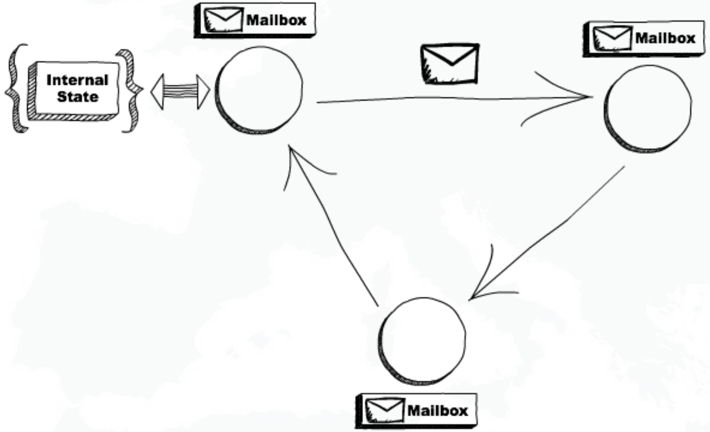
Building An Actor Model In Rust

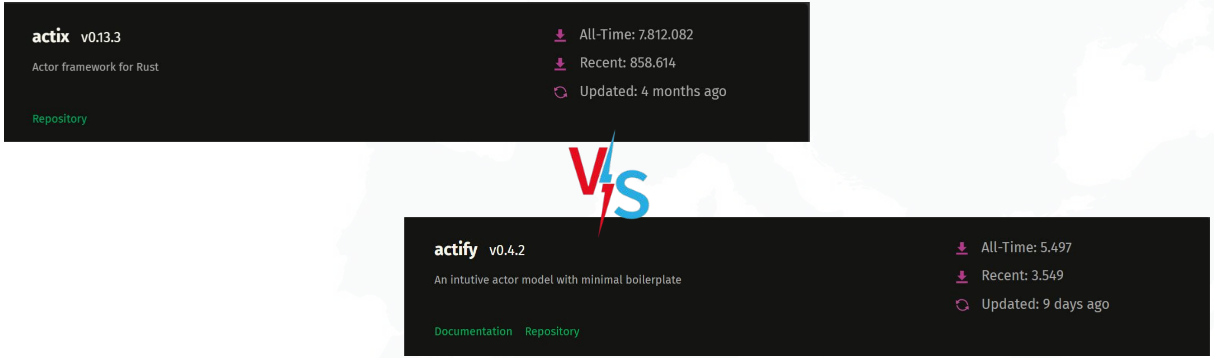
What is an Actor Model and why use it?

- Spawned, self-contained task that performs a job independently
- Communicates using messages, using the ‘address’ of each actor
- Might keep a mutable internal state
- Typically used for complex and concurrent workflows, scaling up to millions of users
- In Robotics, the modularity and concurrency is desirable (comparable to ROS)
- Examples are basically any ‘handler’, like a chatbot, payment processor, authentication, scheduler, etc…



Actor models in Rust

- An actor model in Rust is nice to deal with concurrency but requires boilerplate
- [There are a *lot* of actor framework projects on Cargo. : r/rust]
- Actix is by far the most used, but it is a **framework** not only a **library**
- Actor models in Rust tend to be *ugly*, and fail to leverage async (Tokio) well
- Some strategies towards building an actor model:
  - Using enums (see [Actors with Tokio – Alice Ryhl])
  - Using casting and downcasting to the Any type
  - Using message structs



Using enums

Defining the actor and messages

```rust
// An actor 'Calculator' is defined
1 implementation
struct Calculator {
    receiver: mpsc::Receiver<ActorMessage>,
    current: u32,
}

// An response type 'Pong' is defined
#[derive(Debug)]
1 implementation
pub struct Pong;

// An enum containing all messages is defined
0 implementations
enum ActorMessage {
    Ping(oneshot::Sender<Pong>),
    Sum(u32, oneshot::Sender<u32>),
}
```

Implementing the actor

```rust
impl Calculator {
    fn new(receiver: mpsc::Receiver<ActorMessage>) -> Self {
        Calculator {
            receiver,
            current: 0,
        }
    }

    fn handle_message(&mut self, msg: ActorMessage) {
        match msg {
            ActorMessage::Sum(value: u32, respond_to: Sender<u32>) => {
                self.current += value;
                respond_to.send(self.current).unwrap();
            }
            ActorMessage::Ping(respond_to: Sender<Pong>) => respond_to.send(Pong).unwrap(),
        }
    }
}

// This can be made into some generic setup for all actors
async fn run_my_actor(mut actor: Calculator) {
    while let Some(msg: ActorMessage) = actor.receiver.recv().await {
        actor.handle_message(msg);
    }
}
```

