Quick start

Before you can start writing an actix application, you'll need a version of Rust installed. We recommend you use rustup to install or configure such a version.

Install Rust

Before we begin, we need to install Rust using the rustup installer:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

If you already have rustup installed, run this command to ensure you have the latest version of Rust:

```
rustup update
```

The actix framework requires Rust version 1.40.0 and up.

Running Examples

The fastest way to start experimenting with actix is to clone the actix repository and run the included examples in the examples/ directory. The following set of commands runs the `ping` example:

```
git clone https://github.com/actix/actix
cd actix
cargo run --example ping
```

Check examples/ directory for more examples.

Getting Started

Let's create and run our first actix application. We'll create a new Cargo project that depends on actix and then run the application.

In previous section we already installed required rust version. Now let's create new cargo projects.

Ping actor

Let’s write our first actix application! Start by creating a new binary-based Cargo project and changing into the new directory:

```
cargo new actor-ping
cd actor-ping
```

Now, add actix as a dependency of your project by ensuring your Cargo.toml contains the following:

```
[dependencies]
actix = "0.11.0"
actix-rt = "2.2" # <-- Runtime for actix
```

Let's create an actor that will accept a `Ping` message and respond with the number of pings processed.

An actor is a type that implements the `Actor` trait:

```
use actix::prelude::*;

struct MyActor {
    count: usize,
}

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

Each actor has an execution context, for `MyActor` we are going to use `Context<A>`. More information on actor contexts is available in the next section.

Now we need to define the `Message` that the actor needs to accept. The message can be any type that implements the `Message` trait.

```
use actix::prelude::*;

#[derive(Message)]
#[rtype(result = "usize")]
struct Ping(usize);
```

The main purpose of the `Message` trait is to define a result type. The `Ping` message defines `usize`, which indicates that any actor that can accept a `Ping` message needs to return `usize` value.

And finally, we need to declare that our actor `MyActor` can accept `Ping` and handle it. To do this, the actor needs to implement the `Handler<Ping>` trait.

```
impl Handler<Ping> for MyActor {
    type Result = usize;

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

        self.count
    }
}
```

That's it. Now we just need to start our actor and send a message to it. The start procedure depends on the actor's context implementation. In our case we can use `Context<A>` which is tokio/future based. We can start it with `Actor::start()` or `Actor::create()`. The first is used when the actor instance can be created immediately. The second method is used in case we need access to the context object before we can create the actor instance. In case of the `MyActor` actor we can use `start()`.

All communication with actors goes through an address. You can `do_send` a message without waiting for a response, or `send` to an actor with a specific message. Both `start()` and `create()` return an address object.

In the following example we are going to create a `MyActor` actor and send one message.

Here we use the actix-rt as way to start our System and drive our main Future so we can easily `.await` for the messages sent to the Actor.

```rust
#[actix_rt::main]
async fn main() {
    // start new actor
    let addr = MyActor { count: 10 }.start();

    // send message and get future for result
    let res = addr.send(Ping(10)).await;

    // handle() returns tokio handle
    println!("RESULT: {}", res.unwrap() == 20);

    // stop system and exit
    System::current().stop();
}
```

`#[actix_rt::main]` starts the system and block until future resolves.

The Ping example is available in the examples directory.

Actor

Actix is a rust library providing a framework for developing concurrent applications.

Actix is built on the Actor Model which allows applications to be written as a group of independently executing but cooperating "Actors" which communicate via messages. Actors are objects which encapsulate state and behavior and run within the *Actor System* provided by the actix library.

Actors run within a specific execution context `Context<A>`. The context object is available only during execution. Each actor has a separate execution context. The execution context also controls the lifecycle of an actor.

Actors communicate exclusively by exchanging messages. The sending actor can optionally wait for the response. Actors are not referenced directly, but by means of addresses.

Any rust type can be an actor, it only needs to implement the `Actor` trait.

To be able to handle a specific message the actor has to provide a `Handler<M>` implementation for this message. All messages are statically typed. The message can be handled in an asynchronous fashion. Actor can spawn other actors or add futures or streams to execution context. The `Actor` trait provides several methods that allow controlling the actor's lifecycle.

Actor lifecycle

Started

An actor always starts in the `Started` state. During this state the actor's `started()` method is called. The `Actor` trait provides a default implementation for this method. The actor context is available during this state and the actor can start more actors or register async streams or do any other required configuration.

Running

After an Actor's `started()` method is called, the actor transitions to the `Running` state. The Actor can stay in `running` state indefinitely.

Stopping

The Actor's execution state changes to the `stopping` state in the following situations:

- `Context::stop` is called by the actor itself
- all addresses to the actor get dropped. i.e. no other actor references it.
- no event objects are registered in the context.

An actor can restore from the `stopping` state to the `running` state by creating a new address or adding an event object, and by returning `Running::Continue`.

If an actor changed state to `stopping` because `Context::stop()` is called then the context immediately stops processing incoming messages and calls `Actor::stopping()`. If the actor does not restore back to the `running` state, all unprocessed messages are dropped.

By default this method returns `Running::Stop` which confirms the stop operation.

Stopped

If the actor does not modify the execution context during the stopping state, the actor state changes to `Stopped`. This state is considered final and at this point the actor is

dropped.

Message

An Actor communicates with other actors by sending messages. In actix all messages are typed. A message can be any rust type which implements the `Message` trait. `Message::Result` defines the return type. Let's define a simple `Ping` message - an actor which will accept this message needs to return `Result<bool, std::io::Error>`.

