# Concurrency & Parallelism 2

a little bit of fancy stuff

**Jack Duvall**

Carnegie Mellon University

# Outline

# Outline

# What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

# What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

# What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

# What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads

# What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads
- Event driven (event loops and callbacks)

# What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads
- Event driven (event loops and callbacks)
- Actor based

# What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads
- Event driven (event loops and callbacks)
- Actor based
- **Coroutines**

# Drawbacks Of Threads/Multiprocessing

# Drawbacks Of Threads/Multiprocessing

- Threads/Processes are managed by the OS, expensive to spawn a bunch

# Drawbacks Of Threads/Multiprocessing

- Threads/Processes are managed by the OS, expensive to spawn a bunch
- Inter-Process Communication (IPC) very slow (for processes)

# Drawbacks Of Threads/Multiprocessing

- Threads/Processes are managed by the OS, expensive to spawn a bunch
- Inter-Process Communication (IPC) very slow (for processes)
- Significantly change structure of code

# Drawbacks Of Threads/Multiprocessing

- Threads/Processes are managed by the OS, expensive to spawn a bunch
- Inter-Process Communication (IPC) very slow (for processes)
- Significantly change structure of code
  - Building around race conditions in threads

# Drawbacks Of Threads/Multiprocessing

- Threads/Processes are managed by the OS, expensive to spawn a bunch
- Inter-Process Communication (IPC) very slow (for processes)
- Significantly change structure of code
  - Building around race conditions in threads
  - Explicitly joining threads/processes

# Drawbacks Of Threads/Multiprocessing

- Threads/Processes are managed by the OS, expensive to spawn a bunch
- Inter-Process Communication (IPC) very slow (for processes)
- Significantly change structure of code
  - Building around race conditions in threads
  - Explicitly joining threads/processes
- Still useful for many applications! Just different applicability

# Drawbacks of Callbacks

Async in JavaScript (pre-Promises)

```
$.ajax("https://example.com/thingy").then(function(r){
    // do something with r.status and r.data
});
```

# Drawbacks of Callbacks

Async in JavaScript (pre-Promises)

```
$.ajax("https://example.com/thingy").then(function(r){
    // do something with r.status and r.data
});
```

- Can be verbose, especially when nesting

# Drawbacks of Callbacks

Async in JavaScript (pre-Promises)

```
$.ajax("https://example.com/thingy").then(function(r){
    // do something with r.status and r.data
});
```

- Can be verbose, especially when nesting
- Loops/other control flow is tricky or done outside the core language

# Why Use Async For Concurrency?

# Why Use Async For Concurrency?

- **Built into the language**: most code doesn't need significant changes to become async-compatible, and async code still looks like normal code

# Why Use Async For Concurrency?

- **Built into the language**: most code doesn't need significant changes to become async-compatible, and async code still looks like normal code
- **Zero-cost**: no heap allocations or dynamic dispatch unless specified in the type

# Why Use Async For Concurrency?

- **Built into the language**: most code doesn't need significant changes to become async-compatible, and async code still looks like normal code
- **Zero-cost**: no heap allocations or dynamic dispatch unless specified in the type
- **Flexible choice of runtime**: single- and multi-threaded implementations exist for different platforms

# Clean Async: Network Protocol

```
async fn heartbeat(client: ClientConn) -> Result<(), ConnError> {
    loop {
        client.send("ping").await?;
        if client.recv().await? != "pong" { break; }
    }
    Ok(())
}
```

# Clean Async: Parallel Jobs

Playground Link:

```
#[async_recursion::async_recursion]
async fn reduce_max<T: Ord + Sync>(arr: &[T], lo: usize, hi: usize)
-> &T {
    if lo == hi { return &arr[lo]; }
    let mi = lo + (hi - lo) / 2;
    let fut_lo = reduce_max(arr, lo, mi);
    let fut_hi = reduce_max(arr, mi+1, hi);
    let (res_lo, res_hi) = futures::join!(fut_lo, fut_hi);
    match res_lo.cmp(res_hi) {
        std::cmp::Ordering::Less => res_hi,
        _ => res_lo,
    }
}
```

