 0;

    for x in xs {
        sum += x;
        yield sum;
    }
};
```

```
let xs = vec![1, 2, 3];
let mut gen = || {
    let mut sum = 0;
    for x in xs.iter() {  // iter0
        sum += x;
        yield sum;  // Suspend0
    }
    for x in xs.iter().rev() {  // iter1
        sum -= x;
        yield sum;  // Suspend1
    }
};
```

```
enum SumGenerator {
    Unresumed {
        xs: Vec<i32>,
    },
    Suspend0 {
        xs: Vec<i32>,
        iter0: Iter<'self, i32>,
        sum: i32,
    },
    Suspend1 {
        xs: Vec<i32>,
        iter1: Iter<'self, i32>,
        sum: i32,
    },
    Returned,
}
```

Futures need to have poll() called over and over until a value is produced

Generators let you call yield over and over to get values

async/await is a simpler syntax for a generator that implements the Future trait

Tasks, Executors, & Reactors

“The event loop”

Task: a unit of work to execute, a chain of Futures

Executor: schedules tasks

Reactor: notifies the executor that tasks are ready to execute

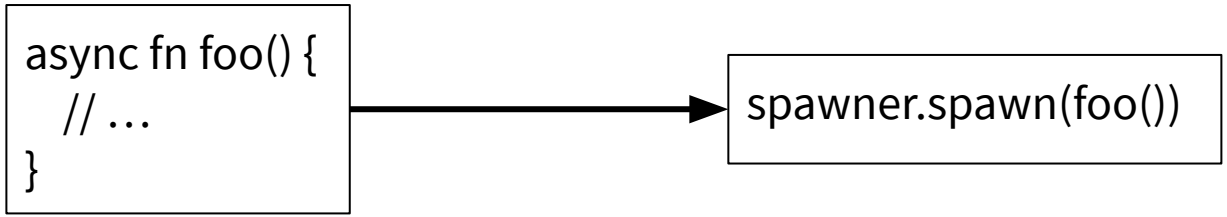
Executor calls poll, and provides a context

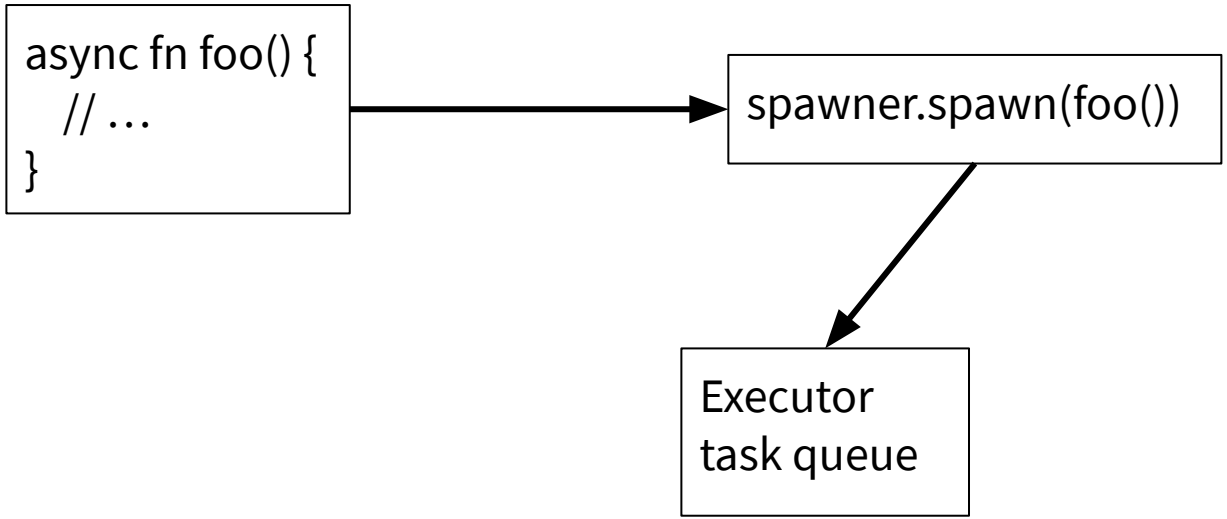
```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

