AKKA.NET

DESIGN PATTERNS

Petabridge

Table of Contents

Petabridge Akka.NET Design & Architecture Patterns

In this Petabridge training course you will learn how to effectively model applications and common programming scenarios using Akka.NET actors!

Concepts You'll Learn

The goal of *Akka.NET Design & Architecture Patterns* is to teach you how to model applications using actors and leverage Akka.NET's modules to do that most effectively.

Thus we've structured the course to do the following:

- **Provide clear guidelines on how to model your systems using actors** - how many actor types do I need? When do I need to use a router? When should I use behavior switching and stashing?
- **Illustrate common messaging and conversation patterns** - when should I use request / response? Or publish / subscribe? Or push / pull? What's the best tool for the job?
- **Show how to use stateful programming to your advantage** - what's possible for me if my application's state becomes the single source of truth? How can I ensure consistent behavior in my distributed applications? And how can I make sure that all of my state ends up in the right place?
- **Design actors and actor interactions that are easily testable** - discover how to design actors that can work in concert with Akka.Testkit to produce reliable unit tests.
- **Integrate actors with popular dependency injection frameworks like Ninject, StructureMap, and more!** - how can I reuse my DI containers from ASP.NET and other frameworks inside my Akka.NET actors? When should and shouldn't I use them?

- **Explain how Akka.NET's built-in mechanics can be put to work for you** - How can I use the location transparency of `IActorRef` s to make my applications more testable? How can I use parent-child supervision to my advantage? How can I use the cheap overhead of actors to my advantage? What about `PipeTo` and the TPL?

After completing this course you should walk away with enough knowledge to be able to begin building effective applications on top of Akka.NET.

You'll know enough about Akka, Akka.DI, and Akka.Persistence to be able to understand their exposed capabilities - but this course doesn't cover how those modules work under the hood.

> If you want to learn how Akka.Remote and Akka.Cluster work, take our Akka.Remote training course and our Akka.Cluster training course. These will both help you really understand how distributed systems built on top of Akka.NET actually work and will help eliminate sources of DevOps issues for you in the future.

How This Course Works

Unlike Akka.NET Bootcamp, this course is not intended to be self-directed learning. This is meant to be done in real-time with an instructor in a small group.

The course works in the following way:

1. There will be a period of lecture plus presentation by the instructor in which you'll be introduced to new concepts, during which you can ask questions about the material;
2. Afterwards you'll do an interactive design exercise with the instructor, where you'll apply the concepts learned during the previous lecture;
3. Following a quick review of the exercise, the results of that exercise will be applied to a specific application the instructor will be iteratively building throughout the course;
4. Rinse and repeat until all lectures have been completed.

Once the live portion of the course is complete, you'll receive all of the code samples and a written handout that covers everything you learned in the course so you can review it all later.

Make Sure You Ask Questions!

You won't get the full value out of this course if you don't ask the instructor questions!

The instructor has budgeted 2.5-3 hours to go through all of the official Petabridge training material, but the training course is meant to be 4-4.5 hours long.

So make sure you ask questions along the way! It's your time - make sure you take full advantage of it!

What You Need to Do Before the Training

Make sure you have the following available during the training:

1. **Paper and pen** - we'll be doing some drawing and modeling exercises on paper;
2. **Visual Studio 2012/2013 and SQL Server Express 2012** - so you can run the downloadable samples after the training and make changes to the code; and
3. **A good pair of headphones and a microphone** so your questions can be clearly heard by the other attendees and the instructor. If you don't have access to a microphone you will be able to type in your questions, but *we strongly recommend* having a good microphone.

Table of Contents

Copyrighted Material

Modeling Applications with Actors

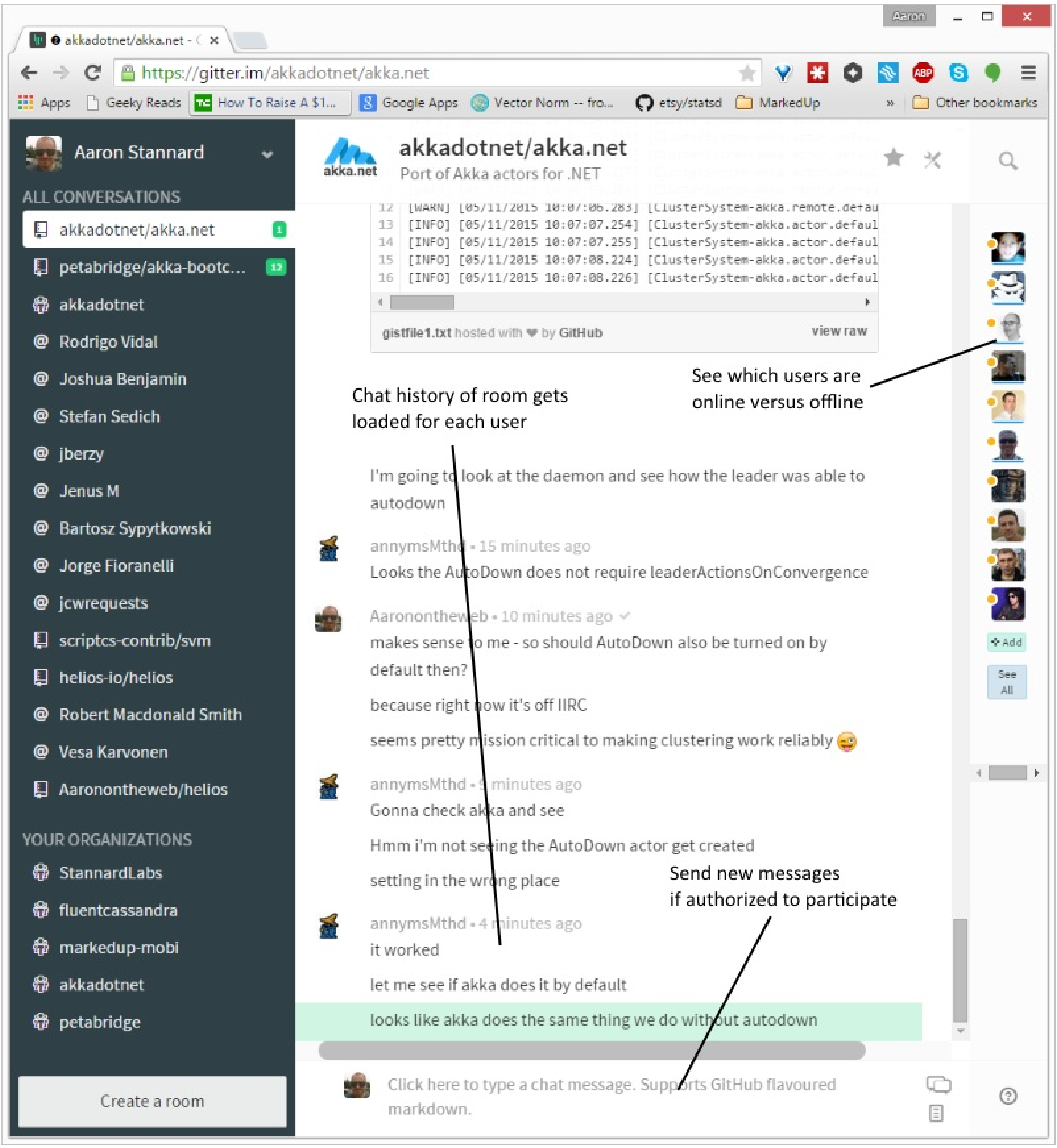
We're going to begin *Akka.NET Design & Architecture Patterns* by learning how to model applications using the actor types provided by Akka.NET!

Working Example: Persistent Chat Room

Throughout *Akka.NET Design & Architecture Patterns* we'll be working on applying what we learn to a single example application throughout the course.

That application will be a persistent chat room!

If you've hung out in the Akka.NET Gitter chat before, then you're already familiar with the concept.



```
[WARN] [05/11/2015 10:07:06.283] [ClusterSystem-akka.remote.defau
[INFO] [05/11/2015 10:07:07.254] [ClusterSystem-akka.actor.defaul
[INFO] [05/11/2015 10:07:07.255] [ClusterSystem-akka.actor.defaul
[INFO] [05/11/2015 10:07:08.224] [ClusterSystem-akka.actor.defaul
[INFO] [05/11/2015 10:07:08.226] [ClusterSystem-akka.actor.defaul
```

gistfile1.txt hosted with ♥ by GitHub view raw

I'm going to look at the daemon and see how the leader was able to autodown

annymsMthd • 15 minutes ago
Looks the AutoDown does not require leaderActionsOnConvergence

Aaronontheweb • 10 minutes ago
makes sense to me - so should AutoDown also be turned on by default then?
because right now it's off IIRC
seems pretty mission critical to making clustering work reliably 😜

annymsMthd • 9 minutes ago
Gonna check akka and see
Hmm i'm not seeing the AutoDown actor get created
setting in the wrong place

annymsMthd • 4 minutes ago
it worked
let me see if akka does it by default
looks like akka does the same thing we do without autodown

We'll be making our own implementation of this Gitter chat in Akka.NET - and we will call it "**AkkaChat**."

AkkaChat Initial Requirements

Here's the set of requirements we're going to start with:

1. The chatroom will be hosted in a web browser, so it's client-agnostic;
2. There's only one chat room;
3. All members who participate in the chatroom must register an account and login

before they can send messages;
4. We can always see the list of people who are currently visible; and
5. Chat history is persistent - the last 30 messages will always be downloaded each time a new user joins.

We will make some modifications to these requirements throughout this training course, but these are the requirements we'll be starting with. We'll refer back to these from time to time.

AkkaChat Technologies

What tools are we going to use to build AkkaChat, aside from Akka.NET of course?

- **Web Framework:** NancyFX - because it's fun;
- **Client Framework:** SignalR - to power client-push for the chatroom;
- **Persistence:** - Akka.Persistence and SQL Server!

Next Step

Now that we have an understanding for what our application requirements are going to be we can take the next step: **Breaking Up Applications into Concerns and Actor Hierarchies**.

Breaking Up Applications into Concerns and Actor Hierarchies

In the previous section we defined the functional requirements for AkkaChat. Now let's take those and start using them to create a workable design for us.