Building the handle manually

```rust
#[derive(Clone)]
2 implementations
pub struct MyActorHandle {
    sender: mpsc::Sender<ActorMessage>,
}

impl MyActorHandle {
    pub fn new() -> Self {
        let (sender: Sender<ActorMessage>, receiver: Receiver<ActorMessa…) = mpsc::channel(8);
        let actor: Calculator = Calculator::new(receiver);
        tokio::spawn(future: run_my_actor(actor));
        Self { sender }
    }

    pub async fn ping(&self) -> Pong {
        let (respond_to: Sender<Pong>, recv: Receiver<Pong>) = oneshot::channel();
        let msg: ActorMessage = ActorMessage::Ping(respond_to);
        self.sender.send(msg).await.unwrap();
        recv.await.unwrap()
    }

    pub async fn sum(&self, value: u32) -> u32 {
        let (respond_to: Sender<u32>, recv: Receiver<u32>) = oneshot::channel();
        let msg: ActorMessage = ActorMessage::Sum(value, respond_to);
        self.sender.send(msg).await.unwrap();
        recv.await.unwrap()
    }
}
```

Interaction with the actor through the handle

```
#[tokio::main]
▶ Run | Debug
async fn main() {
    let handle: MyActorHandle = MyActorHandle::new();
    let _pong: Pong = handle.ping().await;
    assert_eq!(handle.sum(5).await, 5);
    assert_eq!(handle.sum(10).await, 15);
}
```

It can get problematic with complex messages

```rust
// Adding a generic to a single variant requires it to always be known
0 implementations
enum ActorMessage<T: Debug> {
    Ping(oneshot::Sender<Pong>),
    Sum(u32, oneshot::Sender<u32>),
    Log(T),
}

impl MyActorHandle {
    pub async fn ping(&self) -> Pong {
        let (respond_to: Sender<Pong>, recv: Receiver<Pong>) = oneshot::channel();
        let msg: ActorMessage<{unknown}> = ActorMessage::Ping(respond_to);
        self.sender.send(msg).await.unwrap();
        recv.await.unwrap()
    }

    pub async fn sum(&self, value: u32) -> u32 {
        let (respond_to: Sender<u32>, recv: Receiver<u32>) = oneshot::channel();
        let msg: ActorMessage<{unknown}> = ActorMessage::Sum(value, respond_to);
        self.sender.send(msg).await.unwrap();
        recv.await.unwrap()
    }
```

Using dynamic dispatch with the Any trait

Method and arguments are all trait objects

```rust
type ActorMethod<T> = Box<dyn FnMut(&mut T, Box<dyn Any + Send>) -> Box<dyn Any + Send> + Send>;

// An actor 'Calculator' is defined
1 implementation
struct Calculator {
    receiver: mpsc::Receiver<(ActorMethod<Calculator>, Box<dyn Any + Send>)>,
    current: u32,
}

// An response type 'Pong' is defined
#[derive(Debug)]
1 implementation
pub struct Pong;
```

Actor implementation requires horrible runtime downcasting

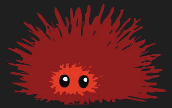
```rust
impl Calculator {
    fn new(receiver: mpsc::Receiver<(ActorMethod<Calculator>, Box<dyn Any + Send>)>) -> Self {
        Calculator {
            receiver,
            current: 0,
        }
    }

    fn ping(&mut self, args: Box<dyn Any + Send>) -> Box<dyn Any + Send> {
        let respond_to: oneshot::Sender<Pong> = *args.downcast().unwrap();
        Box::new(respond_to.send(Pong).unwrap())
    }

    fn sum(&mut self, args: Box<dyn Any + Send>) -> Box<dyn Any + Send> {
        let (value: u32, respond_to: Sender<u32>): (u32, oneshot::Sender<u32>) = *args.downcast().unwrap();
        self.current += value;
        Box::new(respond_to.send(self.current).unwrap())
    }

    fn handle_message(
        &mut self,
        msg: (ActorMethod<Calculator>, Box<dyn Any + Send>),
    ) -> Box<dyn Any + Send> {
        let (mut method: Box<dyn FnMut(&mut Calculator, …) -> … + …, args) = msg;
        (method)(self, args)
    }
} impl Calculator
```