```
use actix::prelude::*;

struct Ping;

impl Message for Ping {
    type Result = Result<bool, std::io::Error>;
}
```

Spawning an actor

How to start an actor depends on its context. Spawning a new async actor is achieved via the `start` and `create` methods of the `Actor` trait. It provides several different ways of creating actors; for details check the docs.

Complete example

```rust
use actix::prelude::*;

/// Define message
#[derive(Message)]
#[rtype(result = "Result<bool, std::io::Error>")]
struct Ping;

// Define actor
struct MyActor;

// Provide Actor implementation for our actor
impl Actor for MyActor {
    type Context = Context<Self>;

    fn started(&mut self, ctx: &mut Context<Self>) {
       println!("Actor is alive");
    }

    fn stopped(&mut self, ctx: &mut Context<Self>) {
       println!("Actor is stopped");
    }
}

/// Define handler for `Ping` message
impl Handler<Ping> for MyActor {
    type Result = Result<bool, std::io::Error>;

    fn handle(&mut self, msg: Ping, ctx: &mut Context<Self>) -> Self::Result
{
        println!("Ping received");

        Ok(true)
    }
}

#[actix_rt::main]
async fn main() {
    // Start MyActor in current thread
    let addr = MyActor.start();

    // Send Ping message.
    // send() message returns Future object, that resolves to message result
    let result = addr.send(Ping).await;

    match result {
        Ok(res) => println!("Got result: {}", res.unwrap()),
        Err(err) => println!("Got error: {}", err),
    }
}
```

Responding with a MessageResponse

Let's take a look at the `Result` type defined for the `impl Handler` in the above example. See how we're returning a `Result<bool, std::io::Error>`? We're able to respond to our actor's incoming message with this type because it has the `MessageResponse` trait implemented for that type. Here's the definition for that trait:

```
pub trait MessageResponse<A: Actor, M: Message> {
    fn handle(self, ctx: &mut A::Context, tx:
Option<OneshotSender<M::Result>>);
}
```

Sometimes it makes sense to respond to incoming messages with types that don't have this trait implemented for them. When that happens we can implement the trait ourselves. Here's an example where we're responding to a `Ping` message with a `GotPing`, and responding with `GotPong` for a `Pong` message.

```
use actix::dev::{MessageResponse,OneshotSender};
use actix::prelude::*;

#[derive(Message)]
#[rtype(result = "Responses")]
enum Messages {
    Ping,
    Pong,
}

enum Responses {
    GotPing,
    GotPong,
}

impl<A, M> MessageResponse<A, M> for Responses
where
    A: Actor,
    M: Message<Result = Responses>,
{
    fn handle(self, ctx: &mut A::Context, tx:
Option<OneshotSender<M::Result>>) {
        if let Some(tx) = tx {
            tx.send(self);
        }
    }
}

// Define actor
```

```
struct MyActor;

// Provide Actor implementation for our actor
impl Actor for MyActor {
    type Context = Context<Self>;

    fn started(&mut self, _ctx: &mut Context<Self>) {
        println!("Actor is alive");
    }

    fn stopped(&mut self, _ctx: &mut Context<Self>) {
        println!("Actor is stopped");
    }
}

/// Define handler for `Messages` enum
impl Handler<Messages> for MyActor {
    type Result = Responses;

    fn handle(&mut self, msg: Messages, _ctx: &mut Context<Self>) ->
Self::Result {
        match msg {
            Messages::Ping => Responses::GotPing,
            Messages::Pong => Responses::GotPong,
        }
    }
}

#[actix_rt::main]
async fn main() {
    // Start MyActor in current thread
    let addr = MyActor.start();