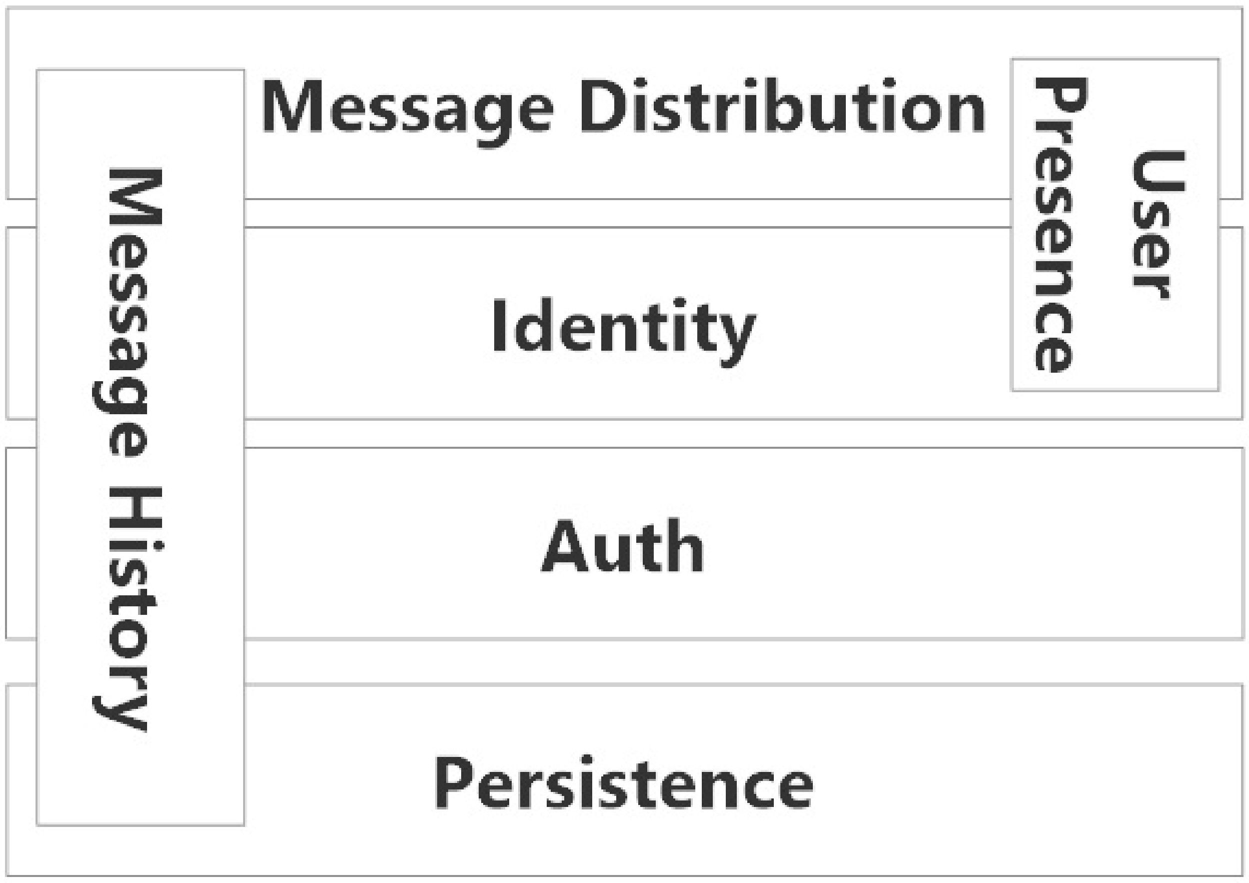
Design Concerns

The first step in designing a system that leverages the actor model isn't fundamentally different from how you'd design any other system - we use the requirements of our application to define vertical (specific) and horizontal (cross-cutting) concerns!

In the AkkaChat example we really have only a handful of concerns:

1. Authentication and authorization;
2. User identity;
3. User presence - who's active and who's not;
4. Message history - what's been said in the chat and in what order; and
5. Message distribution - making sure that the message history is actually sent to all participating users.

Here's what those concerns might look like, illustrated:



So far we can see that **message history** - which includes the sending of new messages, basically depends on everything and that **user presence** depends on message distribution and identity.

This is a good start, but this modeling leaves us with a bunch more questions! For instance:

- Why is persistence treated like a design concern at all? Is that really a part of our application?
- Doesn't authentication and authorization actually depend on identity?
- Doesn't message distribution actually depend on a bunch of stuff that's not even on the screen at the moment?

These questions are the basis of the next part of the exercise - getting specific about the separation of concerns!

Concerns as Conversations

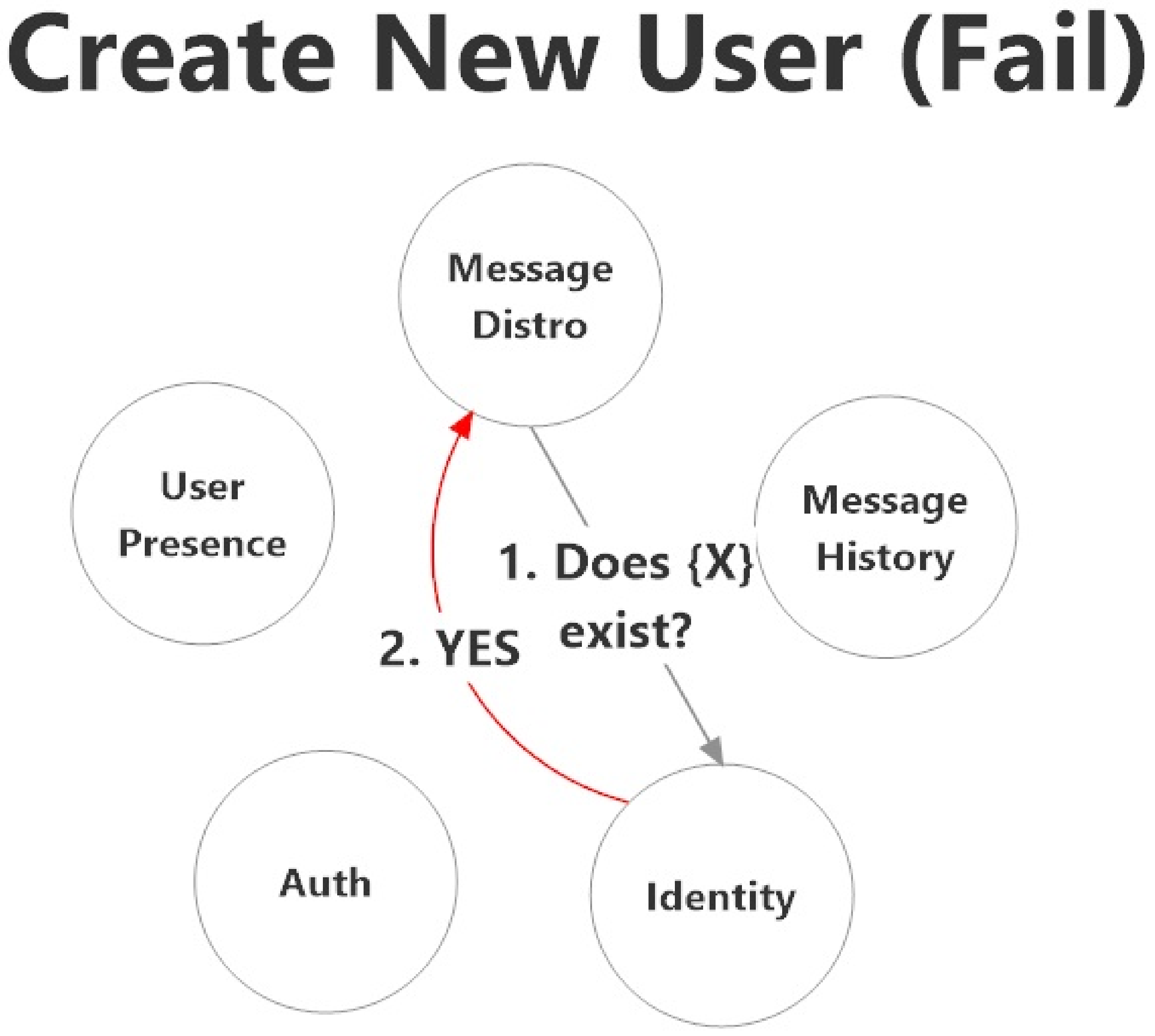
The fundamental problem with the layered we mapped our concerns earlier is that it pushes a square peg into a round hole - in reality our interactions between layers often don't follow a nice linear order of dependencies.

So let's try a different approach to modeling our concerns - **let's model the interactions and *not* the dependencies**!

The first scenario we'll model is the following:

> "What happens if someone tries to register a new user account that already exists?"

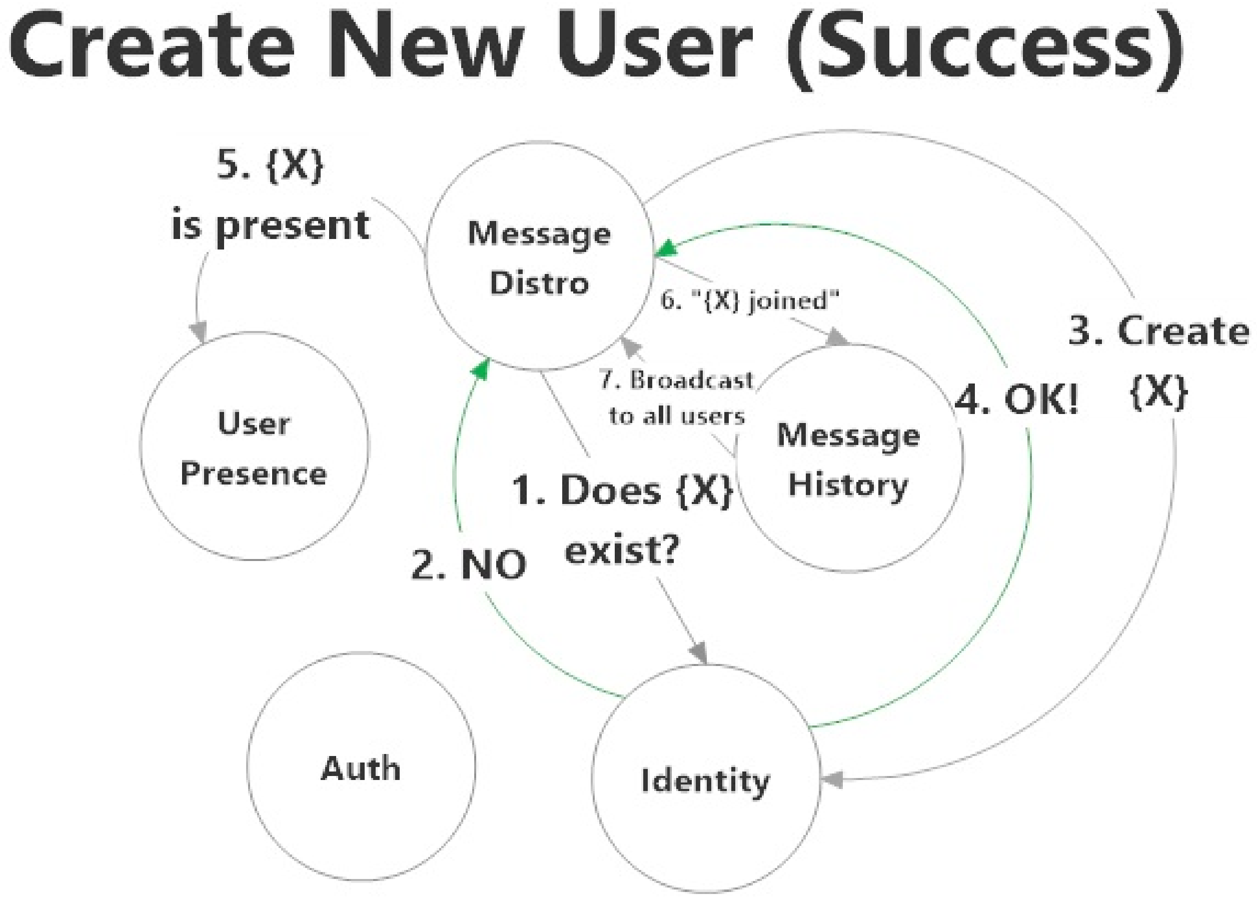
Here's how that interaction might look:



Notice how none of the concerns are presented as layers - just different components having a conversation about the interaction.

