Project: Event-Driven Actor Microservice Framework in Rust

```
// Cargo.toml dependencies:
//
// [dependencies]
// tokio = { version = "1", features = ["full"] }
// async-trait = "0.1"
// thiserror = "1.0"
// uuid = { version = "1", features = ["v4"] }
// serde = { version = "1", features = ["derive"] }
// serde_json = "1"

use async_trait::async_trait;
use std::{
    collections::HashMap,
    sync::{Arc},
};
use thiserror::Error;
use tokio::{
    sync::{mpsc, RwLock},
    task,
};
use uuid::Uuid;
```

```
// ======================
// Core Actor Abstractions
// ======================

#[derive(Debug, Error)]
pub enum ActorError {
    #[error("Actor failed to process message: {0}")]
    ProcessingError(String),

    #[error("Actor communication failed")]
    CommunicationError,
}

#[async_trait]
pub trait Actor: Send + Sync + 'static {
    async fn handle(&mut self, msg: Message) -> Result<(), ActorError>;
}

// Message enum for polymorphic dispatch
#[derive(Debug, Clone)]
pub enum Message {
    Ping,
    Echo(String),
    Shutdown,
    Custom(Box<dyn std::any::Any + Send + Sync>), // Advanced use case
```

```
}

// ======================
// Actor Context and System
// ======================

type ActorId = Uuid;

#[derive(Clone)]
struct ActorRef {
    sender: mpsc::Sender<Message>,
}

impl ActorRef {
    pub async fn send(&self, msg: Message) -> Result<(), ActorError> {
        self.sender.send(msg).await.map_err(|_| ActorError::CommunicationError)
    }
}

struct ActorSystem {
    registry: Arc<RwLock<HashMap<ActorId, ActorRef>>>,
}

impl ActorSystem {
    pub fn new() -> Self {
```

```
    Self {
        registry: Arc::new(RwLock::new(HashMap::new())),
    }
}

pub async fn spawn<A>(&self, mut actor: A) -> ActorId
where
    A: Actor,
{
    let (tx, mut rx) = mpsc::channel::<Message>(100);
    let id = Uuid::new_v4();
    let actor_ref = ActorRef { sender: tx.clone() };

    self.registry.write().await.insert(id, actor_ref);

    let registry = self.registry.clone();
    task::spawn(async move {
        while let Some(msg) = rx.recv().await {
            if let Err(err) = actor.handle(msg).await {
                eprintln!("[Actor {id}] Error: {err}");
            }
        }
        // Cleanup after shutdown
        registry.write().await.remove(&id);
        println!("[Actor {id}] terminated.");
```

```
        });

        id
    }

    pub async fn send(&self, id: ActorId, msg: Message) -> Result<(), ActorError> {
        let registry = self.registry.read().await;
        registry
            .get(&id)
            .ok_or(ActorError::CommunicationError)?
            .send(msg)
            .await
    }

    pub async fn shutdown(&self) {
        let ids: Vec<ActorId> = self.registry.read().await.keys().cloned().collect();
        for id in ids {
            let _ = self.send(id, Message::Shutdown).await;
        }
    }
}

// ======================
// Example Actor Implementations
// ======================
```

```
struct EchoActor;

#[async_trait]
impl Actor for EchoActor {
    async fn handle(&mut self, msg: Message) -> Result<(), ActorError> {
        match msg {
            Message::Ping => {
                println!("[EchoActor] Received Ping.");
                Ok(())
            }
            Message::Echo(text) => {
                println!("[EchoActor] Echoing: {}", text);
                Ok(())
            }
            Message::Shutdown => {
                println!("[EchoActor] Received Shutdown.");
                Ok(())
            }
            _ => Err(ActorError::ProcessingError("Unsupported message".into())),
        }
    }
}

struct CounterActor {
```

```
    count: u64,
}

#[async_trait]
impl Actor for CounterActor {
    async fn handle(&mut self, msg: Message) -> Result<(), ActorError> {
        match msg {
            Message::Ping => {
                self.count += 1;
                println!("[CounterActor] Ping count: {}", self.count);
                Ok(())
            }
            Message::Shutdown => {
                println!("[CounterActor] Shutting down at count: {}", self.count);
                Ok(())
            }
            _ => Err(ActorError::ProcessingError("Unsupported message".into())),
        }
    }
}

// ======================
// Testing Our Framework
// ======================
```

```
#[tokio::main]
async fn main() {
    let system = ActorSystem::new();

    let echo_id = system.spawn(EchoActor).await;
    let counter_id = system.spawn(CounterActor { count: 0 }).await;

    system.send(echo_id, Message::Echo("Hello from Rust Actor!".into())).await.unwrap();
    system.send(counter_id, Message::Ping).await.unwrap();
    system.send(counter_id, Message::Ping).await.unwrap();

    // Demonstrate custom message (only for future advanced actors)
    // system.send(echo_id, Message::Custom(Box::new(42))).await.unwrap();

    tokio::time::sleep(tokio::time::Duration::from_millis(500)).await;

    system.shutdown().await;
}
```