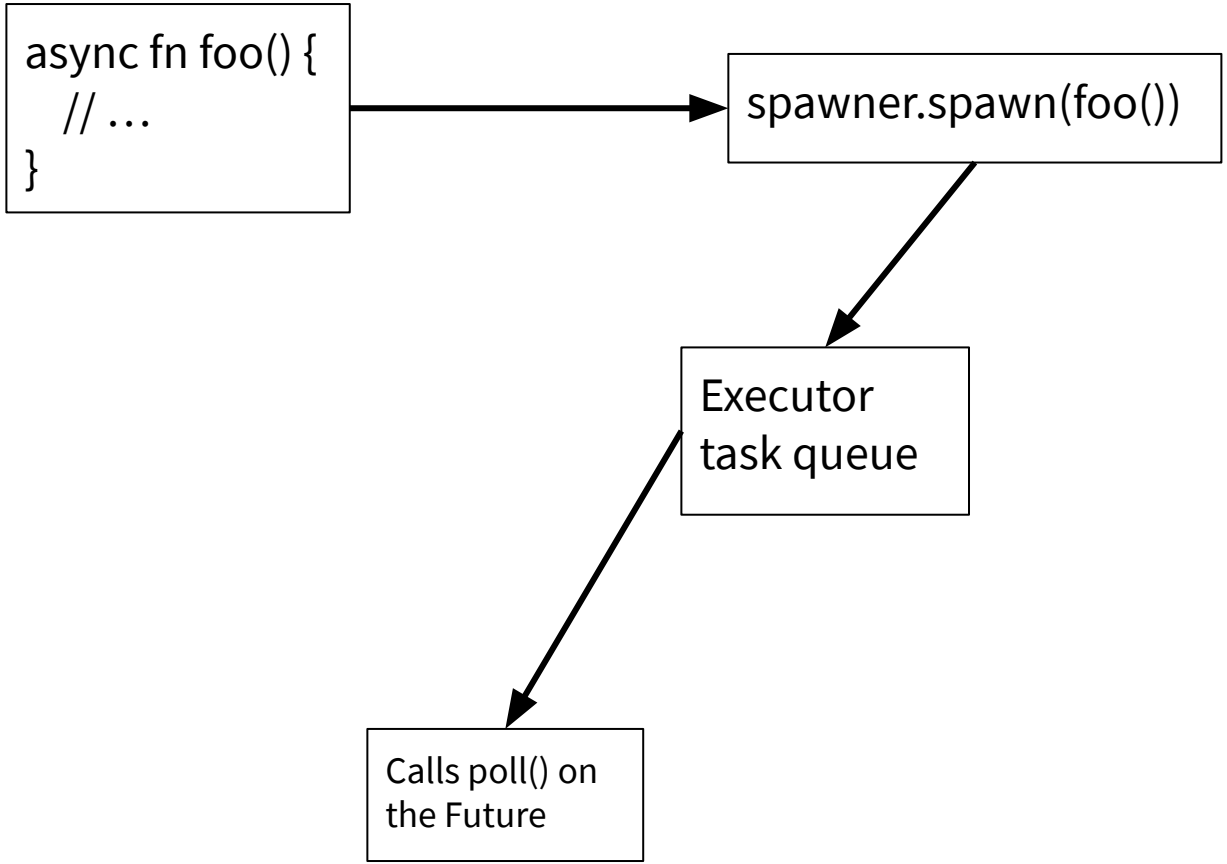
Interface to the reactor