> NOTE: Message Distribution in this case is just a conduit between SignalR / Nancy and the rest of your application. All user requests originate and end there.

Now let's consider what the "create new user" interactions might look like in the success case!



This diagram is *much* busier! But what can you start to see here? It sure seems like *there's a lot more going on inside the message distro layer than what we can currently see.*

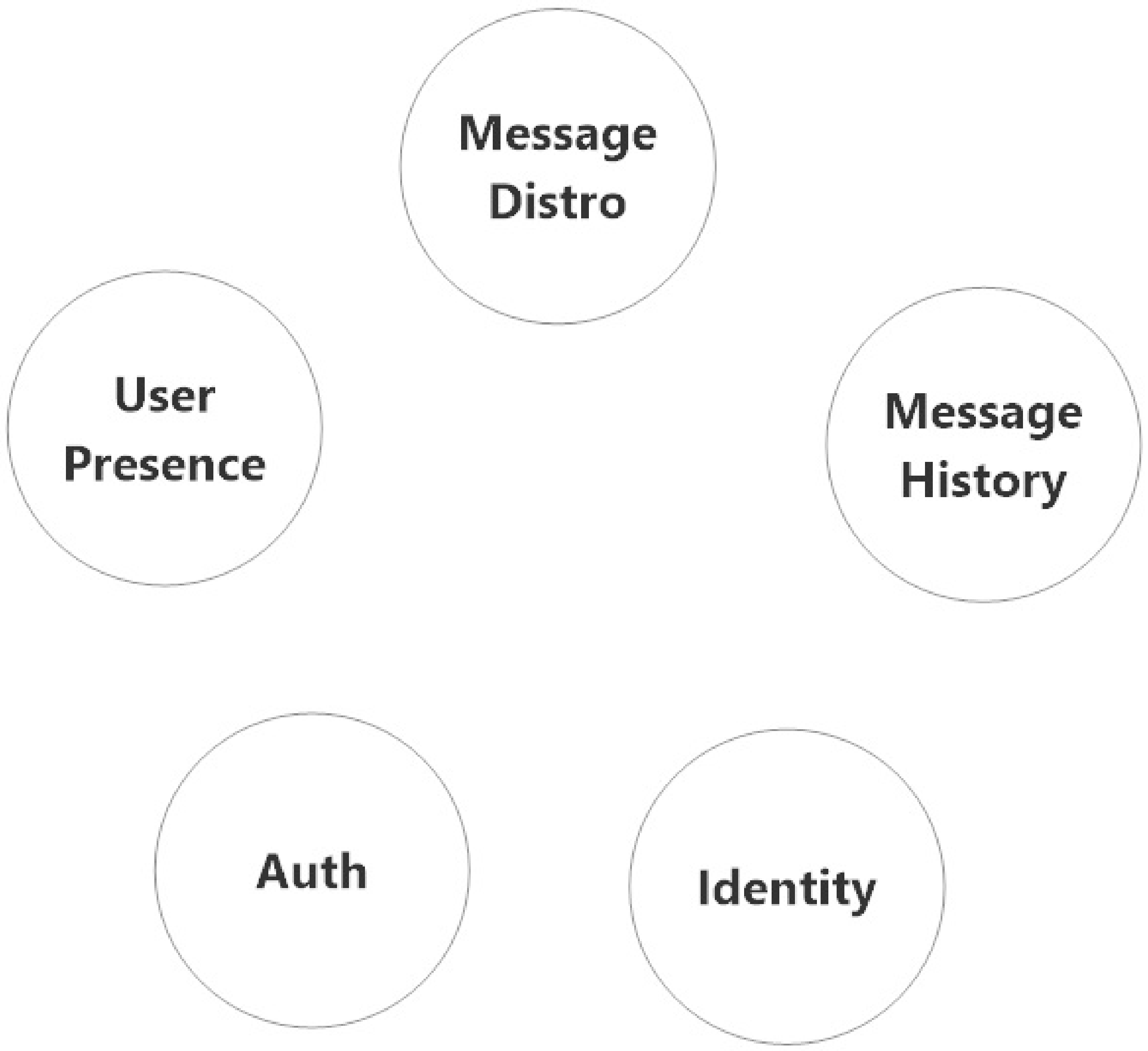
There's some interactions that involve an unauthenticated user attempting to create an account, doing some validation back-and-forth, and some status and presence messages that get pushed out to all of the other users once the user gets registered.

We can also make an educated inference here and assume that after this interaction completes, there will probably be another one that involves authorizing the newly created user to send messages to the channel.

Exercise: Modeling Conversations between Concerns

Given what you've just seen, try this out yourself and see if it fits.

Use the following blank diagram:



And model the following conversations:

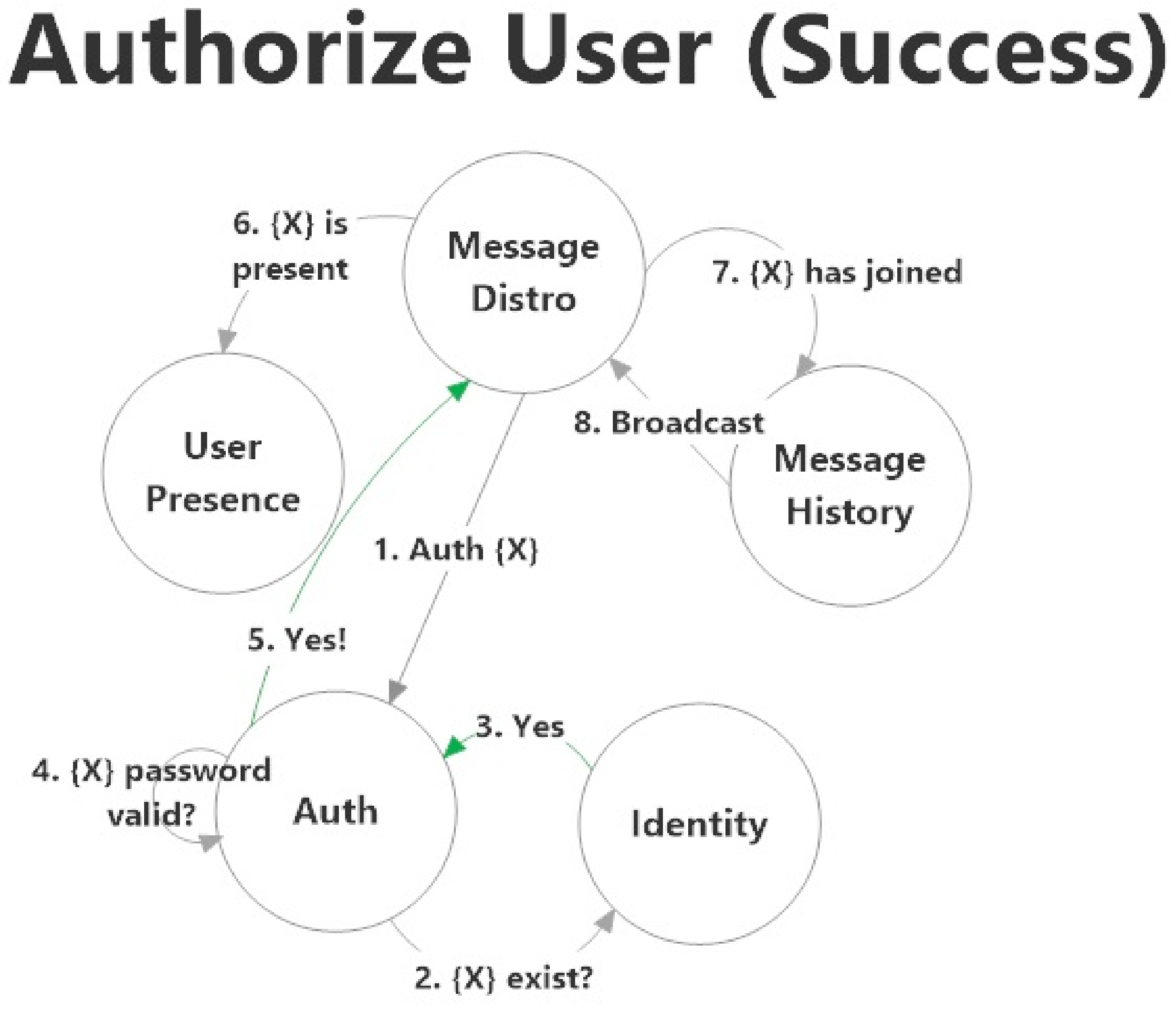
1. Authorizing an existing user;
2. Authorized user sends message to the chatroom;
3. Retrieving the message history for someone who's just loaded the chatroom in their browser.

Take a few minutes to do these on your own - and then we'll go through the results together.

Exercise Results

Here's what the instructors came up with for the three exercises!

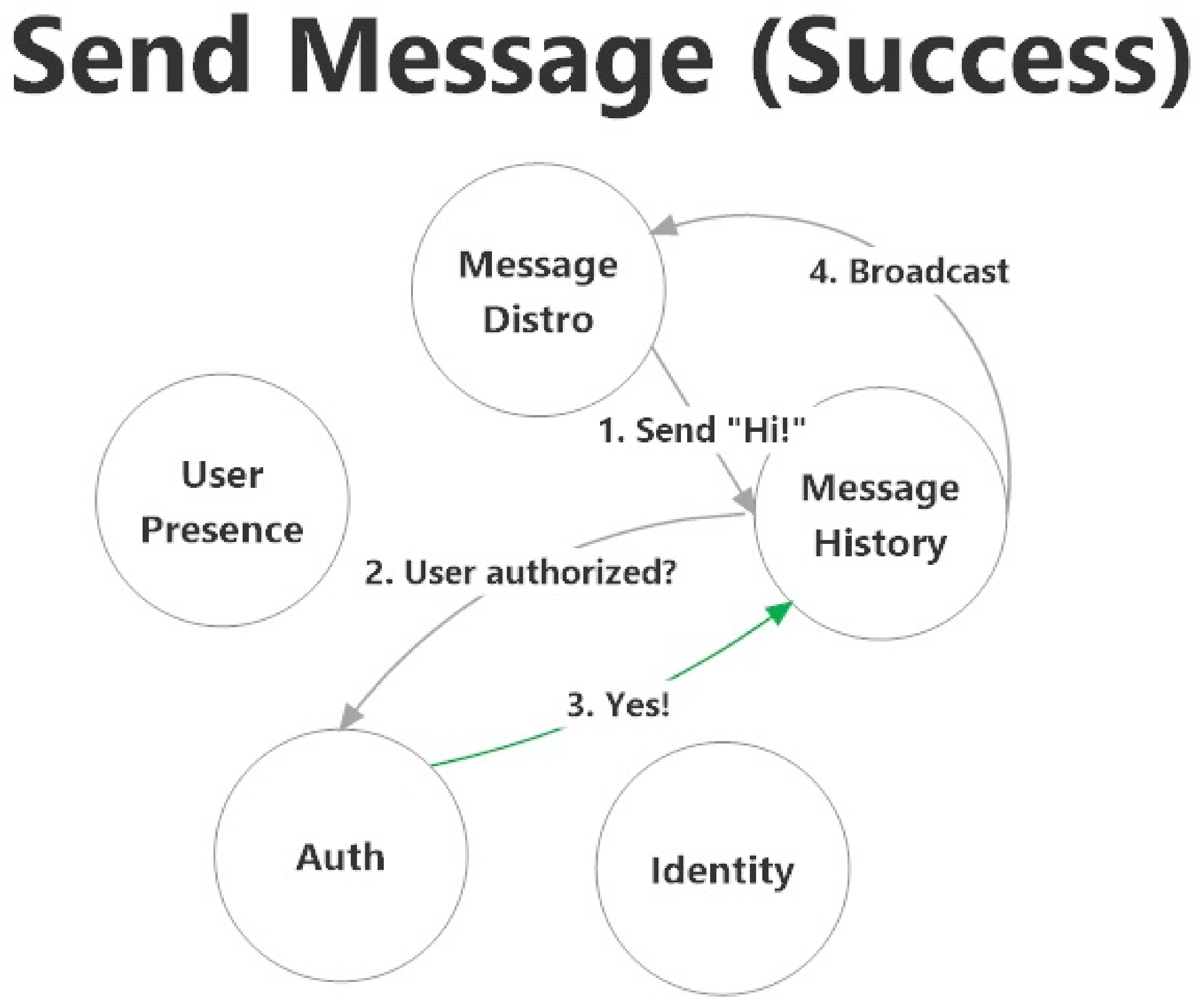
Authorizing an existing user



The steps here aren't terribly different from the steps used to create a brand new user - the only differences are really in where the conversation begins: between the message distribution layer and the authentication and authorization layer.