The handle is still comparable

```
#[derive(Clone)]
2 implementations
pub struct MyActorHandle {
    sender: mpsc::Sender<(ActorMethod<Calculator>, Box<dyn Any + Send>)>,
}

impl MyActorHandle {
    pub async fn ping(&self) -> Pong {
        let (respond_to: Sender<Pong>, recv: Receiver<Pong>) = oneshot::channel();
        self.sender Sender<(Box<dyn FnMut(&mut Calculator, …) -…
            .send((Box::new(Calculator::ping), Box::new(respond_to))) impl Future<Output = Result<…, …>>
            .await Result<(), SendError<(Box<…>, …)>>
            .unwrap();
        recv.await.unwrap()
    }

    pub async fn sum(&self, value: u32) -> u32 {
        let (respond_to: Sender<u32>, recv: Receiver<u32>) = oneshot::channel();
        self.sender Sender<(Box<dyn FnMut(&mut Calculator, …) -…
            .send((Box::new(Calculator::sum), Box::new((value, respond_to)))) impl Future<Output = Result<…, …>>
            .await Result<(), SendError<(Box<…>, …)>>
            .unwrap();
        recv.await.unwrap()
    }

    pub fn new() -> Self {
        let (sender: Sender<(Box<dyn FnMut(&mut Calculator, …)…, receiver) = mpsc::channel(8);
        let actor: Calculator = Calculator::new(receiver);
        tokio::spawn(future: run_my_actor(actor));
        Self { sender }
    }
} impl MyActorHandle
```

Interaction is clean, but not type checked at compile time!

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    let handle: MyActorHandle = MyActorHandle::new();
    let _pong: Pong = handle.ping().await;
    assert_eq!(handle.sum(5).await, 5);
    assert_eq!(handle.sum(10).await, 15);
}
```



```
     Running `target\debug\actor-model-examples.exe`
thread 'tokio-runtime-worker' panicked at src\main.rs:39:81:
called `Result::unwrap()` on an `Err` value: Any { .. }
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
thread 'main' panicked at src\main.rs:81:20:
called `Result::unwrap()` on an `Err` value: RecvError(())
error: process didn't exit successfully: `target\debug\actor-model-examples.exe` (exit code: 101)
```

Using message structs with Actix

Defining each messages

```
// Specific message structs needs to be created
#[derive(Message)]
#[rtype(u32)]
1 implementation
struct Sum(u32);

#[derive(Message)]
#[rtype(Pong)]
1 implementation
struct Ping;

#[derive(MessageResponse)]
1 implementation
struct Pong;
```

Implementing the actor once and the handle for each message

```rust
// An actor 'Calculator' is defined
#[derive(Default)]
4 implementations
struct Calculator {
    current: u32,
}

impl Actor for Calculator {
    type Context = Context<Self>;
}

// Implement `Handler` on `Calculator` for the `Sum` message.
impl Handler<Sum> for Calculator {
    type Result = u32; // <- Message response type

    fn handle(&mut self, msg: Sum, _ctx: &mut Context<Self>) -> Self::Result {
        self.current += msg.0;
        self.current
    }
}

// Implement `Handler` on `Calculator` for the `Ping` message.
impl Handler<Ping> for Calculator {
    type Result = Pong;

    fn handle(&mut self, _msg: Ping, _ctx: &mut Context<Self>) -> Self::Result {
        Pong
    }
}
```

Interaction feels less ‘natural’ and there is no local reuse

```
#[actix::main]
▶ Run | Debug
async fn main() {
    // Spawn the Calculator actor
    let addr: Addr<Calculator> = Calculator::default().start();

    // Send messages and await the result
    let _pong: Pong = addr.send(Ping).await.unwrap();
    assert_eq!(addr.send(Sum(5)).await.unwrap(), 5);
    assert_eq!(addr.send(Sum(10)).await.unwrap(), 15);
    let _pong = Calculator::default().ping();    no method named `ping` found for struct `Calculator` in the current scope
}
```

Async is not included in the trait definition

```rust
#[derive(MessageResponse)]
2 implementations
struct Pong;

impl Pong {
    async fn create() -> Self {
        Self
    }
}

impl Handler<Ping> for Calculator {
    type Result = Pong;

    fn handle(&mut self, _msg: Ping, _ctx: &mut Context<Self>) -> Self::Result {
        Pong::create().await    `await` is only allowed inside `async` functions and blocks
    }
}
```

Using Actify

Implementing the actor is ‘business as usual’

```
// An actor 'Calculator' is defined
#[derive(Clone, Debug, Default)]
4 implementations
struct Calculator {
    current: u32,
}

0 implementations
struct Pong;