# Clean Async: Multi-Client Server

```
let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
loop {
    let (socket, _) = listener.accept().await.unwrap();
    tokio::spawn(async move {
        process(socket).await;
    });
}
```

# Drawbacks of Async

# Drawbacks of Async

- Async is cooperative

# Drawbacks of Async

- Async is cooperative
    - Doing CPU-heavy work may block other coroutines from running

# Drawbacks of Async

- Async is cooperative
  - Doing CPU-heavy work may block other coroutines from running
  - Not yielding via `await` will block other coroutines

# Drawbacks of Async

- Async is cooperative
  - Doing CPU-heavy work may block other coroutines from running
  - Not yielding via `await` will block other coroutines
  - Ideal for applications where busy time is minimal, and most time would be spent waiting for the OS if all coroutines ran on a single thread

# Drawbacks of Async

- Async is cooperative
  - Doing CPU-heavy work may block other coroutines from running
  - Not yielding via `await` will block other coroutines
  - Ideal for applications where busy time is minimal, and most time would be spent waiting for the OS if all coroutines ran on a single thread
- Lots of tricky type and trait errors!

# Outline

# Under The Hood Of Async

```
trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Self::Output>;
}1

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

# Under The Hood Of Async

```
trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Self::Output>;
}1

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

- If a type implements `Future`, you can use the `await` syntax with it!

# Under The Hood Of Async

```
trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Self::Output>;
}1

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

- If a type implements `Future`, you can use the `await` syntax with it!
- `Pin`, `Context`: we'll get to these later

# Even Further Under The Hood

How does Rust even turn an `async fn` into a `Future`?

# Even Further Under The Hood

How does Rust even turn an `async fn` into a `Future`?

- State Machines!

# Even Further Under The Hood

How does Rust even turn an `async fn` into a `Future`?

- State Machines!
- Each time you `await` another future, all the variables that could be used in later execution are saved into the current state

# Example Async Function: Sugared

```
async fn serve(addr: String){
    let client = get_client(addr).await;
    heartbeat(client).await.unwrap();
}
```

# Example Async Function: Desugared (1/3)

```
enum Serve {
    State0(String),     // Initial argument
    State1(GetClient), // First internal future
    State2(Heartbeat), // Second internal future
    Terminated,         // Completed state
}
```

# Example Async Function: Desugared (2/3)

```
impl Future for Serve {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, cx: &mut Context) -> Poll<()> {
        use Serve::*; // for convenience
        loop { match *self { /* next slide */ } }
    }
}
```

# Example Async Function: Desugared (3/3)

```
State0(addr) => { *self = State1(get_client(addr)); }
State1(ref mut get_client) => match Pin::new(get_client).poll(cx) {
    Poll::Ready(client) => { *self = State2(heartbeat(client)); }
    Poll::Pending => { return Poll::Pending; }
}
State2(ref mut heartbeat) => match Pin::new(heartbeat).poll(cx) {
    Poll::Ready(()) => {
        *self = Terminated;
        return Poll::Ready(());
    }
    Poll::Pending => { return Poll::Pending; }
}
Terminated => { unreachable!("Terminated future cannot be polled"); }
```

# This ”Desugaring” Is Approximate!

# This ”Desugaring” Is Approximate!

- I have no idea what the compiler actually does

# This ”Desugaring” Is Approximate!

- I have no idea what the compiler actually does
- Not sure if anyone except the compiler team truly does

# This ”Desugaring” Is Approximate!

- I have no idea what the compiler actually does
- Not sure if anyone except the compiler team truly does
- We don’t need to worry about the details, it’s all done for us :)

# Outline

# What Does `Pin` Mean?

# What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`

# What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
  - This means `P` is a "pointer-like" type; `Deref` and `DerefMut` control what happens when you do `*p`

# What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
  - This means `P` is a "pointer-like" type; `Deref` and `DerefMut` control what happens when you do `*p`
  - Examples for `P`: `&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`

# What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
  - This means `P` is a "pointer-like" type; `Deref` and `DerefMut` control what happens when you do `*p`
  - Examples for `P`: `&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`
- Guarantee: "In a `Pin<P>`, the value pointed to by `P` will have a stable location in memory, and is only deallocated when `P` is dropped"

# What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
  - This means `P` is a "pointer-like" type; `Deref` and `DerefMut` control what happens when you do `*p`
  - Examples for `P`: `&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`
- Guarantee: "In a `Pin<P>`, the value pointed to by `P` will have a stable location in memory, and is only deallocated when `P` is dropped"
- How it's enforced: `Pin<&mut T>` is *not* a `&mut T`! This limits what you can do with it

# Why Do We Need Pin?

Remember how futures store local variables as states: what if references to these variables are passed to other futures?