Aside from the initial interaction, everything else pretty much follows the same conversational pattern as before.

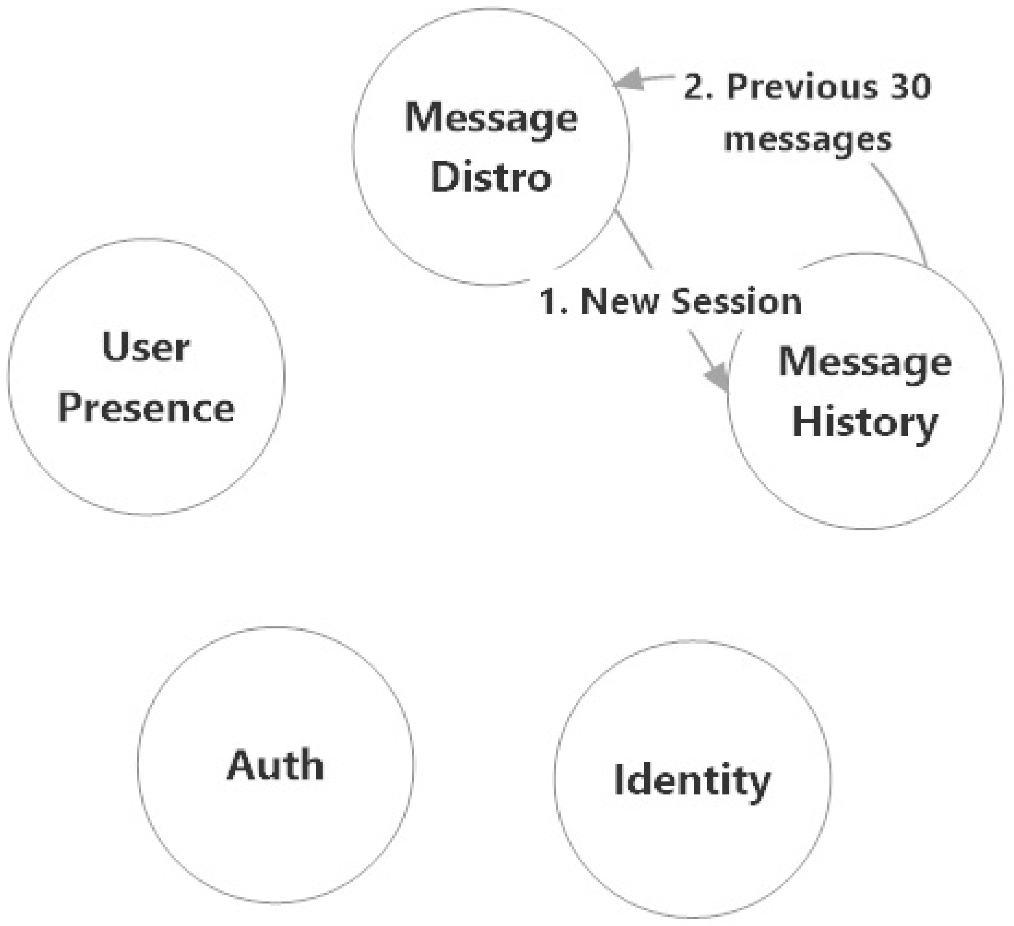
Authorized user sends message to chatroom



The only thing important to point out here is that the message history concern verifies that the user is authorized before the history gets updated and broadcast to all of the chatroom members.

Retrieving the message history for new user session

Fetch Message History



Wow, this one is *really* simple! All of the data is right there in the message history concern!

Critical Questions

After doing this exercise and the previous lecture, here are some questions you might be asking:

1. What happened to the `Persistence` layer?
2. Seems like there's a lot going on inside `Message Distribution`, `Message History`, and some of the other layers - what's underneath the surface?
3. What do the interactions between all of these layers look like? Certainly seems like there's multiple types of interactions even between the same two layers.

Keep these in mind! We're going to try to answer them by modeling our concerns explicitly as actors!

Model Conversations with Actors

So now we're going to turn our attention to actors and hierarchies explicitly... But what was one thing you might have noticed in the previous section?

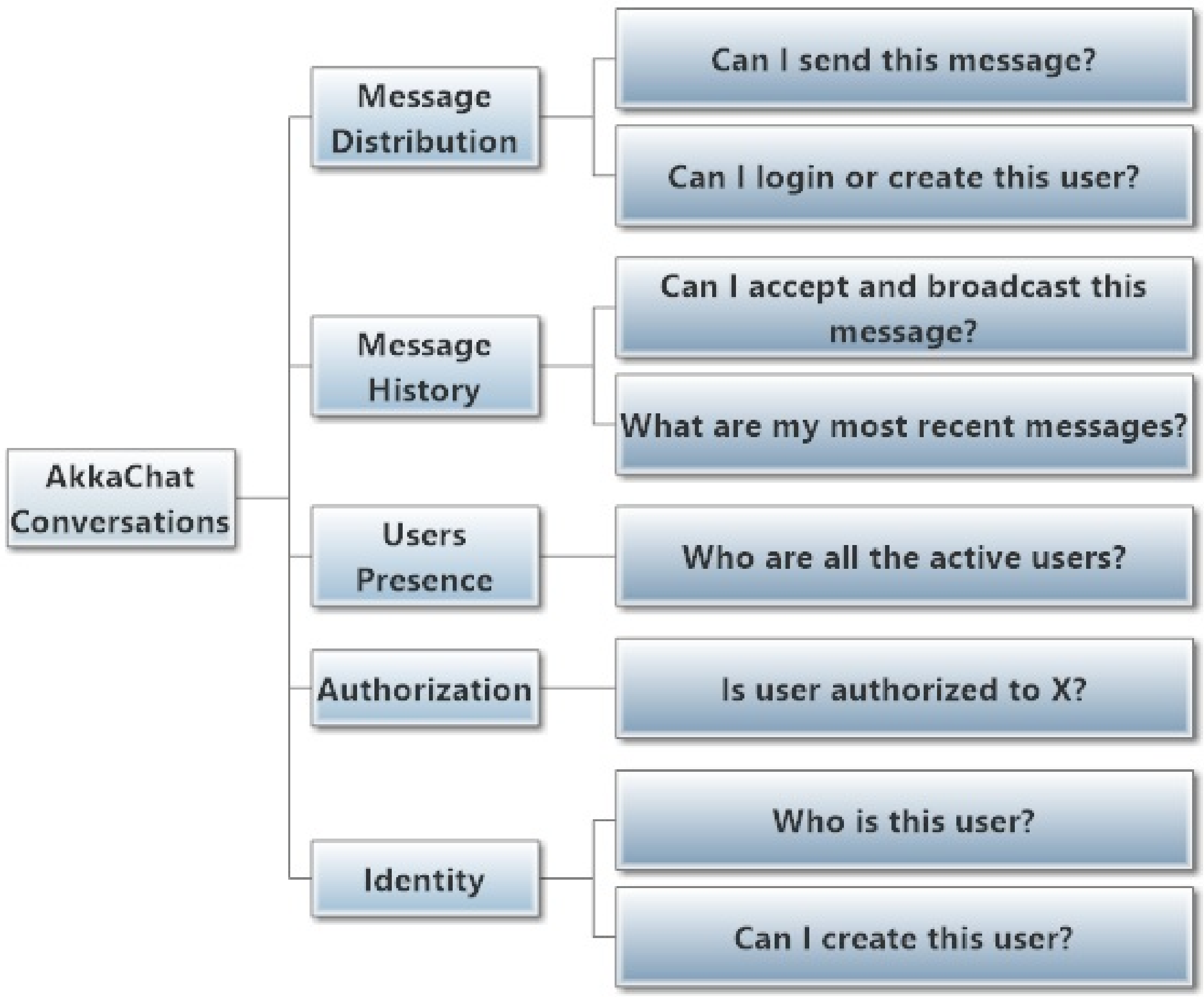
> Those "conversations" between concerns *sure looked like conversations* between actors - didn't they?

Well that's exactly right! And we can use these conversations as the basis of our actor model for AkkaChat.

Finding Dependencies with Questions

The first step in being able to model our system with actors isn't to think about which actors we need - it's to make some distinctions about **which questions are really being addressed within each conversation?**

Here's an example of what that might look like:



If we take a close look at those conversations, written as questions, we can see some common patterns:

1. There needs to be a common way for actors involved in the sending of messages to know if they're authorized to commit the send or not.
2. There needs to be some central actors responsible for determining whether or not a user is currently active **and** when they stop being active.
3. The `Message Distribution` concern effectively acts like a user interface - it doesn't need to be bothered with the details for every specific user interaction, but it needs to know how to ask the other appropriate actors for permission to execute some command.
4. The `User Presence` and `Authorization` concerns are both going to need to ask the `Identity` concern for information periodically. This indicates that the `Identity` concern is probably stateful.

This is enough information for us to attempt to model something using actors.

Actors as Participants in a Conversation

Let's take one of the specific conversations from our previous diagram and explore how we can break that down into actors.

> **Authorization: Is user authorized to do X?**

What do we really need to know to be able to participate in this conversation?

1. Does this user actually exist?
2. Is this user who they claim to be?
3. Does this user have the rights to do whatever it is they need to do?

