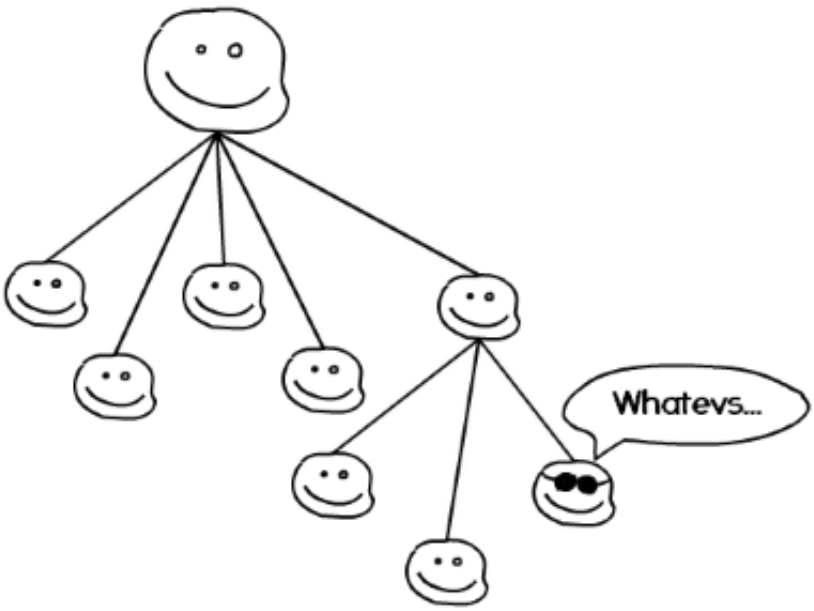
Erlang-style Supervisors in C# with Akka.NET and the Actor Model

One reason Erlang is considered a fault-tolerant language is its support for the Actor Model in its standard library

> *"... Erlang has been used to build systems with **99.9999999 (that's nine 9s)**. Used correctly, the error handling mechanisms can help make your program run forever (well, almost)" (Armstrong, 13)*

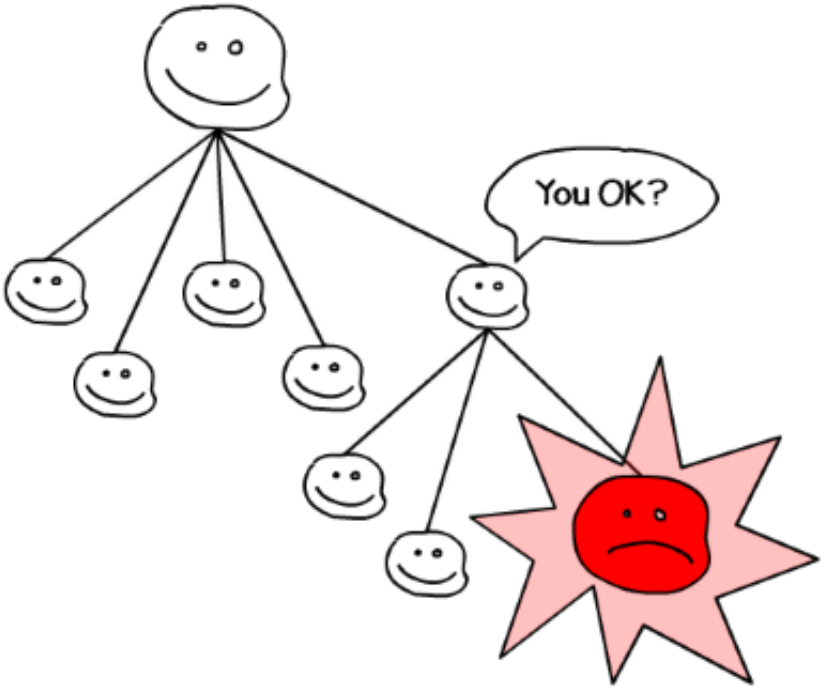
The Actor Model is effective when working with concurrency in F#

The resilience of the Actor Model comes from the ability to create hierarchies of specialized Actors. Some Actors have specializations that are more **risky** than others.



Risk-taking Actors are pushed lower in the hierarchy, away from the life-sustaining parts of the system (a.k.a. the Error Kernel). Since Actors **don't share state**, failure can be isolated from the Error Kernel.