#[actify] // Only add the actify attribute
impl Calculator {
    fn sum(&mut self, value: u32) -> u32 {
        self.current += value;
        self.current
    }

    fn ping(&self) -> Pong {
        Pong
    }
}
```

Interaction with the actor is natural

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    // Spawn the Calculator actor
    let handle: Handle<Calculator> = Handle::new(Calculator::default());

    // Send message and await the result
    let _pong: Pong = handle.ping().await.unwrap();
    assert_eq!(handle.sum(5).await.unwrap(), 5);
    assert_eq!(handle.sum(10).await.unwrap(), 15);
}
```

Async and generics work as expected

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    let handle: Handle<Calculator> = Handle::new(Calculator::default());
    handle.log("warn: something failed!").await.unwrap();
    handle.log(15).await.unwrap();
}

#[actify] // Only add the actify attribute
impl Calculator {
    async fn log<T: Debug + Send + Sync + 'static>(&self, log: T) {
        write_to_file(log).await
    }

    fn sum(&mut self, value: u32) -> u32 {
        self.current += value;
        self.current
    }
```

So how does it work?

- There are two structs defined in the Actify lib:
  - Actor<T>
  - Handle<T>
- If a method is defined on T, you want an *identical method* to be available in Handle<T>
- Calling the method copy should invoke the actual method in the actor, and return the result
- The question is: how to ‘couple’ the identical method? How to send the message? And how to automate this using a macro?
- Enums in messages cannot easily be generated through a macro, but dynamic dispatch can.
- Although the Box<dyn Any> approach is a bad approach manually, it is guaranteed to work with macros

```rust
#[actify]
impl Greeter {
    fn say_hi(&self, name: String) -> String {
        format!("hi {}", name)
    }
}
```

The macro generates and impls a trait using the Any approach

```rust
// Defines the wrappers that execute the original methods on the struct in the actor
trait GreeterActor {
    fn _say_hi(&mut self, args: Box<dyn std::any::Any + Send>) -> Result<Box<dyn std::any::Any + Send>, ActorError>;
}

// Implements the methods on the actor for this specific type
impl GreeterActor for Actor<Greeter>
{
    fn _say_hi(&mut self, args: Box<dyn std::any::Any + Send>) -> Result<Box<dyn std::any::Any + Send>, ActorError> {
        let name: String = *args.downcast().unwrap();

        // This call is the actual execution of the method from the user-defined impl block, on the struct held by the actor
        let result: String = self.inner.say_hi(name);
        Ok(Box::new(result))
    }
}
```

A similar trait is implemented to call the hidden Any method

```rust
// Defines the custom function signatures that should be added to the handle
#[async_trait::async_trait]
pub trait GreeterHandle {
    async fn say_hi(&self, name: String) -> Result<String, ActorError>;
}

// Implements the methods on the handle, and calls the generated method for the actor
#[async_trait::async_trait]
impl GreeterHandle for Handle<Greeter> {
    async fn say_hi(&self, name: String) -> Result<String, ActorError> {
        let res = self
            .send_job(FnType::Inner(Box::new(GreeterActor::_say_hi)), Box::new(name))
            .await?;
        Ok(*res.downcast().unwrap())
    }
}
```

Type checks work as the macro does not make typos

```
#[actify]
impl Greeter {
    fn say_hi(&self, name: String) -> String {
        format!("hi {}", name)
    }
}

#[tokio::main]
async fn main() {
    // An actify handle is created and initialized with the Greeter struct
    let handle = Handle::new(Greeter {});

    // The say_hi method is made available on its handle through the actify! macro
    let greeting = handle.say_hi("Alfred".to_string()).await.unwrap();

    // The method is executed on the initialized Greeter and returned through the handle
    assert_eq!(greeting, "hi Alfred".to_string())
}
```

Async is more difficult: BoxFuture ≠ impl Future

```rust
type ActorMethod<T> = Box<dyn FnMut(&mut T, Box<dyn Any + Send>) -> Box<dyn Any + Send> + Send>;

type BoxFuture<'a, R> = Pin<Box<dyn Future<Output = R> + Send + 'a>>;
type AsyncActorMethod<T> =
    Box<dyn FnMut(&mut T, Box<dyn Any + Send>) -> BoxFuture<Box<dyn Any + Send>> + Send + Sync>;
```

an async method returns impl future, which you cannot define yourself

```
`impl Trait` only allowed in function and inherent method argument and return types, not in `Fn` trait return types
```

An async method as example