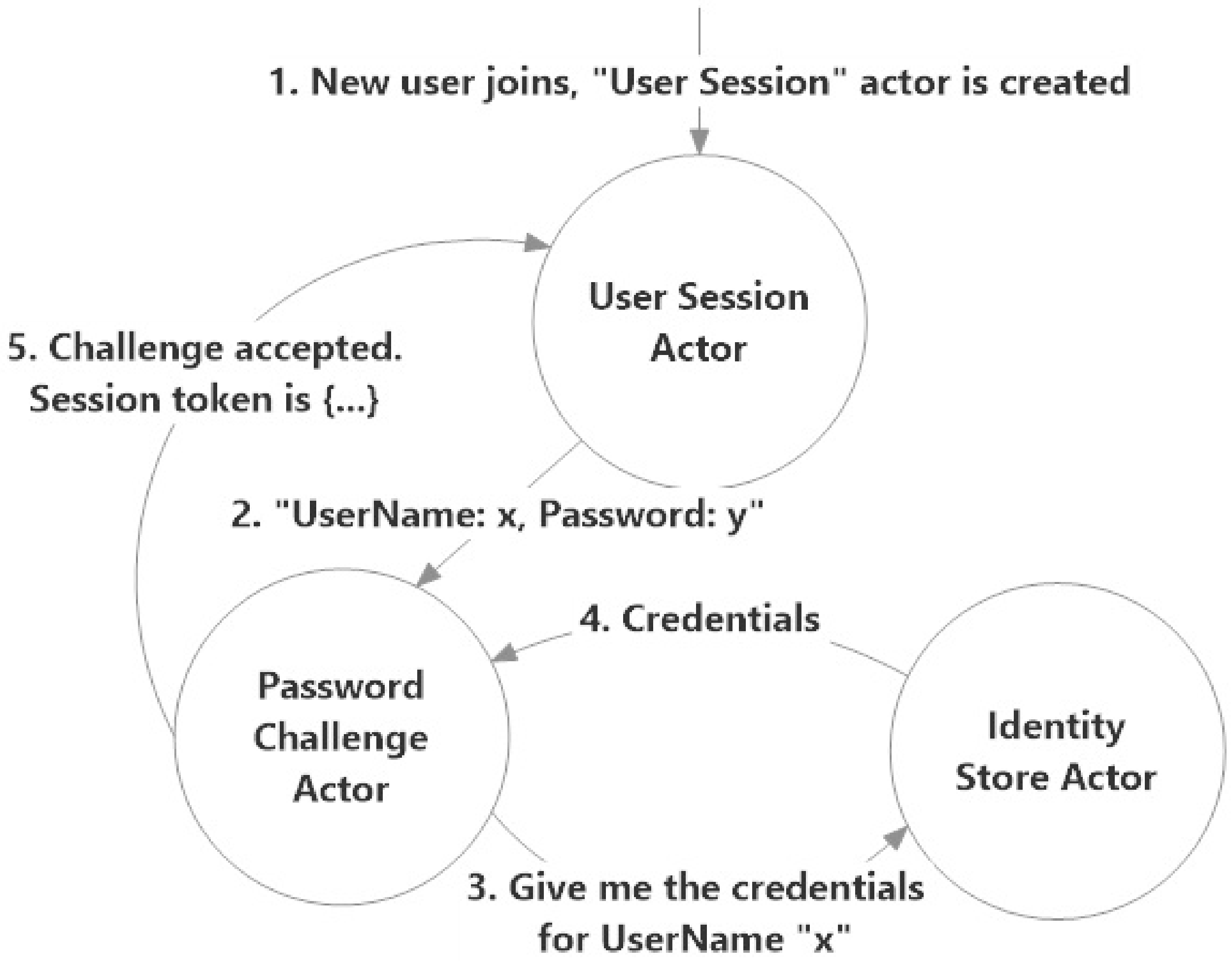
In this case question #3 is irrelevant - the only thing a user needs permission to do is write to the chat room, so there's no need to make a more granular permissions system than that.

Separation of Concerns

But we can see a clear separation of concerns within this conversation though:

- **one party needs to retain the identity and credentials of the user**,
- **another needs to be able to test the provided credentials** against the stored credentials to authenticate the user, and
- **a third party needs to retain some proof of the successful authentication** challenge for the duration of the user's session, so they don't have to keep logging in over and over again.

This should give us enough information to model this conversation as an interaction between three actors!


1. New user joins, "User Session" actor is created
User Session Actor
5. Challenge accepted. Session token is {...}
2. "UserName: x, Password: y"
4. Credentials
Password Challenge Actor
Identity Store Actor
3. Give me the credentials for UserName "x"

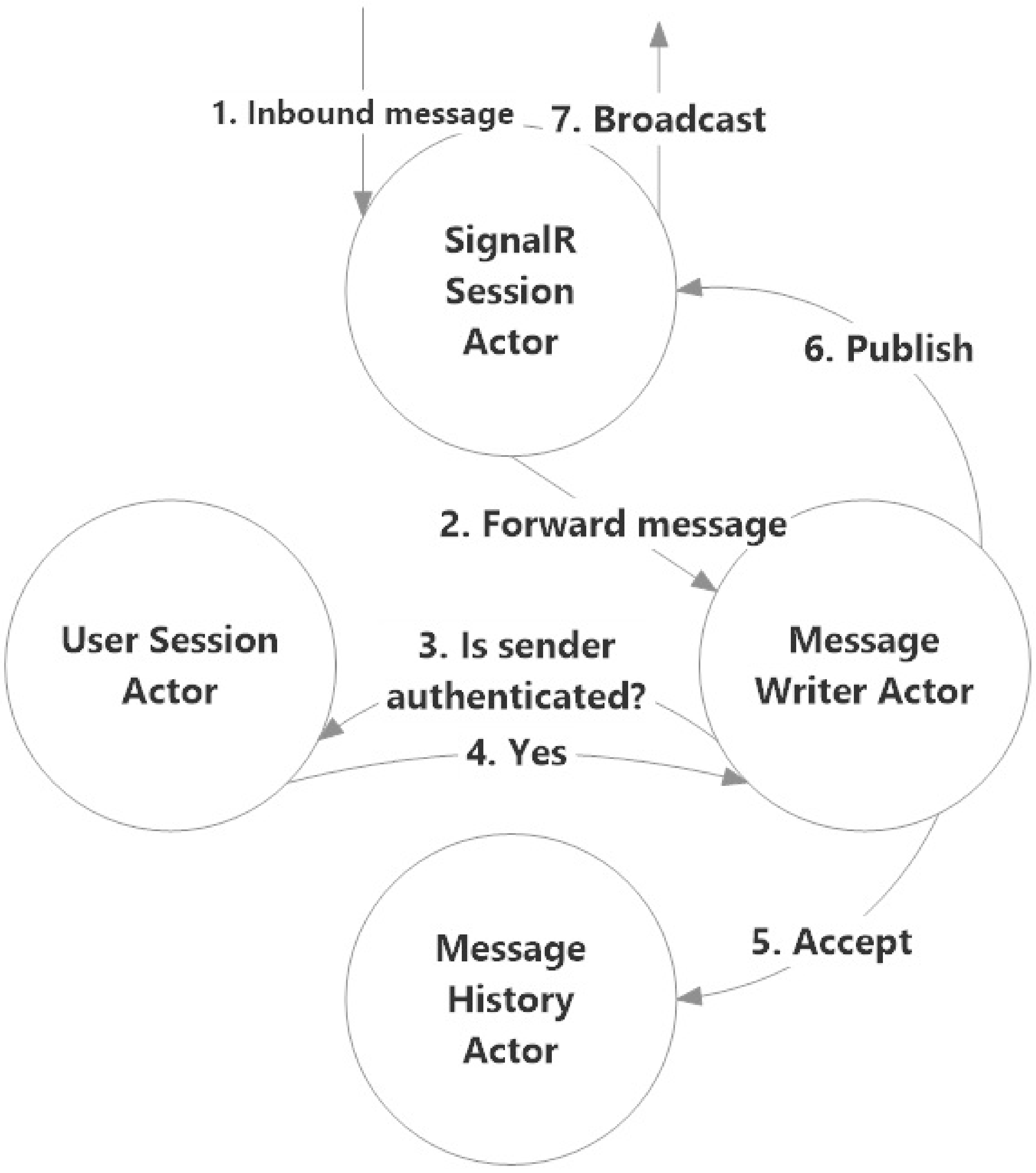
Awesome! So why is this a good design? Well, to answer this question - let's take a look at another workflow related to this one.

> **Message history: can I accept and broadcast this message?**

So what needs to happen in this scenario?

- **A message needs to be propagated from SignalR** / "the front-end" to an actor responsible for its delivery;
- **This message needs to be validated** - specifically we need to verify the identity and permissions of the author;
- **The message needs to be added to the message history**; and
- **The message needs to be broadcast** back to all of the other users in the chat.

This conversation might look like the following:



Now we can see where the `UserSessionActor` might need to participate in multiple conversations within AkkaChat, and recognizing these conversation flows throughout your domains is the framework we'll use for modeling and designing our actors.

Building Effective Actor Hierarchies

So far we've shown you some examples of breaking up your applications up into distinct concerns, breaking those concerns down into interactions, and then modeling those interactions as actor conversations. This is all fine and good, but we need to learn how to take this information and turn it into a tangible actor hierarchy that you can actually use.

The Differences between Modeling with .NET Classes and Modeling with Actors

Before we dive into how to do put the hierarchy together, let's take a moment to review some key distinctions and differences between how we'd model a traditional .NET application using C# classes from an Akka.NET system using actors.

Actors are micro-processes - they're alive!

Unlike C# classes, actors have a distinct life cycle. They get created, process some messages, restart if they fail, and eventually shutdown.

We can use this knowledge to design actors that are alive for specific periods of time.

For instance, you can model an actor that's design to be alive only as long as a specific user is having active sessions on AkkaChat!

Or, you can use the Akka.NET `IScheduler` to create an actor that runs a special job once every 30 minutes and then goes back to sleep.

You can also design actors that self-terminate and garbage collect themselves plus their children when they're out of work to do. Or you can also design them to reboot themselves in the event of an unexpected error.

The fact that actors have a lifecycle is a massive difference and ultimately helps eliminate lots of code from your applications.

Actors are always thread-safe

If you follow the design constraints you learned while going through Akka.NET Bootcamp, you'll remember that if your message types are always immutable then the state of an actor is always inherently thread-safe.

This means that you can safely use `IActorRef`s (actor references) in static contexts across your ASP.NET MVC controllers, NServiceBus subscribers, and more. You can pass around actor references much more freely than you ever could with shared objects.

Actors exist independent of other contexts

As long as your `ActorSystem` is up and running within your application, your actors will always be available for work regardless of the other contexts present in your application. This is why you can safely combine Akka.NET actors with ASP.NET MVC, NancyFX, WCF, WPF, and any other application run time.

The same cannot be said for .NET objects - if you create an actor instance inside an ASP.NET MVC request, that actor will still be present even after the request completes unless you explicitly terminated it. Whereas a traditional .NET object would usually be garbage collected along with the parent request object.

Actors have location transparency

Akka.NET actors have a property called location transparency which allows actors that are deployed in your local process to look exactly the same as actors deployed on a remote machine in a data center on the other side of the country.

Learning how to leverage this property early will make it *hilariously* easy to refactor monolithic applications into microservices later on, and we'll show you how to do that as part of this course.

Actors are message-oriented and thus, always asynchronous

Everything an actor does is asynchronous, unlike method calls on .NET objects. Appreciating this early on in your design will allow you to take full advantage of actors' ability to rapidly switch contexts and do other types of work while waiting for responses to requests.

Actors have explicit relationships with parents and children

Unlike real-life for us humans, actors *always* know where their children are and can *always* get a hold of their parents.

.NET objects don't have any innate relationships to any other objects, other than what's defined in each individual class. Not true for actors - *all* actors have parents and many of them have children.