Every Actor has an Actor above it that **Supervises** it. When a child Actor throws an error, its **Supervisor** can chose how to handle it.



The two languages I've mentioned so far, Erlang and F#, are both pretty much **functional languages**.

What if you aren't comfortable in these languages yet? **Is there a way to take advantage of Actor Supervision using C#?**

C# Actor Models

There are, in fact, multiple ways to use the Actor Model in C#, including:

- Akka.NET
- Microsoft Service Fabric
- TPL DataFlow

Of these options, **Akka.NET** is a good way to get started.

> *Akka.NET is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on .NET & Mono (getakka.net)*

If this post gets you excited about Akka.NET, I highly encourage you to do the Akka.NET Bootcamp. Their documentation and community support is excellent.

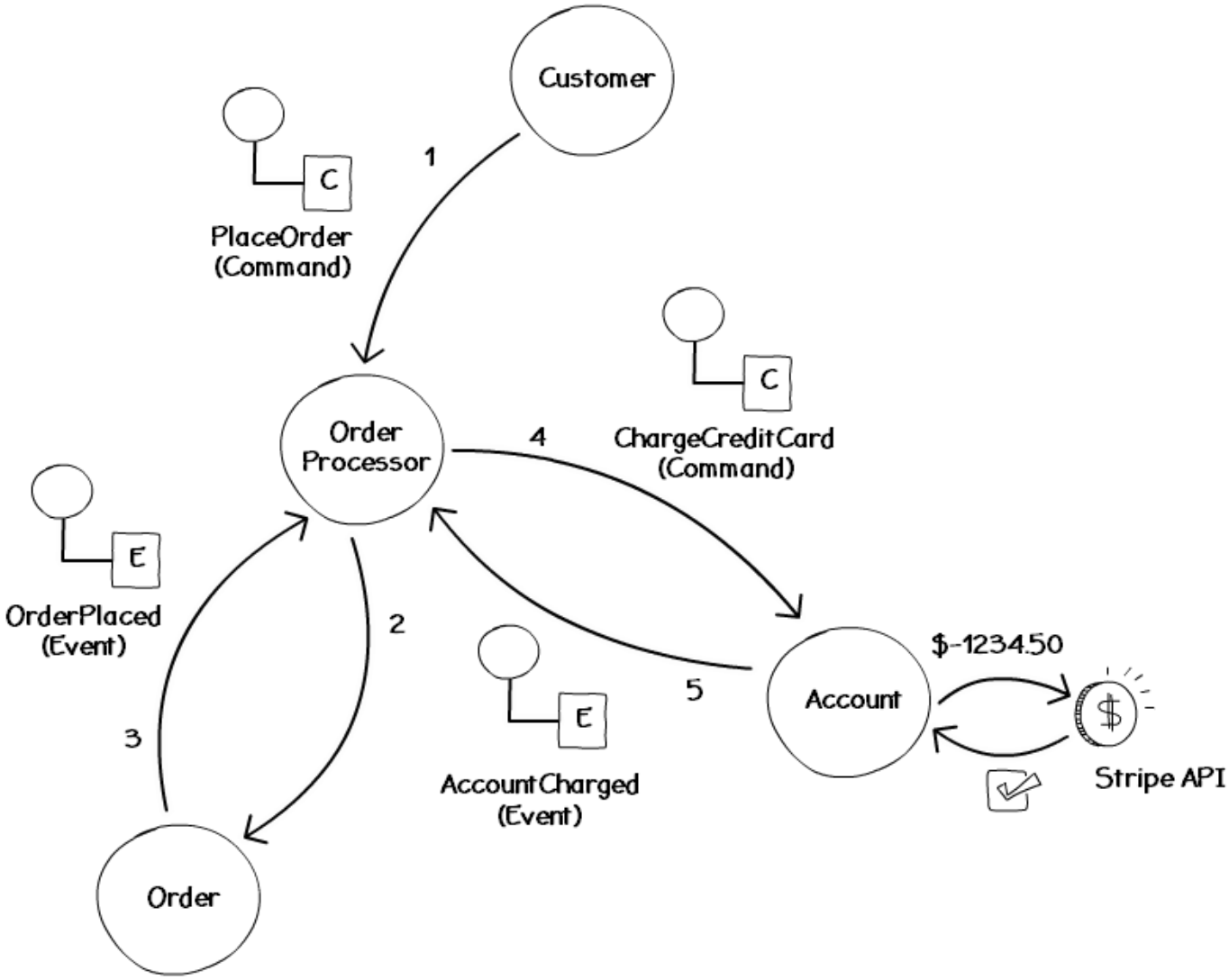
Example Use Case

Let's get our hands on an example of how Akka.NET uses Actor Supervision to give our system **Fault Tolerance**.