```
async fn incr(x: &mut i32) {
    x += 1;
}
async fn main() {
    let mut x = 0;
    incr(&mut x).await;
    assert_eq!(x, 1);
}
```

# Why Do We Need Pin?

```
/// Approximate states for `async fn main()`:
enum Main {
    State0,             // Initial state
    State1 {
        x: i32,         // Local variable
        incr: Incr,     // Child, has a reference to this local!
    },
    Terminated,
}1
```

# Why Do We Need `Pin`?

- In order for `incr`'s reference to `x` to stay valid, the address of the `Main` value must not change.

# Why Do We Need `Pin`?

- In order for `incr`'s reference to `x` to stay valid, the address of the `Main` value must not change.
  - How could a local variable change address? If it gets moved (which can be just a `memcpy`)

# Why Do We Need `Pin`?

- In order for `incr`'s reference to `x` to stay valid, the address of the `Main` value must not change.
  - How could a local variable change address? If it gets moved (which can be just a `memcpy`)
- `Pin` enforces this exactly! Value pointed to by `Pin<P>` guaranteed to have a stable address in memory

# Another Example Of `Pin` Doing Something

```rust
fn take1(v: &mut Option<String>) -> String {
    v.take()
}
fn take2(v: Pin<&mut Option<String>>) -> String {
    v.take() // compiler error!
}
```

# Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

# Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

- Compiler can't guarantee data will stay pinned (that's what the type is for!)

# Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

- Compiler can't guarantee data will stay pinned (that's what the type is for!)
- Have to prove safety for yourself, or

# Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

- Compiler can't guarantee data will stay pinned (that's what the type is for!)
- Have to prove safety for yourself, or
- Usually use convenience wrappers (`Box::pin`, `pin_utils::pin_mut!`) that already have proven safety

# The `Unpin` Trait

# The `Unpin` Trait

- Sometimes, you know it's OK for a value to not have a stable location in memory, because it **cannot** be self-referrential

# The `Unpin` Trait

- Sometimes, you know it's OK for a value to not have a stable location in memory, because it **cannot** be self-referrential
  - `bool`, `i32`, `f64`, etc.

# The `Unpin` Trait

- Sometimes, you know it's OK for a value to not have a stable location in memory, because it **cannot** be self-referrential
  - `bool`, `i32`, `f64`, etc.
- Only matters for `Pin<P>` when `<P as Deref>::Target: Unpin`, **not** for `P: Unpin` itself.

# The `Unpin` Trait

- Sometimes, you know it's OK for a value to not have a stable location in memory, because it **cannot** be self-referrential
  - `bool`, `i32`, `f64`, etc.
- Only matters for `Pin<P>` when `<P as Deref>::Target: Unpin`, **not** for `P: Unpin` itself.
- The difference is between the value being pointed to being able to move (useful), or the pointer iself being able to move (pointers are just numbers, this is useless).

# Outline

# Streams Are Iterators Polled Like Futures

# Streams Are Iterators Polled Like Futures

- "If `Future<Output=T>` is the async version of a `T`, then `Stream<Item=T>` is the async version of `Iterator<Item=T>`"

# Streams Are Iterators Polled Like Futures

- "If `Future<Output=T>` is the async version of a `T`, then `Stream<Item=T>` is the async version of `Iterator<Item=T>`"
- Main difference from a regular `Future`: can be polled for multiple items instead of just one

# Streams Are Iterators Polled Like Futures

- "If `Future<Output=T>` is the async version of a `T`, then `Stream<Item=T>` is the async version of `Iterator<Item=T>`"
- Main difference from a regular `Future`: can be polled for multiple items instead of just one
- Not part of standard library, but de-facto standard `futures` crate which is used everywhere in the ecosystem

1

# `Stream` Trait Definition