Actors can accept any message, but not necessarily handle all of them

One of the really interesting properties of actors in Akka.NET today is that they're all untyped - meaning that they can *accept any type of message you send them.* This makes is tremendously easy to refactor an actor to adopt new behaviors and new message contracts in the future!

However, the cost of this is that you don't know upfront whether or not an actor can handle a given type of message. And if an actor gets killed while its processing a message, you may not know whether or not your request has been handled or not.

This is why we'll be using the `GuaranteedDeliveryActor` (soon to be renamed to `AtLeastOnceDeliveryActor`) from Akka.Persistence in some key areas of AkkaChat.

Actors control the flow of message processing

Each actor only processes one message at a time, except for the built-in routers, so this means that actors have the ability to make decisions about how they're going to handle future messages.

Effectively an actor can defer processing of specific messages until it achieves some state or receives some notification from another actor that it's ok to keep going again.

This is a powerful flow control tool that makes it really easy to abstract details away from other actors, which can further atomize your code base.

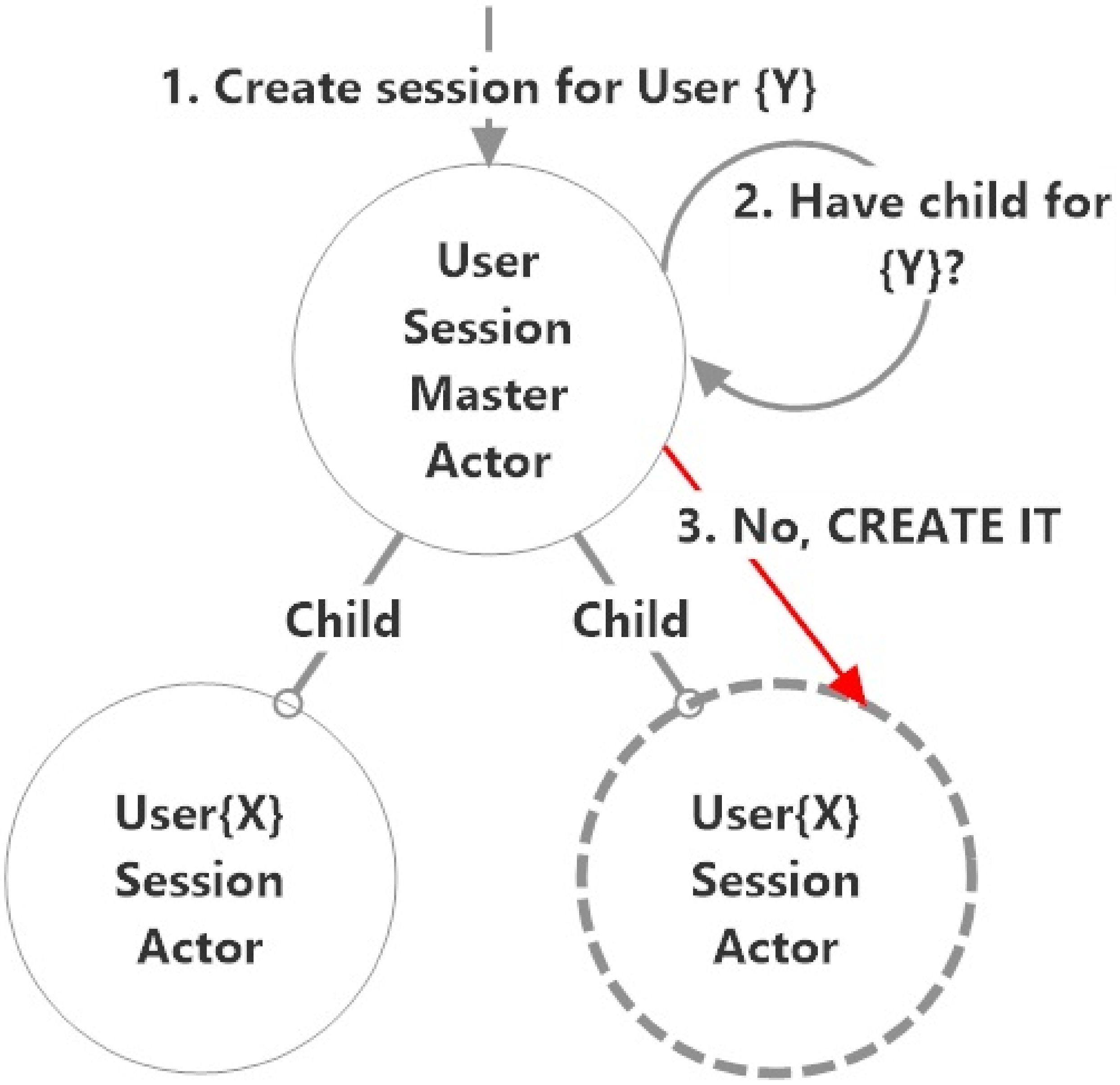
Leveraging the Unique Properties of Actors in Hierarchies

Given these differences, how can we use them to model our applications more effectively with actors?

Parents have direct access to their children, and vice-versa

One of the most important things to remember about actors is that the cost of looking up a parent or a child is virtually zero. Parents already know if their children exist and vice-versa.

So imagine if we needed to create a `UserSessionActor` for every single concurrent user in one of our AkkaChat sessions? What might be an effective way of modeling that on the hierarchy?



We can effectively use the parent's collection of children as a lookup table for user sessions. That's real power that we've used at the scale of hundreds of thousands of parallel actors in production.

> **Pattern: Parent-child lookup and Create-if-not-exists!**

This way of using the actor hierarchy, as illustrated above, is also an effective way of enforcing "create if not exists" lifecycles for your actors.

Parents can (and should) delegate risky operations to their children

One of the patterns we'll be referring to by name throughout this course is the Error kernel pattern, an Erlang programming pattern which basically means "delegate dangerous stuff to your children and see if they die or not."

> **Pattern: Error kernel pattern!**

Using this pattern in combination with custom `SupervisorStrategy` implementations can be a very effective way of providing robust error handling throughout your application. Deal with errors and exceptions locally rather than letting them cascade further up throughout your application.

Actor classes can be deployed *anywhere* in the hierarchy

Suppose you use the *Character Actor pattern* to create a purpose-built actor who's only job is to save stuff to SQL Server using a specific repository you created. You can create an instance of that actor *anywhere* in your actor hierarchy!

> **Pattern: Character Actor pattern!**

The parent / child relationships of any actor are totally decoupled from the details of that actor's implementation, so you can use this to place specific types of utility actors in lots of different places throughout your hierarchy if needed.

Akka.NET's built-in router actors are a good example of this - these are special actors needed for high-throughput work distribution scenarios and they're often used simultaneously throughout many different places inside a single Akka.NET application.

Switchable behavior, FSMs, and Stashing can be used instead of more actors

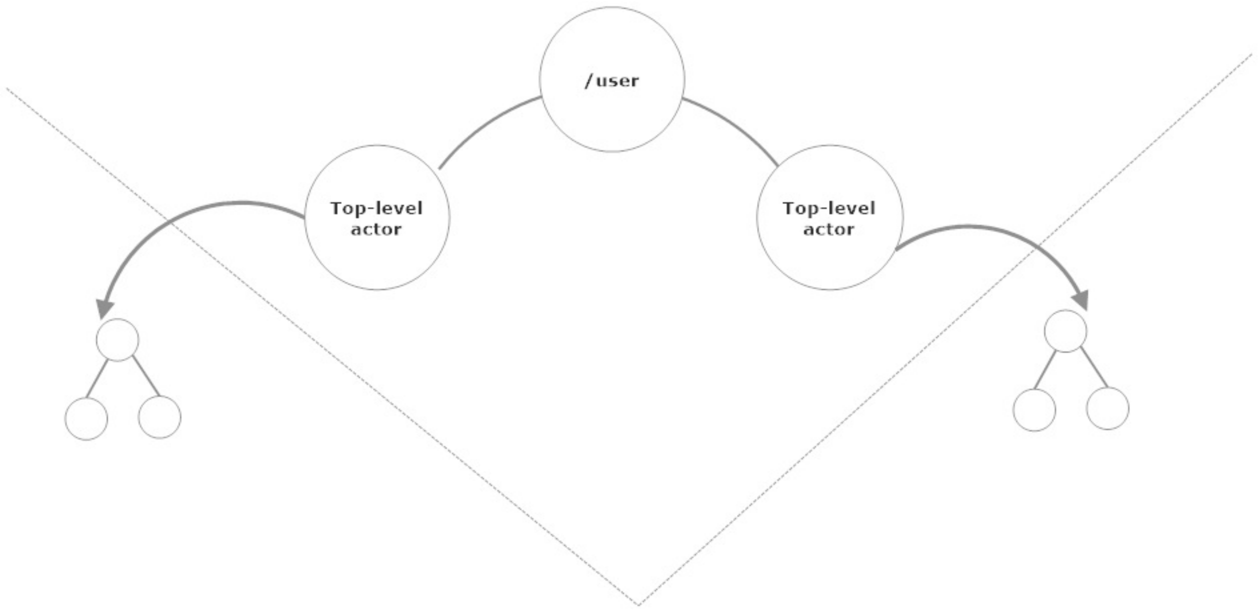
For context-switching scenarios such as changing a user from an unauthenticated user to an authenticated user, you can use Actor behavior-switching in combination with stashing messages to allow a single actor to take on multiple behaviors within the context of a single responsibility.

> **Pattern: Finite State Machine (FSM) pattern!**

This can allow you to control the flow of many different types of conversations using a small number of actors.

Actors should be desgined with no / limited knowledge about their cousins in the hierarchy

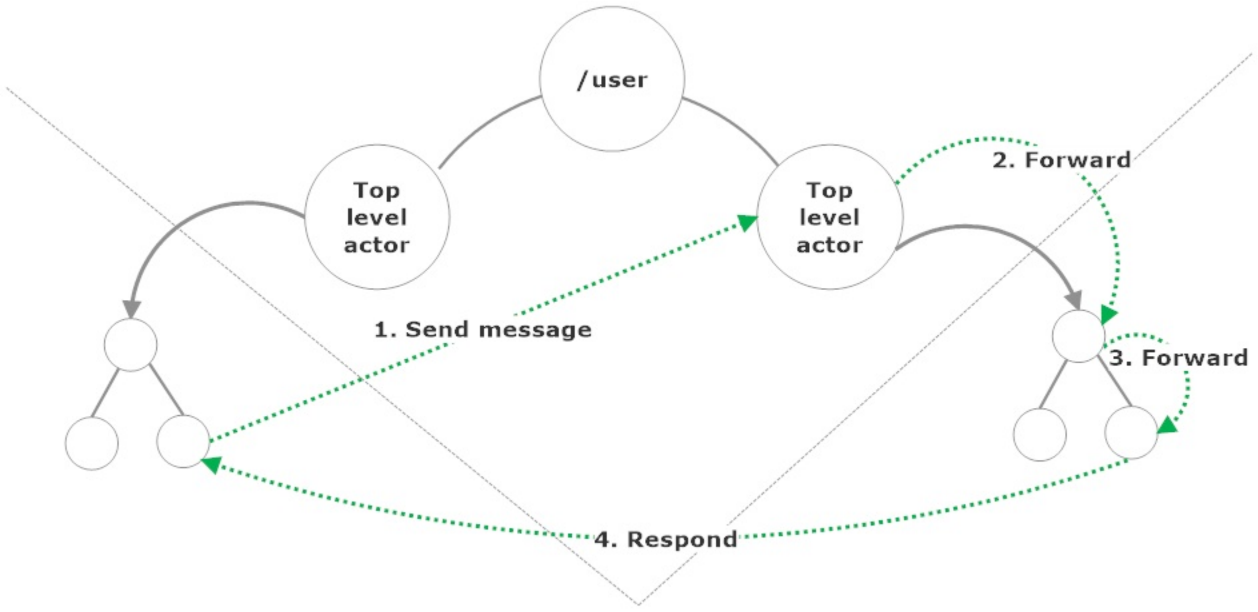
An extremely important design constraint that's used throughout actor-based systems is limiting the knowledge about the existence of "cousin actors" in the hierarchy.