I'll borrow a use case from Vaughn Vernon's book, Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka...

> 1. *The `Customer` tells `OrderProcessor` to place an `Order` for book `B`.*
> 2. *The `OrderProcessor` creates `Order` for book `B`, and as a result an event `OrderPlaced` is created to indicate the fact.*
> 3. *Event `OrderPlaced` is delivered from `Order` to `OrderProcessor`.*
> 4. *`OrderProcessor` tells the `Account` to `ChargeCreditCard`, which succeeds, and event `AccountCharged` is created to indicate the fact.*
> 5. *Event `AccountCharged` is delivered from `Account` to `OrderProcessor`, which in turn tells `Order` to hold a reference to information conveyed by `AccountCharged`.*
> 6. *The use case continues by telling `Inventory` to reserve book `B` and telling the `Shipper` to ship the `Order`.*
>
> *(Vernon, 1)*

Here's a simple visualization of this use case:



Implementation

Let's briefly go through the Actors in our use case. Don't worry too much about the specifics here. This post is more about **Supervision** strategy than the Actor Model as a whole. The code can be found here.

The OrderProcessorActor

The `OrderProcessorActor` handles our Command and Event messages and coordinates the communication between them:

```
public class OrderProcessorActor : ReceiveActor {
    private readonly ILoggingAdapter _logger = Context.GetLogger();
    public OrderProcessorActor() {
        Receive<PlaceOrder>(placeOrder => PlaceOrderHandler(placeOrder));
        Receive<OrderPlaced>(orderPlaced => OrderPlacedHandler(orderPlaced));
        Receive<AccountCharged>(accountCharged =>
AccountChargedHandler(accountCharged));
    }
    private void PlaceOrderHandler(PlaceOrder placeOrder) {
        var orderActor = Context.ActorOf(
            Props.Create(
                () => new OrderActor(
                    (int)DateTime.Now.Ticks)),
            "orderActor" + DateTime.Now.Ticks);
        orderActor.Tell(placeOrder);
    }
    private void OrderPlacedHandler(OrderPlaced orderPlaced) {
        var accountActor = Context.ActorOf(
            Props.Create(
                () => new AccountActor(
                    orderPlaced.OrderInfo.AccountId)),
            "accountActor" + orderPlaced.OrderInfo.AccountId);
        accountActor.Tell(new ChargeCreditCard(orderPlaced.OrderInfo.ExtPrice));
    }
    private void AccountChargedHandler(AccountCharged accountCharged) {
        if (accountCharged.Success) {
            _logger.Info("Account charged!\n{0}",
                JsonConvert.SerializeObject(accountCharged));
            // Sends to TestActor (Test) or CustomerActor (Production)
            Context.Parent.Tell(accountCharged);
        }
        else {
            _logger.Warning("Error! Account not charged!");
            // Sends to TestActor (Test) or CustomerActor (Production)
            Context.Parent.Tell(accountCharged);
        }
    }
```

We'll use the `OrderProcessor` as an entry point to our Actor System. We'll also use Akka.NET's `TestKit` to drive this implementation with tests like this one:

```
[Fact]
public void OrderProcessorActor_end_to_end() {
    var message = new PlaceOrder(12345, 10, 25, 50.00m);
    var orderProcessorActor = ActorOfAsTestActorRef(
                () => new OrderProcessorActor(), TestActor
            );
    orderProcessorActor.Tell(message);

    Assert.True(ExpectMsg<AccountCharged>().Success);
}
```

The OrderActor

An `OrderActor` is **created** by the `OrderProcessorActor` for **every** `PlaceOrder` message it receives. It tells the `OrderProcessor` that the Order has been placed by responding with a `OrderPlaced` message.

```
class OrderActor : ReceiveActor {
    public OrderActor() {
        Receive<PlaceOrder>(placeOrder => PlaceOrderHandler(placeOrder));
    }

    public void PlaceOrderHandler(PlaceOrder placeOrder) {
        Context.Parent.Tell(
            new OrderPlaced(DateTime.Now.Ticks.ToString(), placeOrder));
    }
}
```