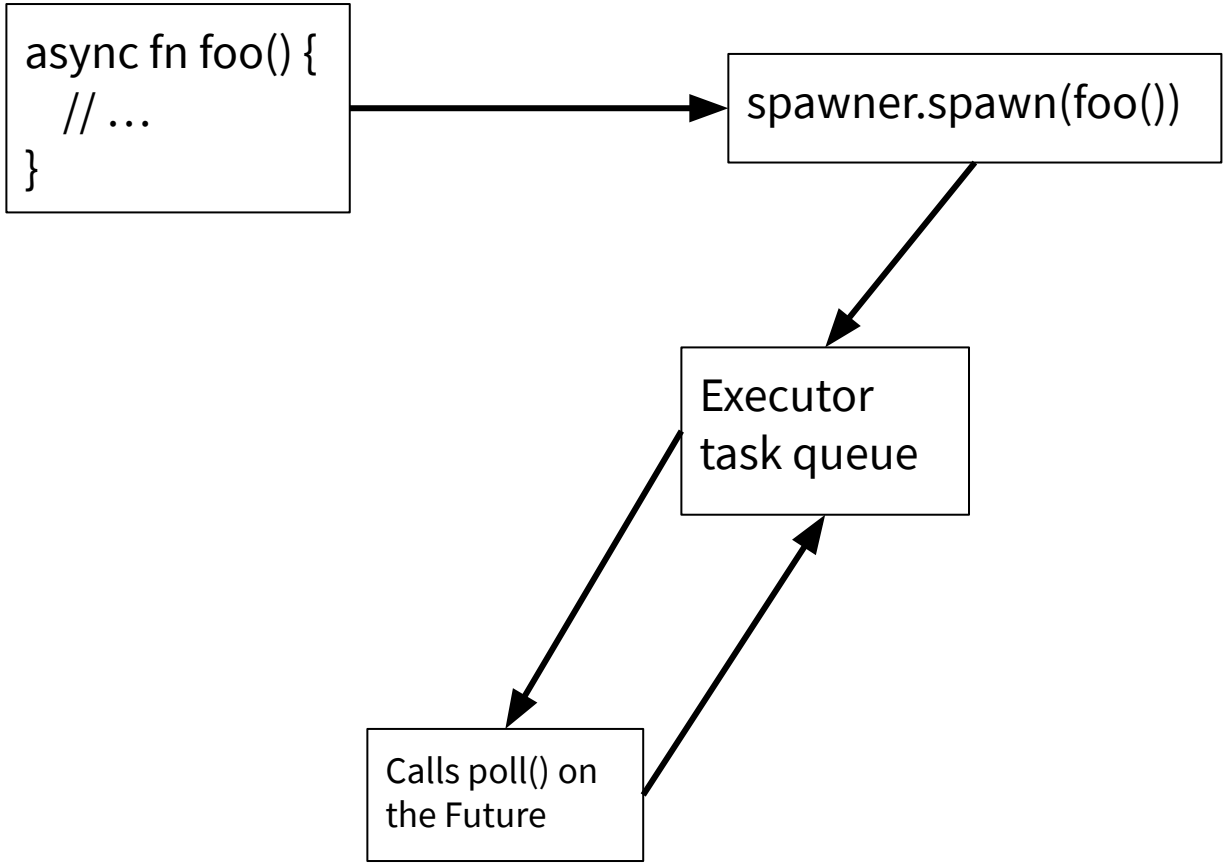
Let’s build an executor!