    // Send Ping message.
    // send() message returns Future object, that resolves to message result
    let ping_future = addr.send(Messages::Ping).await;
    let pong_future = addr.send(Messages::Pong).await;

    match pong_future {
        Ok(res) => match res {
            Responses::GotPing => println!("Ping received"),
            Responses::GotPong => println!("Pong received"),
        },
        Err(e) => println!("Actor is probably dead: {}", e),
    }

    match ping_future {
        Ok(res) => match res {
            Responses::GotPing => println!("Ping received"),
            Responses::GotPong => println!("Pong received"),
```

```
        },
        Err(e) => println!("Actor is probably dead: {}", e),
    }
}
```

Address

Actors communicate exclusively by exchanging messages. The sending actor can optionally wait for the response. Actors cannot be referenced directly, only by their addresses.

There are several ways to get the address of an actor. The `Actor` trait provides two helper methods for starting an actor. Both return the address of the started actor.

Here is an example of `Actor::start()` method usage. In this example `MyActor` actor is asynchronous and is started in the same thread as the caller - threads are covered in the SyncArbiter chapter.

```
struct MyActor;
impl Actor for MyActor {
    type Context = Context<Self>;
}

let addr = MyActor.start();
```

An async actor can get its address from the `Context` struct. The context needs to implement the `AsyncContext` trait. `AsyncContext::address()` provides the actor's address.

```
struct MyActor;

impl Actor for MyActor {
    type Context = Context<Self>;

    fn started(&mut self, ctx: &mut Context<Self>) {
       let addr = ctx.address();
    }
}
```

Message

To be able to handle a specific message the actor has to provide a `Handler<M>` implementation for this message. All messages are statically typed. The message can be handled in an asynchronous fashion. The actor can spawn other actors or add futures or streams to the execution context. The actor trait provides several methods that allow controlling the actor's lifecycle.

To send a message to an actor, the `Addr` object needs to be used. `Addr` provides several ways to send a message.

- `Addr::do_send(M)` - this method ignores any errors in message sending. If the mailbox is full the message is still queued, bypassing the limit. If the actor's mailbox is closed, the message is silently dropped. This method does not return the result, so if the mailbox is closed and a failure occurs, you won't have an indication of this.
- `Addr::try_send(M)` - this method tries to send the message immediately. If the mailbox is full or closed (actor is dead), this method returns a `SendError`.
- `Addr::send(M)` - This message returns a future object that resolves to a result of a message handling process. If the returned `Future` object is dropped, the message is cancelled.

Recipient

Recipient is a specialized version of an address that supports only one type of message. It can be used in case the message needs to be sent to a different type of actor. A recipient object can be created from an address with `Addr::recipient()`.

Address objects require an actor type, but if we just want to send a specific message to an actor that can handle the message, we can use the Recipient interface.

For example recipient can be used for a subscription system. In the following example `OrderEvents` actor sends a `OrderShipped` message to all subscribers. A subscriber can be any actor that implements the `Handler<OrderShipped>` trait.

```
use actix::prelude::*;

#[derive(Message)]
#[rtype(result = "()")]
struct OrderShipped(usize);
```

```rust
#[derive(Message)]
#[rtype(result = "()")]
struct Ship(usize);

/// Subscribe to order shipped event.
#[derive(Message)]
#[rtype(result = "()")]
struct Subscribe(pub Recipient<OrderShipped>);

/// Actor that provides order shipped event subscriptions
struct OrderEvents {
    subscribers: Vec<Recipient<OrderShipped>>,
}

impl OrderEvents {
    fn new() -> Self {
        OrderEvents {
            subscribers: vec![]
        }
    }
}

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

impl OrderEvents {
    /// Send event to all subscribers
    fn notify(&mut self, order_id: usize) {
        for subscr in &self.subscribers {
           subscr.do_send(OrderShipped(order_id));
        }
    }
}

/// Subscribe to shipment event
impl Handler<Subscribe> for OrderEvents {
    type Result = ();

    fn handle(&mut self, msg: Subscribe, _: &mut Self::Context) {
        self.subscribers.push(msg.0);
    }
}

/// Subscribe to ship message
impl Handler<Ship> for OrderEvents {
    type Result = ();
    fn handle(&mut self, msg: Ship, ctx: &mut Self::Context) -> Self::Result
{
        self.notify(msg.0);
```