Here's a test to show that an `OrderActor` gets created for every `PlaceOrder` message:

```
[Fact]
public void OrderProcessorActor_handles_placeOrderCommand_creates_orderActor() {
    var message = new PlaceOrder(12345, 10, 25, 5000);
    var orderProcessorActor = ActorOfAsTestActorRef(
                () => new OrderProcessorActor(), "orderProcessor");
    orderProcessorActor.Tell(message);
    var orderChild = ActorSelection("/user/orderProcessor/orderActor*")
        .ResolveOne(TimeSpan.FromSeconds(3))
        .Result;

    Assert.True(orderChild.Path.ToString()
        .StartsWith("akka://test/user/orderProcessor/orderActor"));
}
```

The AccountActor

The `AccountActor` is going to be our focus, since it is responsible for the **risky** task of making sure the `Customer`'s credit card is charged successfully. This is the Actor that we want to put some **Fault Tolerance** around.

```
public class AccountActor : ReceiveActor {

public class AccountActor : ReceiveActor {
    public int AccountId { get; }
    private readonly IStripeGateway _stripeGateway = new StripeGateway();
    private readonly ILoggingAdapter _logger = Context.GetLogger();
    public AccountActor(int accountId) {
        AccountId = accountId;
        Receive<ChargeCreditCard>(
            chargeCreditCard => ChargeCreditCardHandler(chargeCreditCard));
    }
    private void ChargeCreditCardHandler(ChargeCreditCard chargeCreditCard) {
        StripeCharge stripeCharge = null;
        try {
            stripeCharge = _stripeGateway
                .CreateCharge(chargeCreditCard.Amount);
            if (stripeCharge != null)
                Context.Parent.Tell(new AccountCharged(chargeCreditCard, true));
        }
        catch (Exception) {
            Context.Parent.Tell(new AccountCharged(chargeCreditCard, false));
            throw;
        }
    }
```

Since the `AccountActor` is responsible for charging the `Customer`'s credit card, it probably uses a service like Stripe to process payments. We'll write a `StripeGateway` that the `AccountActor` uses as a dependency for calling the Stripe API.

We'll pull in the stripe.net library to use some of its types.

```
internal interface IStripeGateway {
    StripeCharge CreateCharge(decimal amount);
}

public class StripeGateway : IStripeGateway {
    public StripeCharge CreateCharge(decimal amount) {
        // Create the Stripe Charge
    }
}
```

`StripeGateway` has the potential to throw **Exceptions**. In a **Non-Actor** world, these Exceptions could impact the whole application. In our **Actor** world, we can isolate failure to the `AccountActor`.

Let's make the `StripeGateway.CreateCharge()` method throw an Exception when it's passed a negative `amount`:

```
public class StripeGateway : IStripeGateway {
    public StripeCharge CreateCharge(int amount) {
        if (amount < 0) {
            throw new StripeException(
                System.Net.HttpStatusCode.OK,
                new StripeError() {
                    Code = "card_error"
                }, "Can't charge card a negative value");
        }
        return new StripeCharge() {
            Amount = amount,
            Captured = true
        };
    }
}
```

Here's a test that shows that this `StripeException` is raised:

```
[Fact]
public void AccountActor_throws_exception_from_stripeGateway() {
    var message = new ChargeCreditCard(-1234.50m);
    var accountActor = ActorOf(Props.Create(() => new AccountActor(12345)));

    EventFilter.Exception<StripeException>()
        .ExpectOne(() => accountActor.Tell(message));
}
```