```
pub trait Stream {
    type Item;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Option<Self::Item>>;
    fn size_hint(&self) -> (usize, Option<usize>) { ... }
}
```

# `Stream` Trait Definition

```
pub trait Stream {
    type Item;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Option<Self::Item>>;
    fn size_hint(&self) -> (usize, Option<usize>) { ... }
}
```

Wait, doesn't `Iterator` have a lot more associated methods than this??

# `futures` extension traits

# `futures` extension traits

- There are `FutureExt` and `StreamExt` traits, implemented for anything implementing `Future` and `Stream`. These take the place of default associated functions

# `futures` extension traits

- There are `FutureExt` and `StreamExt` traits, implemented for anything implementing `Future` and `Stream`. These take the place of default associated functions
  - Useful composition functions like `.map()`, utilities like `.boxed()`, etc.

# `futures` extension traits

- There are `FutureExt` and `StreamExt` traits, implemented for anything implementing `Future` and `Stream`. These take the place of default associated functions
  - Useful composition functions like `.map()`, utilities like `.boxed()`, etc.
  - Default impls cause vtable to blow up in size, bad for things that are often boxed like `Future`s and `Stream`s.

# `futures` extension traits

- There are `FutureExt` and `StreamExt` traits, implemented for anything implementing `Future` and `Stream`. These take the place of default associated functions
  - Useful composition functions like `.map()`, utilities like `.boxed()`, etc.
  - Default impls cause vtable to blow up in size, bad for things that are often boxed like `Future`s and `Stream`s.
- `futures` also provides `join!`, `pin_mut!`, and other useful macros

# `futures` extension traits

- There are `FutureExt` and `StreamExt` traits, implemented for anything implementing `Future` and `Stream`. These take the place of default associated functions
  - Useful composition functions like `.map()`, utilities like `.boxed()`, etc.
  - Default impls cause vtable to blow up in size, bad for things that are often boxed like `Future`s and `Stream`s.
- `futures` also provides `join!`, `pin_mut!`, and other useful macros
- By default, doesn't actually have any way to run futures! (There's a feature for that, but usually you'll use a third-party crate)

1

# Outline

# How Do We Run Futures?

(Content taken from Tokio's Async Tutorital)

Recall: `Future`s just have a poll method. So let's call that in a loop! This actually just works!

```
fn run(mut fut: impl Future<Output = ()>, cx: &mut Context) {
    pin_mut!(fut);
    loop {
        if let Poll::Ready(()) = fut.poll(cx) {
            break;
        }
    }
}
```

# Ok But How Do We Actually Run Futures?

Use a pre-built Async Reactor like the ones in `tokio`, `futures::executor`, or `async-std`