```
        System::current().stop();
    }

}

/// Email Subscriber
struct EmailSubscriber;
impl Actor for EmailSubscriber {
    type Context = Context<Self>;
}

impl Handler<OrderShipped> for EmailSubscriber {
    type Result = ();
    fn handle(&mut self, msg: OrderShipped, _ctx: &mut Self::Context) ->
Self::Result {
        println!("Email sent for order {}", msg.0)
    }

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

impl Handler<OrderShipped> for SmsSubscriber {
    type Result = ();
    fn handle(&mut self, msg: OrderShipped, _ctx: &mut Self::Context) ->
Self::Result {
        println!("SMS sent for order {}", msg.0)
    }

}

fn main() {
    let system = System::new("events");
    let email_subscriber = Subscribe(EmailSubscriber{}.start().recipient());
    let sms_subscriber = Subscribe(SmsSubscriber{}.start().recipient());
    let order_event = OrderEvents::new().start();
    order_event.do_send(email_subscriber);
    order_event.do_send(sms_subscriber);
    order_event.do_send(Ship(1));
    system.run();
}
```

Context

Actors all maintain an internal execution context, or state. This allows an actor to determine its own Address, change mailbox limits, or stop its execution.

Mailbox

All messages go to the actor's mailbox first, then the actor's execution context calls specific message handlers. Mailboxes in general are bounded. The capacity is specific to the context implementation. For the `Context` type the capacity is set to 16 messages by default and can be increased with `Context::set_mailbox_capacity()`.

```
struct MyActor;

impl Actor for MyActor {
    type Context = Context<Self>;

    fn started(&mut self, ctx: &mut Self::Context) {
        ctx.set_mailbox_capacity(1);
    }
}

let addr = MyActor.start();
```

Remember that this doesn't apply to `Addr::do_send(M)` which bypasses the Mailbox queue limit, or `AsyncContext::notify(M)` and `AsyncContext::notify_later(M, Duration)` which bypasses the mailbox entirely.

Getting your actors Address

An actor can view its own address from its context. Perhaps you want to requeue an event for later, or you want to transform the message type. Maybe you want to respond with your address to a message. If you want an actor to send a message to itself, have a look at `AsyncContext::notify(M)` instead.

To get your address from the context you call `Context::address()`. An example is:

```
struct MyActor;

struct WhoAmI;

impl Message for WhoAmI {
    type Result = Result<actix::Addr<MyActor>, ()>;
}

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

impl Handler<WhoAmI> for MyActor {
    type Result = Result<actix::Addr<MyActor>, ()>;

    fn handle(&mut self, msg: WhoAmI, ctx: &mut Context<Self>) ->
Self::Result {
        Ok(ctx.address())
    }
}

let who_addr = addr.do_send(WhoAmI{});
```

Stopping an Actor

From within the actors execution context you can choose to stop the actor from processing any future Mailbox messages. This could be in response to an error condition, or as part of program shutdown. To do this you call `Context::stop()`.

This is an adjusted Ping example that stops after 4 pings are received.

```rust
impl Handler<Ping> for MyActor {
    type Result = usize;

    fn handle(&mut self, msg: Ping, ctx: &mut Context<Self>) -> Self::Result
{
        self.count += msg.0;

        if self.count > 5 {
            println!("Shutting down ping receiver.");
            ctx.stop()
        }

        self.count
    }
}

#[actix_rt::main]
async fn main() {
    // start new actor
    let addr = MyActor { count: 10 }.start();

    // send message and get future for result
    let addr_2 = addr.clone();
    let res = addr.send(Ping(6)).await;

    match res {
        Ok(_) => assert!(addr_2.try_send(Ping(6)).is_err()),
        _ => {}
    }
}
```

Arbiter

`Arbiter`s provide an asynchronous execution context for `Actor`s, `functions` and `futures`. Where an actor contains a `Context` that defines its Actor specific execution state, Arbiters host the environment where an actor runs.

As a result Arbiters perform a number of functions. Most notably, they are able to spawn a new OS thread, run an event loop, spawn tasks asynchronously on that event loop, and act as helpers for asynchronous tasks.

System and Arbiter

In all our previous code examples the function `System::new` creates an Arbiter for your actors to run inside. When you call `start()` on your actor it is then running inside of the System Arbiter's thread. In many cases, this is all you will need for a program using Actix.

While it only uses one thread, it uses the very efficient event loop pattern which works well for asynchronous events. To handle synchronous, CPU-bound tasks, it's better to avoid blocking the event loop and instead offload the computation to other threads. For this usecase, read the next section and consider using `SyncArbiter`.

The event loop

One `Arbiter` is in control of one thread with one event pool. When an Arbiter spawns a task (via `Arbiter::spawn`, `Context<Actor>::run_later`, or similar constructs), the Arbiter queues the task for execution on that task queue. When you think `Arbiter`, you can think "single-threaded event loop".

Actix in general does support concurrency, but normal `Arbiter`s (not `SyncArbiter`s) do not. To use Actix in a concurrent way, you can spin up multiple `Arbiter`s using `Arbiter::new`, `ArbiterBuilder`, or `Arbiter::start`.

When you create a new Arbiter, this creates a new execution context for Actors. The new thread is available to add new Actors to it, but Actors cannot freely move between Arbiters: they are tied to the Arbiter they were spawned in. However, Actors on different Arbiters can still communicate with each other using the normal `Addr`/`Recipient` methods. The method of passing messages is agnostic to whether the Actors are running on the same or different Arbiters.

Using Arbiter for resolving async events

If you aren't an expert in Rust Futures, Arbiter can be a helpful and simple wrapper to resolving async events in order. Consider we have two actors, A and B, and we want to run an event on B only once a result from A is completed. We can use `Arbiter::spawn` to assist with this task.