The children and grand-children of top-level actors in this actor hierarchy are programmed to be unaware of the existence of any children or grand-childern of other top-level actors. This is intentional, because it loosely couples the two actor hierarchies and allows for greater flexibility when it comes to refactoring and taking advantage of location transparency.

Communicate through the top of the hierarchy

Building on the previous section about limiting knowledge about the hierarchy, in order for children and grand-children to work with other parts of the actor hierarchy then we must pass our messages through the part of the hierarchy those actors know about: the top-level actors!



This is why top-level actors are often created as routers - they're a source for initial message flow for many conversations that occur between actors.

However, after the initial message has been routed to the appropriate actor, children and grand-children can communicate with eachother directly through their `IActorRef`s if they're designed to reply to `Sender`. This is how you can put the entire actor hierarchy to work without strongly coupling it at all layers.

It's easy to pass top-level actors to external contexts

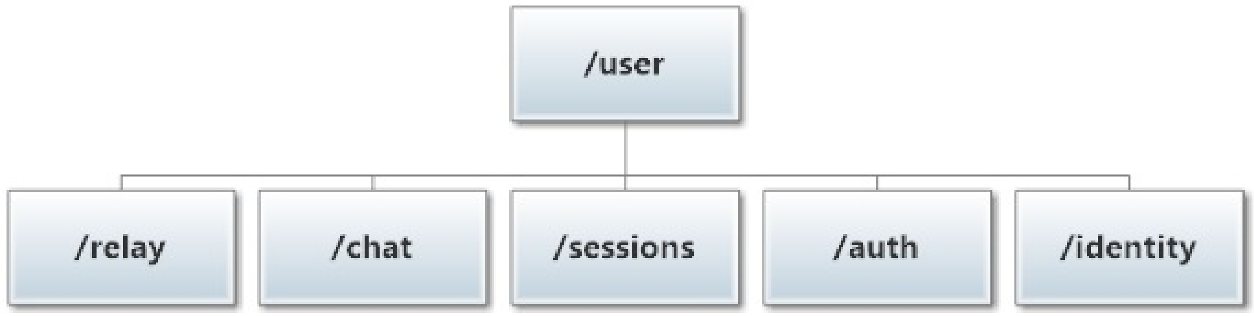
A "top-level" actor refers to the user-defined actors at the top of your hierarchy. These actors are typically created at application startup and live for a long time, and references to them are often stored inside a `static` class.

Therefore, these top-level actors are often really easy to pass around to external contexts such as ASP.NET MVC controllers and so forth. Think of them as possible touchpoints between your actor system and the rest of your application.

With this in mind, let's create our first actor hierarchy for AkkaChat!

Organizing Hierarchies

First things first - let's set aside a top-level actor for each of our primary areas of concern. Our final hierarchy may not end up looking that way, but it's a convenient place to start.



We've arranged for five top-level actors thus far - each one owning an area of responsibility.

- `/relay` - interacts directly with SignalR and the Nancy front-end. Message conduit.
- `/chat` - makes changes to the chat history and determines whether or not to accept new messages from users.
- `/sessions` - determines which authenticated users are actively participating in this chat.
- `/auth` - authenticates and authorizes user sessions.
- `/identity` - creates and identifies users.

This is a good place to start - but how do we determine where else to go from here?

Exercise: Expand the Actor Hierarchy

You've seen some examples now of how we've been able to create interaction diagrams between different areas of concern and model those as actor conversations. Now it's time to use your imagination and give this a try for yourself!

How might you fill out the rest of the actor heirarchy organization chart given the current functional requirements for AkkaChat?

Just as a reminder, here's what those functional requirements are:

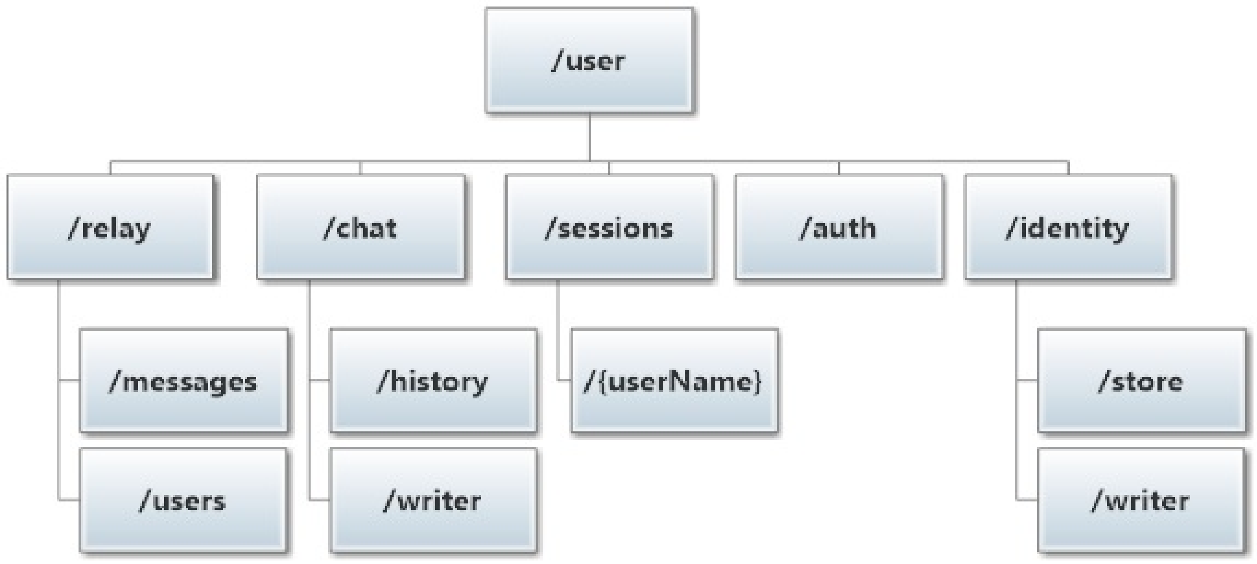
> 1. The chatroom will be hosted in a web browser, so it's client-agnostic;
> 2. There's only one chat room;
> 3. All members who participate in the chatroom must register an account and login before they can send messages;
> 4. We can always see the list of people who are currently visible; and
> 5. Chat history is persistent - the last 30 messages will always be downloaded

> each time a new user joins.

Put together a diagram on paper and we'll review the results together.

Exercise Results

Let's see what the instructions came up with and compare!



So what did we do here?

1. The children of `/relay`, `/messages` and `/users` represent conduits for dealing with two different type of user interface concerns: reading and writing of messages and creating / authenticating users.
2. The children of `/chat` - since we only have one chatroom we have a single actor dedicated to acting as the single source of truth for the history of the chatroom (`/history`) and another actor who has the risky job of negotiating writes (`/writer`.)
3. The children of the `/sessions` actor represent all of the individual sessions for each connected user.
4. `/auth` has no children - in this model it's able to perform authentication challenges in a stateless manner. That may change in a subsequent design.
5. `/identity` uses the same pattern that `/chat` does - has one actor responsible for saving users and another for writing changes to that store.

The reasons why we've broken them up this way will become clear as the course goes on, but all of them are related to some of the unique properties of actors we listed earlier.

Next Step

Now that we have a general modeling process for working with actors, lets explore some specific **Actor Composition Patterns.**

Actor Composition Patterns

In the previous section we learned about the high-level process for modeling applications using actors. In this section we're going to learn some patterns for creating groups of interrelated actors in order to accomplish specific goals.

Here's the list of patterns we're going to tackle in this section:

- Child-per-Entity Pattern
- Fan-out Pattern
- Parent Proxy Pattern

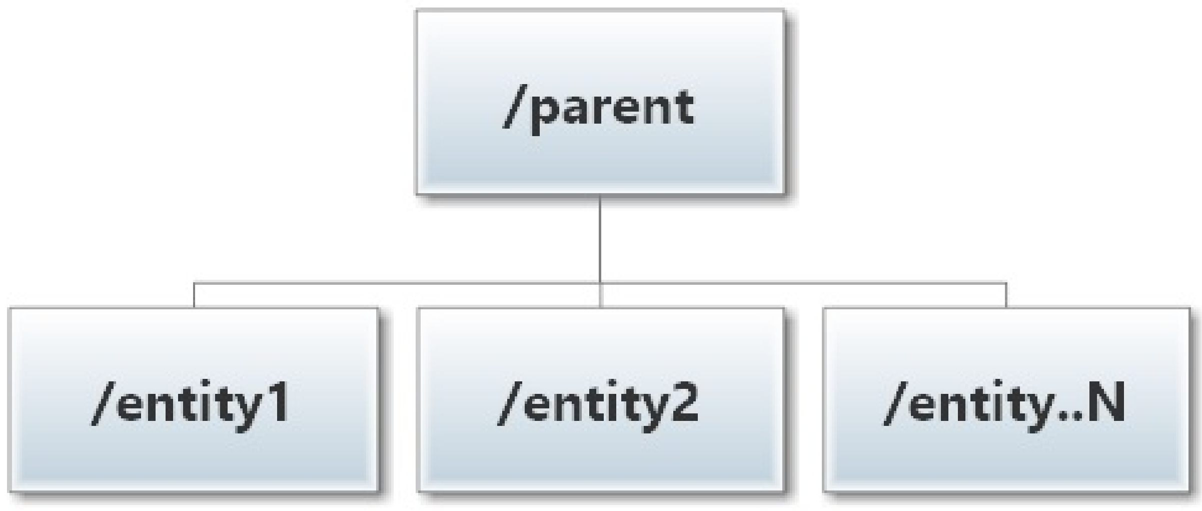
These patterns are aimed at allowing you to use Akka's supervision hierarchy to your advantage in order to achieve maximum reliability and transparency when working with actors.

Child-per-Entity Pattern

In this scenario we're going to take a look at the `UserSessionManager` actor built into AkkaChat, as it implements this pattern in a particularly clever set of circumstances.

But first, let's define what the *Child-per-Entity Pattern* actually is.

Definition: The Child-per-Entity pattern is an instance where some `/parent` actor is responsible for a domain of entities and elects to represent of each entity instance with its own unique actor. The parent actor maintains an mapping to know exactly which actor corresponds with entity instance and will subsequently create new children or kill existing ones each time a domain object enters or leaves `/parent` 's going concerns.



Creates 1 child per domain entity.