Supervision Strategies

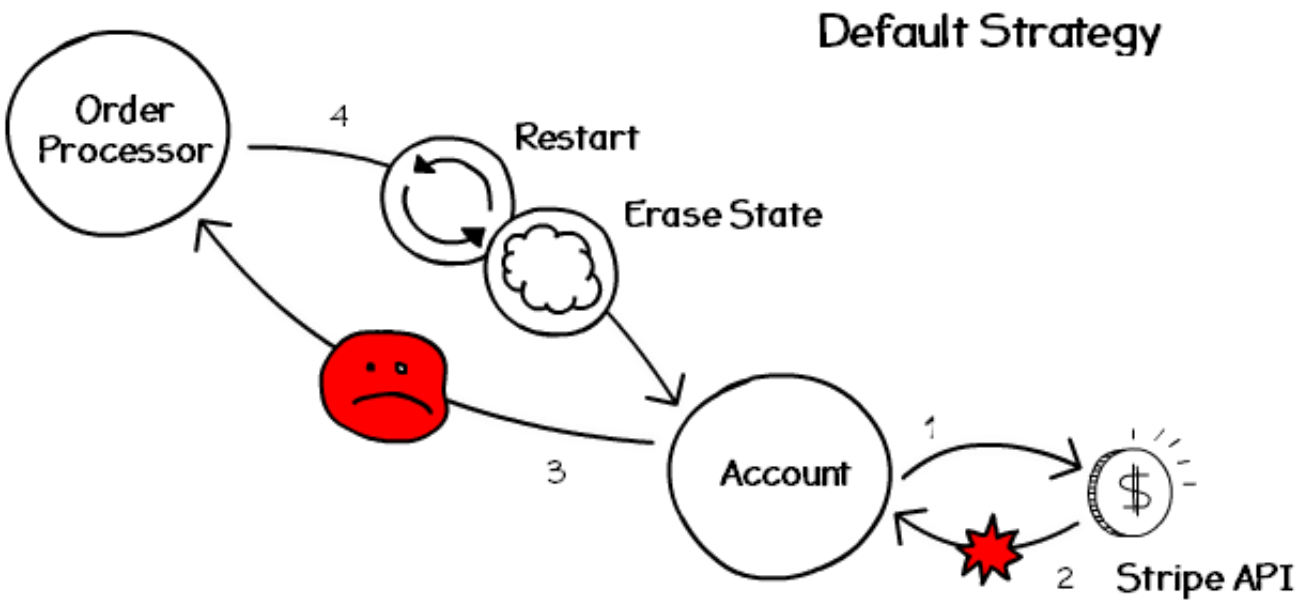
> *... supervision describes a dependency relationship between actors: the supervisor delegates tasks to subordinates and therefore must respond to their failures (Akka.NET Docs)*

There are 4 ways that a Supervisor can respond to a Child's failure:

- **Resume**: Keep state and continue processing messages
- **Restart**: Clear out internal state (**default**)
- **Stop**: Permanently terminate the child
- **Escalate**: Fail itself and tell its Parent

Default Strategy

With the default Supervision Strategy of **Restart** our AccountActor will clear its internal state, and restart once:



We can prove that the `AccountActor` was restarted by putting a log statement in the `AccountActor.PostStop()` method:

```
protected override void PostStop() {
    _logger.Warning("AccountActor stopped!");
    base.PostStop();
}
```

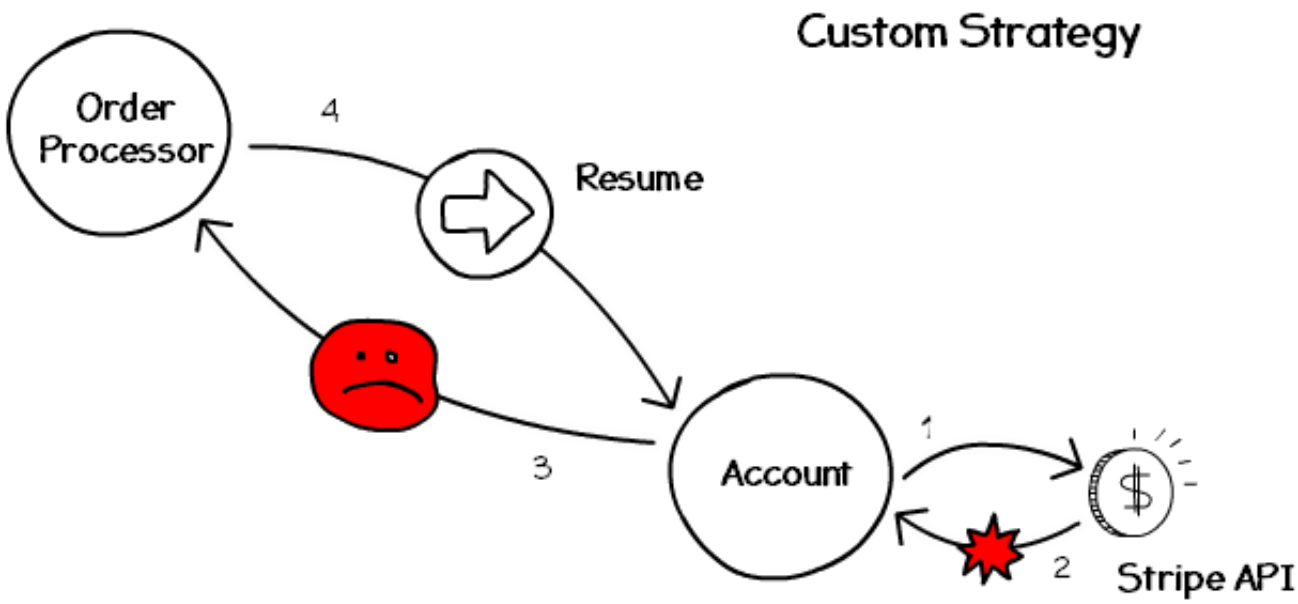
Here's what the log entry looks like when we use this Supervisor Strategy:

```
[Fact]
public void AccountActor_gets_stopping_directive() {
    var message = new ChargeCreditCard(-5000);
    var accountActor = ActorOf(
        Props.Create(() => new AccountActor(12345),
        SupervisorStrategy.DefaultStrategy));

    EventFilter.Warning("AccountActor stopped!")
        .ExpectOne(() => accountActor.Tell(message));
}

// Log Event
[WARNING][2/9/2016 8:31:11 PM][Thread 0020]
[[akka://test/user/testActor1/$$a/accountActor12345]]
 AccountActor stopped
```