```
use actix::prelude::*;

struct SumActor {}
```

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

#[derive(Message)]
#[rtype(result = "usize")]
struct Value(usize, usize);

impl Handler<Value> for SumActor {
    type Result = usize;

    fn handle(&mut self, msg: Value, _ctx: &mut Context<Self>) ->
Self::Result {
        msg.0 + msg.1
    }
}

struct DisplayActor {}

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

#[derive(Message)]
#[rtype(result = "()")]
struct Display(usize);

impl Handler<Display> for DisplayActor {
    type Result = ();

    fn handle(&mut self, msg: Display, _ctx: &mut Context<Self>) ->
Self::Result {
        println!("Got {:?}", msg.0);
    }
}

fn main() {
    let system = System::new("single-arbiter-example");

    // Define an execution flow using futures
    let execution = async {
        // `Actor::start` spawns the `Actor` on the *current* `Arbiter`,
which
        // in this case is the System arbiter
        let sum_addr = SumActor {}.start();
        let dis_addr = DisplayActor {}.start();

        // Start by sending a `Value(6, 7)` to our `SumActor`.
        // `Addr::send` responds with a `Request`, which implements `Future`.
```

```
        // When awaited, it will resolve to a `Result<usize, MailboxError>`.
        let sum_result = sum_addr.send(Value(6, 7)).await;

        match sum_result {
            Ok(res) => {
                // `res` is now the `usize` returned from `SumActor` as a
response to `Value(6, 7)`
                // Once the future is complete, send the successful response
(`usize`)
                // to the `DisplayActor` wrapped in a `Display
                dis_addr.send(Display(res)).await;
            }
            Err(e) => {
                eprintln!("Encountered mailbox error: {:?}", e);
            }
        };
    };

    // Spawn the future onto the current Arbiter/event loop
    Arbiter::spawn(execution);

    // We only want to do one computation in this example, so we
    // shut down the `System` which will stop any Arbiters within
    // it (including the System Arbiter), which will in turn stop
    // any Actor Contexts running within those Arbiters, finally
    // shutting down all Actors.
    System::current().stop();

    system.run();
}
```

SyncArbiter

When you normally run Actors, there are multiple Actors running on the System's Arbiter thread, using its event loop. However for CPU bound workloads, or highly concurrent workloads, you may wish to have an Actor running multiple instances in parallel.

This is what a SyncArbiter provides - the ability to launch multiple instances of an Actor on a pool of OS threads.

It's important to note a SyncArbiter can only host a single type of Actor. This means you need to create a SyncArbiter for each type of Actor you want to run in this manner.

Creating a Sync Actor

When implementing your Actor to be run on a SyncArbiter, it requires that your Actor's Context is changed from `Context` to `SyncContext`.

```
use actix::prelude::*;

struct MySyncActor;

impl Actor for MySyncActor {
    type Context = SyncContext<Self>;
}
```

Starting the Sync Arbiter

Now that we have defined a Sync Actor, we can run it on a thread pool, created by our `SyncArbiter`. We can only control the number of threads at SyncArbiter creation time - we can't add/remove threads later.

```
use actix::prelude::*;

struct MySyncActor;

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

let addr = SyncArbiter::start(2, || MySyncActor);
```

We can communicate with the addr the same way as we have with our previous Actors that we started. We can send messages, receive futures and results, and more.

Sync Actor Mailboxes

Sync Actors have no Mailbox limits, but you should still use `do_send`, `try_send` and `send` as normal to account for other possible errors or sync vs async behavior.

WIP

WIP

WIP

WIP

WIP