```rust
struct Calculator {
    receiver: mpsc::Receiver<(AsyncActorMethod<Calculator>, Box<dyn Any + Send>)>,
    current: u32,
}

impl Calculator {
    async fn async_sum(&mut self, args: Box<dyn Any + Send>) -> Box<dyn Any + Send> {
        let (value: u32, respond_to: Sender<u32>): (u32, oneshot::Sender<u32>) = *args.downcast().unwrap();
        self.current += value;
        Box::new(respond_to.send(self.current).unwrap())
    }
}

impl Calculator {
    fn new(receiver: mpsc::Receiver<(AsyncActorMethod<Calculator>, Box<dyn Any + Send>)>) -> Self {
        Calculator {
            receiver,
            current: 0,
        }
    }
}
```

Naively boxing the method does not work as expected

```rust
impl MyCalculatorHandle {
    pub async fn async_sum(&self, value: u32) -> u32 {
        let (respond_to: Sender<u32>, recv: Receiver<u32>) = oneshot::channel();

        self.sender Sender<(Box<dyn FnMut(&mut Calculator, …) -…
            .send((
                Box::new(Calculator::async_sum),
                Box::new((value, respond_to)),
            )) impl Future<Output = Result<…, …>>
            .await Result<(), SendError<(Box<…>, …)>>
            .unwrap();
        recv.await.unwrap()
    }
}
```

```
error[E0271]: expected `async_sum` to be a fn item that returns `Pin<Box<dyn Future<Output = Box<dyn Any + Send>> + Send>>`,
 but it returns `impl Future<Output = Box<dyn Any + Send>>`
   |
31 |                 Box::new(Calculator::async_sum),
   |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected `Pin<Box<dyn Future<Output = Box<dyn Any + Send>> + Send>   |
   |
31 |                 Box::new(Calculator::async_sum),
```

You need a boxed closure that pins a future

```rust
impl MyCalculatorHandle {
    pub async fn async_sum(&self, value: u32) -> u32 {
        let (respond_to: Sender<u32>, recv: Receiver<u32>) = oneshot::channel();

        let boxed_future: AsyncActorMethod<Calculator> = Box::new(
            |actor: &mut Calculator, args: Box<dyn std::any::Any + Send>| {
                Box::pin(async move { Calculator::async_sum(self: actor, args).await })
            },
        );

        self.sender Sender<(Box<dyn FnMut(&mut Calculator, …) -…
            .send((boxed_future, Box::new((value, respond_to)))) impl Future<Output = Result<…, …>>
            .await Result<(), SendError<(Box<…>, …)>>
            .unwrap();
        recv.await.unwrap()
    }
}
```

Then it works as expected

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    let handle: MyCalculatorHandle = MyCalculatorHandle::new();
    assert_eq!(handle.async_sum(5).await, 5);
    assert_eq!(handle.async_sum(10).await, 15);
}
```

A concurrent example

A simple counter

```rust
#[derive(Debug, Clone, Default)]
4 implementations
struct Counter {
    count: usize,
}

#[actify]
impl Counter {
    fn increment(&mut self) -> usize {
        self.count += 1;
        self.count
    }
}
```

Multi threaded increments update a single actor

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    let counter_handle: Handle<Counter> = Handle::new(Counter::default());

    let counter_handle_clone: Handle<Counter> = counter_handle.clone();
    tokio::spawn(async move {
        for _ in 0..10 {
            counter_handle_clone.increment().await.unwrap();
            sleep(Duration::from_millis(100)).await;
        }
    });

    let counter_handle_clone: Handle<Counter> = counter_handle.clone();
    tokio::spawn(async move {
        for _ in 0..10 {
            counter_handle_clone.increment().await.unwrap();
            sleep(Duration::from_millis(100)).await;
        }
    });

    sleep(Duration::from_millis(2000)).await;

    let current_count: usize = counter_handle.increment().await.unwrap();
    assert_eq!(current_count, 21)
}
```

Concluding

- The Actify actor model allows for easy message passing in a Tokio runtime environment.
- No messages or enums are required.
- Generics and async methods work as expected.
- The method is also available on the local struct without any actors involved.
- It’s used extensively in our production environment, without any issues.
- Performance is excellent; barely any overhead more than Tokio itself
- The drawback is you have to import the generated “[insert actor]Handle” trait to expose the method.

Probably still a lot of improvements exist; more use cases and contributions are welcome!

Q&A

```
#[derive(Debug, Clone, Default)]
4 implementations
struct RustMeetup<S> {
    _speaker: S,
}

#[derive(Debug, Clone, Default)]
3 implementations
struct Maurits {}

#[actify]
impl RustMeetup<Maurits> {
    async fn has_done_talk(&self) -> bool {
        true
    }
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let speaker: Handle<RustMeetup<Maurits>> = Handle::new(RustMeetup::<Maurits>::default());
    assert!(speaker.has_done_talk().await.unwrap())
}
```