Custom Supervision Strategy

What if we didn't want to just **Restart** when `AccountActor` encountered a `StripeException`?

Let's give `OrderProcessorActor` a Custom Supervisor Strategy that tells `AccountActor` to **Resume** and keep its existing **State** when it encounters a `StripeException`.



We can add this custom Supervisor Strategy to our `OrderProcessorActor` to allow children `AccountActors` to continue processing messages and keep state when a `StripeException` happens.

```
public class OrderProcessorActor : ReceiveActor {

    //... Actor Implementation

    protected override SupervisorStrategy SupervisorStrategy() {
        return new OneForOneStrategy(
            Decider.From(x => {
                if (x is StripeException) return Directive.Resume;
                return Directive.Restart;
            })
        );
    }
```

Here's a test that verifies that the `AccountActor.PostStop()` method is **not** called, therefore `_logger.Warning("AccountActor stopped!")` is **not** called, therefore the Actor has **Resumed** processing messages:

```
[Fact]
public void OrderProcessorActor_applies_custom_supervisor_strategy() {
    var message = new PlaceOrder(12345, 10, 25, -5000);
    var orderProcessorActor = ActorOfAsTestActorRef(
        () => new OrderProcessorActor(), TestActor);

    // Expect 0 Log Entries
    EventFilter.Warning("AccountActor stopped!")
        .Expect(0, () => orderProcessorActor.Tell(message));
}
```

Feel free to pull down the sample code from here and run the tests.

Summary

We began by noticing the power of the **Actor Model**, more specifically **Actor Supervision**, in languages like Erlang and F#.

Supervision allows us to use Actor Hierarchy to **isolate faults** so that errors from risky behaviors don't break our entire application. We started searching for ways to implement the Actor Model in C#...

There happens to be a few ways to use the Actor Model in C#. **Akka.NET** is a well-tested and well-documented Actor Model framework for C#. It's based on Akka for the JVM.

We finished up by implementing a real-world Actor Supervision scenario, borrowed from Vaughn Vernon. We wrote an Order Processing system in the Actor Model and tested a couple **Supervision Strategies** that were provided by Akka.NET.

Sources

- Vernon, Vaughn. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional, August 2015.
- Armstrong, Joe. Programming Erlang, 2nd Edition. Pragmatic Bookshelf, September 2013.
- Butcher, Paul. Seven Concurrency Models in Seven Weeks. Pragmatic Bookshelf, June 2014.
- Akka.NET Documentation. http://getakka.net/docs/
- Azure Service Fabric - Getting Started
- Olsson, Johan, http://www.jayway.com/. An actor model implementation in C# using TPL DataFlow
- Fred Hebert, http://ferd.ca/. The Zen of Erlang