```
#[tokio::main]
async fn main() {
    // Now you can call async functions in here!
}
```

# Well How Do Those Work?

Back to low-level stuff >:)

# Well How Do Those Work?

Back to low-level stuff >:)

- Polling in a loop considered harmful

# Well How Do Those Work?

Back to low-level stuff >:)

- Polling in a loop considered harmful
  - Wastes CPU cycles, busy loops in general ”make the fans turn on”

# Well How Do Those Work?

Back to low-level stuff >:)

- Polling in a loop considered harmful
  - Wastes CPU cycles, busy loops in general "make the fans turn on"
- Ideally, if a polling a future gives you `Poll::Pending`, you'd only poll it again when it's likely to return `Poll::Ready`

# Well How Do Those Work?

Back to low-level stuff >:)

- Polling in a loop considered harmful
  - Wastes CPU cycles, busy loops in general "make the fans turn on"
- Ideally, if a polling a future gives you `Poll::Pending`, you'd only poll it again when it's likely to return `Poll::Ready`
- How do we know when a future would be likely to return `Poll::Ready`? `Waker`s!

# Wakers

# `Waker`s

- Main use: calling `.wake()` on any `Waker` derived from the original should signal the async reactor to poll the `Future` again.

# `Waker`s

- Main use: calling `.wake()` on any `Waker` derived from the original should signal the async reactor to poll the `Future` again.
- Remember the `Context` that got passed in to the `Future`'s `poll()` function? Literally its only job is to hold a `Waker`!

# `Waker`s

- Main use: calling `.wake()` on any `Waker` derived from the original should signal the async reactor to poll the `Future` again.
- Remember the `Context` that got passed in to the `Future`'s `poll()` function? Literally its only job is to hold a `Waker`!
- `Waker: Clone + Send + Sync + Unpin` so you can basically do whatever you want with them

# `Future` Example Using A `Waker`

```rust
impl Future for Delay {
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        if Instant::now() >= self.when { Poll::Ready(()) } else {
            let waker = cx.waker().clone();
            let when = self.when;
            thread::spawn(move || {
                let now = Instant::now();
                if now < when { thread::sleep(when - now); }
                waker.wake();
            });
            Poll::Pending
        }
    }
}
```

# Recap

# Recap

- Types implementing `Future` must be `.await`-ed

# Recap

- Types implementing `Future` must be `.await`-ed
- Use `async fn` to make a function-like future, letting you use `.await` inside

# Recap

- Types implementing `Future` must be `.await`-ed
- Use `async fn` to make a function-like future, letting you use `.await` inside
- Use an async runtime like `tokio` to run your top-level `async fn main()`

# Recap

- Types implementing `Future` must be `.await`-ed
- Use `async fn` to make a function-like future, letting you use `.await` inside
- Use an async runtime like `tokio` to run your top-level `async fn main()`
- Use the `futures` crate for lots of good utilities

# Recap

- Types implementing `Future` must be `.await`-ed
- Use `async fn` to make a function-like future, letting you use `.await` inside
- Use an async runtime like `tokio` to run your top-level `async fn main()`
- Use the `futures` crate for lots of good utilities
- This barely scratches the surface! Async is big, lots of libraries to explore, have fun with it!

# Homework

Work on the final

# Homework

Work on the final  Ask us anything!

# Outline

# Why Would You Want To Do This?

# Why Would You Want To Do This?

(Recall: sync code can be made async with `tokio::spawn_blocking`)

# Why Would You Want To Do This?

(Recall: sync code can be made async with `tokio::spawn_blocking`)

- Having your top-level function be `async` isn't the best, sometimes you want to architecture your own event loop for GUI things

# Why Would You Want To Do This?

(Recall: sync code can be made async with `tokio::spawn_blocking`)

- Having your top-level function be `async` isn't the best, sometimes you want to architecture your own event loop for GUI things

Before, we just used the `#[tokio::main]` macro. What does that expand do/can we do it ourselves?

# Doing What #[tokio::main] Does

```
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            println!("Hello world");
        })
}
```

# Manual Expansion Gives More Power

# Manual Expansion Gives More Power

- Change parameters of the runtime.

# Manual Expansion Gives More Power

- Change parameters of the runtime.
- Spawn multiple futures onto runtime at once, without `join!`

# Manual Expansion Gives More Power

- Change parameters of the runtime.
- Spawn multiple futures onto runtime at once, without `join!`
- Run futures ”in background”, while running other sync code

# Manual Expansion Gives More Power

- Change parameters of the runtime.
- Spawn multiple futures onto runtime at once, without `join!`
- Run futures ”in background”, while running other sync code

See The Tokio Docs for lots of code examples

# You Can't Have `async fn` In Traits (right now)