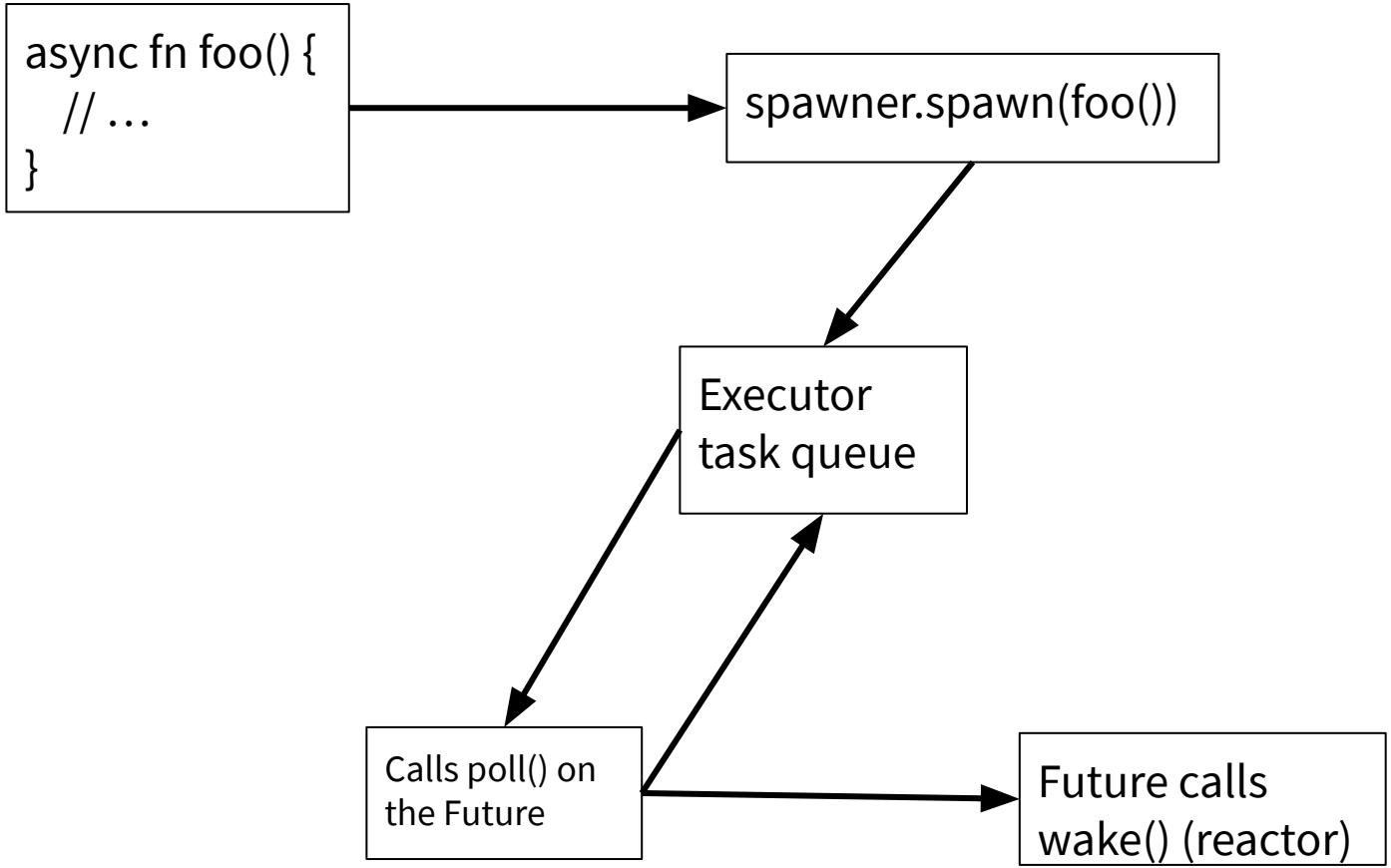
```
async fn foo() {
    // …
}
```

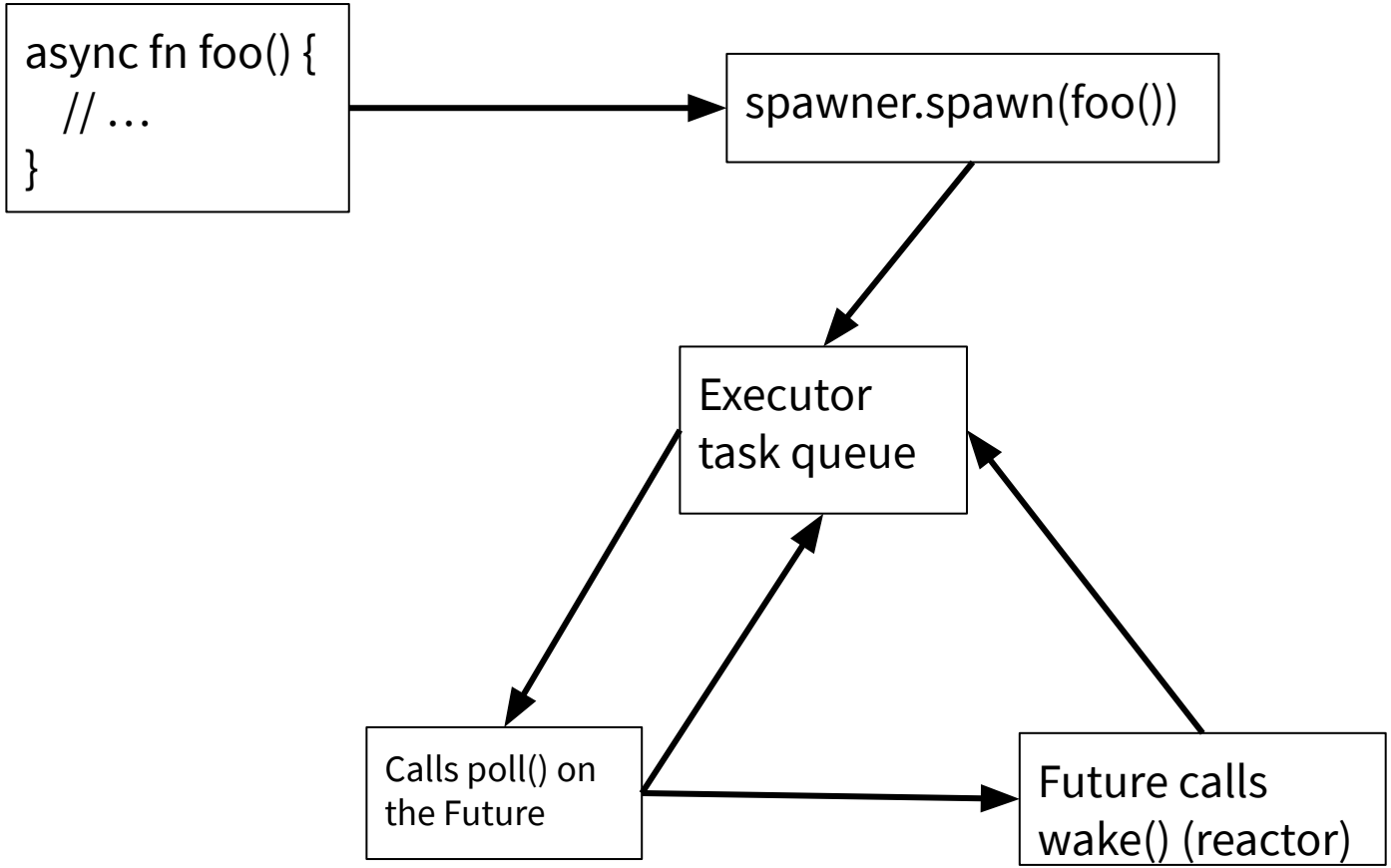

async fn foo() {
// …
}
spawner.spawn(foo())


async fn foo() {
// …
}
spawner.spawn(foo())
Executor
task queue


async fn foo() {
// …
}
spawner.spawn(foo())
Executor
task queue
Calls poll() on
the Future


async fn foo() {
// …
}
spawner.spawn(foo())
Executor
task queue
Calls poll() on
the Future


async fn foo() {
// …
}
spawner.spawn(foo())
Executor
task queue
Calls poll() on
the Future
Future calls
wake() (reactor)


async fn foo() {
// …
}
spawner.spawn(foo())
Executor
task queue
Calls poll() on
the Future
Future calls
wake() (reactor)

```rust
/// Task executor that receives tasks off of a channel and runs them.
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

/// A future that can reschedule itself to be polled by an `Executor`.
struct Task {
    /// In-progress future that should be pushed to completion.
    future: Mutex<Option<BoxFuture<'static, ()>>>,

    /// Handle to place the task itself back onto the task queue.
    task_sender: SyncSender<Arc<Task>>,
}
```

```rust
/// `Spawner` spawns new futures onto the task channel.
#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}


fn new_executor_and_spawner() -> (Executor, Spawner) {
    // Maximum number of tasks to allow queueing in the channel at once.
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender})
}
```

```
impl Spawner {
    fn spawn(&self, future: impl Future<Output = ()> + 'static + Send) {
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender.send(task).expect("too many tasks queued");
    }
}
```

```rust
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        let cloned = arc_self.clone();

        arc_self.task_sender.send(cloned).expect("too many tasks queued");
    }
}
```

```rust
impl Executor {
    fn run(&self) {
        while let Ok(task) = self.ready_queue.recv() {
            // Take the future, and if it has not yet completed (is still Some),
            // poll it in an attempt to complete it.
            let mut future_slot = task.future.lock().unwrap();

            if let Some(mut future) = future_slot.take() {
                // Create a `LocalWaker` from the task itself
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);

                if let Poll::Pending = future.as_mut().poll(context) {
                    // We're not done processing the future, so put it
                    // back in its task to be run again in the future.
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

```rust
fn main() {
    let (executor, spawner) = new_executor_and_spawner();

    // Spawn a task to print before and after waiting on a timer.
    spawner.spawn(async {
        println!("howdy!");

        // Wait for our timer future to complete after two seconds.
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("done!");
    });

    // Drop the spawner so that our executor knows it is finished and won't
    // receive more incoming tasks to run.
    drop(spawner);

    // Run the executor until the task queue is empty.
    // This will print "howdy!", pause, and then print "done!".
    executor.run();
}
```

A quick aside about Pin<P>

Before a future starts executing, we need to be able to move it around in memory.

(For example, to create a task out of it, we need to move it to the heap)

Once a future starts executing, it **must not** move in memory.

(otherwise, borrows in the body of the future would become invalid)

When you turn some sort of pointer type into a Pin<P>, you're promising that what the pointer to will no longer move.

Box<T> turns into Pin<Box<T>>

There's an extra trait, "Unpin", that says "I don't care about this", similar to how Copy says "I don't care about move semantics.

Let’s build a reactor!

(We’re not gonna build a reactor)

(We technically *did* build a reactor)

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        let cloned = arc_self.clone();

        arc_self.task_sender.send(cloned).expect("too many tasks queued");
    }
}
```

Bonus round: async fn in traits

```
async fn foo() -> i32 {
    // ...
}

trait Foo {
    async fn foo() -> i32;
}
```

A function is only one function

A trait is implemented for many types, and so is many functions

```
trait Foo {
    async fn foo() -> i32;
}
```

```
trait Foo {
    async fn foo() -> i32;
}
```

```
trait Foo {
    type FooReturn: Future<Output=i32>;

    async fn foo() -> Self::FooReturn;
}
```

It gets way more complicated

```
trait Database {
    async fn get_user(&self) -> User;
}
```

```rust
trait Database {
    async fn get_user(&self) -> User;
}
```

```rust
impl MyDatabase {
    fn get_user(&self) -> impl Future<Output = User> + '_;
}
```

```rust
trait Database {
    type GetUser<'s>: Future<Output = User> + 's;

    fn get_user(&self) -> Self::GetUser<'_>;
}
```

It gets way *way way* more complicated

async-trait 0.1.17

Documentation Repository Dependent crates

Cargo.toml

```
async-trait = "0.1.17"
```

Async trait methods

build passing crates.io v0.1.17 api rustdoc

```
#[async_trait]
trait Foo {
    async fn foo() -> i32;
}
```

```rust
#[async_trait]
trait Foo {
    async fn foo() -> i32;
}
```

```rust
trait Foo {
    fn foo() -> Pin<Box<dyn Future<Output = i32> + Send>;
}
```

Thanks!

@steveklabnik

- https://rust-lang.github.io/async-book
- https://tmandry.gitlab.io/blog/posts/optimizing-await-1/
- https://smallcultfollowing.com/babysteps/blog/2019/10/26/async-fn-in-traits-are-hard/