```
trait Webserver {
    async fn handle(&self, r: Request) -> Response;
}
```

Too bad Rust doesn't like this... Why?

# You Can't Have `async fn` In Traits (right now)

```
trait Webserver {
    async fn handle(&self, r: Request) -> Response;
}
```

Too bad Rust doesn't like this... Why?

- Short Answer: `async fn` only guarantees a trait, not a type

# You Can't Have `async fn` In Traits (right now)

```
trait Webserver {
    async fn handle(&self, r: Request) -> Response;
}
```

Too bad Rust doesn't like this... Why?

- Short Answer: `async fn` only guarantees a trait, not a type
- Long Answer: mostly stolen from Niko Matsakis' Blog

# async fn Is Syntatic Sugar For This

```
trait Webserver {
    fn handle(&self, r: Request) ->
        impl Future<Output = Response> + '_;
}
```

…roughly speaking, that is

# It Gets Funkier

```
trait Webserver {
    type HandleFuture<'a>: Future<Output = Response> + 'a;
    fn handle(&'a self, r: Request) -> Self::HandleFuture<'a>;
}
```

# It Gets Funkier

```
trait Webserver {
    type HandleFuture<'a>: Future<Output = Response> + 'a;
    fn handle(&'a self, r: Request) -> Self::HandleFuture<'a>;
}
```

This is a “Generic Associated Type” (that is, generic over lifetimes, not types). This is supported in Rust now, but wasn’t for a long time.

# More Unresolved Questions

What if you wanted to constrain futures returned by an implementation?

```
fn launch_on_multiple_threads<W>(webserver: W)
where for<'a> W::HandleFuture<'a>: Send
{
    // `Send` lets us share futures returned by
    // `webserver.handle(r)` between threads
}
```

# More Unresolved Questions

What if you wanted to constrain futures returned by an implementation?

```
fn launch_on_multiple_threads<W>(webserver: W)
where for<'a> W::HandleFuture<'a>: Send
{
    // `Send` lets us share futures returned by
    // `webserver.handle(r)` between threads
}
```

- We needed to know the name of the associated type. Is it auto-generated? Do people always need to desugar manually?

# More Unresolved Questions

What if you wanted to constrain futures returned by an implementation?

```
fn launch_on_multiple_threads<W>(webserver: W)
where for<'a> W::HandleFuture<'a>: Send
{
    // `Send` lets us share futures returned by
    // `webserver.handle(r)` between threads
}
```

- We needed to know the name of the associated type. Is it auto-generated? Do people always need to desugar manually?
- If you use a lot of futures, there's a lot more `Send` bounds you need; is there a better way to combine them all?

# Even More Considerations

If you use regular generics, many copies of code are made. Could be better to force the use of trait objects:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        dyn Future<Output = Response> + '_;
}
```

# Even More Considerations

If you use regular generics, many copies of code are made. Could be better to force the use of trait objects:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        dyn Future<Output = Response> + '_;
}
```

New problem: now the return type isn't `Sized` (don't know the size at compile time), so we can't generate code!

# Even More Considerations

If you use regular generics, many copies of code are made. Could be better to force the use of trait objects:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        dyn Future<Output = Response> + '_;
}
```

New problem: now the return type isn't `Sized` (don't know the size at compile time), so we can't generate code! Need a wrapper, but how to choose between `Box`, `Arc`, others?

# A Good Enough Solution: `async-trait` Crate

Applying *#[async_trait]* to the original trait with an `async fn` results in the following desugaring:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        Pin<Box<dyn Future<Output=Response> + Send + '_>>;
}
```

# A Good Enough Solution: `async-trait` Crate

Applying *#[async_trait]* to the original trait with an `async fn` results in the following desugaring:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        Pin<Box<dyn Future<Output=Response> + Send + '_>>;
}
```

mm delicous type + trait soup