Use Cases: The Child-per-Entity pattern is most useful in scenarios where each represented entity closely resembles the real-time state of another concurrent entity. Some examples include:

1. Stream processing and aggregation / accumulation;
2. Real-time user or device activity;
3. Authentication sessions; and
4. Long-running, but real-time transactions.

Benefits: So what benefits does the Child-per-Entity pattern offer to developers?

1. 1:1 correlation between domain model and actor model, which simplifies many stateful programming scenarios;
2. Extremely reliable stateful routing in clustered and network environments - every domain entity can map to one and only one destination actor; and
3. Simplified and smaller code footprint. Rather than write code to manage N number of entities at once, write an actor who manages exactly one instance of the entity and deploy N of them.

Code Sample

Inside `AkkaChat` we use this pattern to maintain real-time authentication sessions for chat users, and the `UserSessionManager` is the actor ultimately responsible for powering the entire authentication system Nancy relies on in this sample.

```
/// <summary>
/// Responsible for managing all user sessions
/// </summary>
public class UserSessionManager : ReceiveActor
{
    private readonly IActorRef _identityStore;
    private readonly IActorRef _authChallenge;
    private Dictionary<string, IActorRef> _usersByName;

    public UserSessionManager(IActorRef identityStore, IActorRef authChallenge)
    {
        _identityStore = identityStore;
        _authChallenge = authChallenge;
        _usersByName = new Dictionary<string, IActorRef>();
        Ready();
    }

    private void Ready()
    {
        Receive<UserSessionActor.CheckSession>(session =>
        {
            var userSessionActor = LookupOrCreateUserSessionActor(session.CookieId);
            userSessionActor.Forward(session);
        });

        Receive<UserIdentity>(identity =>
        {
            _usersByName[identity.DisplayName] = LookupOrCreateUserSessionActor(identity.Coo
        });

        Receive<UserSessionActor.FetchIdentity>(identity =>
        {
            if (_usersByName.ContainsKey(identity.DisplayName))
            {
                _usersByName[identity.DisplayName].Forward(identity);
            }
            else
            {
                Sender.Tell(new MissingUserIdentity(identity.DisplayName));
            }
        });
    }

    private IActorRef LookupOrCreateUserSessionActor(Guid cookieId)
    {
       var child = Context.Child(cookieId.ToString());
```

```
        if (child.Equals(ActorRefs.Nobody)) //child doesn't exist
        {
            return Context.ActorOf(Props.Create(() => new UserSessionActor(cookieId, _identi
                cookieId.ToString());
        }
        return child;
    }

    protected override void PreRestart(Exception reason, object message)
    {
        //preserve all children in the event of a restart
    }
}
```

There are three things to take note of in the `UserSessionManager`.

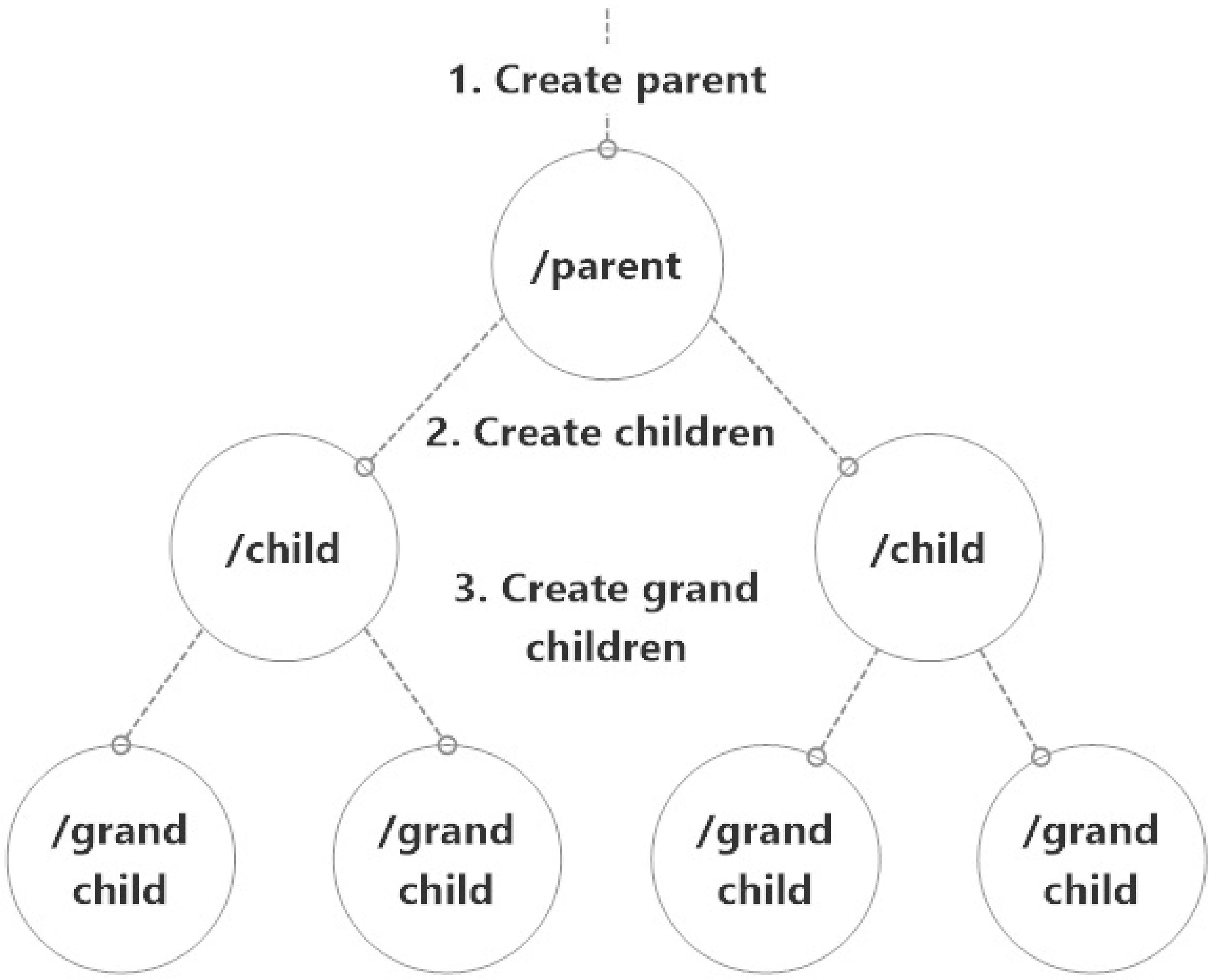
- The `LookupOrCreateUserSessionActor` method - it creates one actor per the the `Guid` used inside Nancy's authentication cookie. The `Guid` itself is the *name of the child* actor, so we're actually using the built-in `Context.Child` method to serve as our mapping.
- However, we also need to be able to look up our children by the `DisplayName` of the user - this comes up during SignalR authentication. So we created `private Dictionary<string, IActorRef> _usersByName;` to act as a secondary mapping.
- Notice how we overrode the `PreRestart` actor lifecycle method? This is because by default actors will dispose of their children during restarts. We want to preserve all of these entities, so we need to eliminate the call to `base.PreRestart` in order to ensure that this happens.

Fan-out Pattern

Imagine if you need to be able to deploy large hierarchies of actors onto remote machines, or if you need to have a large pool of actors carry out a long-running task?

This is exactly what the *Fan-out Pattern* can help us do!

Description: The Fan-out Pattern occurs when entire hierarchies of actors need to be able to be deployed in order to fulfill a task. Rather than have one actor responsible for creating a large number of children, have a single actor be deployed who is then responsible for creating a small number of children who create grandchildren and so on.



Use Cases:

- Parallel work loads that can be distributed across multiple machines and
- Refactoring complex operations into a series of small ones.

Benefits:

- Adds layers of local decision making and supervision away from the original parent, reducing the likelihood of a large rolling restart while also simplifying the deployment process;
- Makes it possible to deploy hierarchies of actors onto remote or clustered actor systems with a single command; and
- Simplifies the design of actors at every level - breaks up areas of responsibilities into small, discrete parts.

Code Sample

The code sample that best illustrates this concept actually comes from Petabridge's Cluster WebCrawl sample - it uses the fan-out pattern extensively to deploy hierarchies of workers who crawl web pages across multiple processes on the network.

Here's an example of an actor who fans out:

```
/// <summary>
/// Actor responsible for managing pool of <see cref="DownloadWorker"/> and <see cref="Parse
///
/// Can be remote-deployed to other systems.
///
/// Publishes statistics updates to its parent.
/// </summary>
public class DownloadCoordinator : ReceiveActor
{
    #region Constants

    public const string Downloader = "downloader";
    public const string Parser = "parser";

    #endregion

    #region Messages

    /// <summary>
    /// Used to signal that it's time to publish to the JobMaster
    /// </summary>
    public class PublishStatsTick
    {
        private PublishStatsTick() { }
        private static readonly PublishStatsTick _instance = new PublishStatsTick();

        public static PublishStatsTick Instance
        {
            get { return _instance; }
        }
    }

    #endregion

    protected readonly IActorRef DownloadsTracker;
    protected readonly IActorRef Commander;

    protected IActorRef DownloaderRouter;
    protected IActorRef ParserRouter;

    protected CrawlJob Job;
    protected CrawlJobStats Stats;

    protected readonly long MaxConcurrentDownloads;

    private ICancelable _publishStatsTask;
    private ILoggingAdapter _logger = Context.GetLogger();

    public DownloadCoordinator(CrawlJob job, IActorRef commander, IActorRef downloadsTracker
    {
        Job = job;
        DownloadsTracker = downloadsTracker;
        MaxConcurrentDownloads = maxConcurrentDownloads;
        Commander = commander;
        Stats = new CrawlJobStats(Job);
        Receiving();
    }

    protected override void PreStart()
    {

        // Create our downloader pool
        if (Context.Child(Downloader).Equals(ActorRefs.Nobody))
        {
            DownloaderRouter = Context.ActorOf(
                Props.Create(() => new DownloadWorker(HttpClientFactory.GetClient, Self, (in
                .WithRouter(new RoundRobinPool(10)), Downloader);
        }

        // Create our parser pool
        if (Context.Child(Parser).Equals(ActorRefs.Nobody))
        {
            ParserRouter = Context.ActorOf(
                Props.Create(() => new ParseWorker(Job, Self)).WithRouter(new RoundRobinPool
                Parser);
        }

        // Schedule regular stats updates
        _publishStatsTask = new Cancelable(Context.System.Scheduler);
       Context.System.Scheduler.ScheduleTellRepeatedly(TimeSpan.FromMilliseconds(250),
        TimeSpan.FromMilliseconds(250), Self, PublishStatsTick.Instance, Self, _publishStats
    }
}
```

The `DownloadCoordinator` gets deployed by a clustered router onto new machines who join the network, and it in turn deploys two routers who deploy between 10-20 routees each! That's a heirarchy of 40+ actors that unpacked themselves automatically from a single deployment!

Parent Proxy Pattern

In production applications, often times you can have complex dependencies between lots of components in a system. Actors are no different! There are lots of cases where one actor will take a reference to another as a dependency in its constructor (because it's an important testability pattern!)

What if you want to just defer this entire process and give any actors who take a dependency on `FooActor` a reference to an `IActorRef` that already exists, and will faithfully forward all of their messages onto `FooActor` as soon as it's ready?

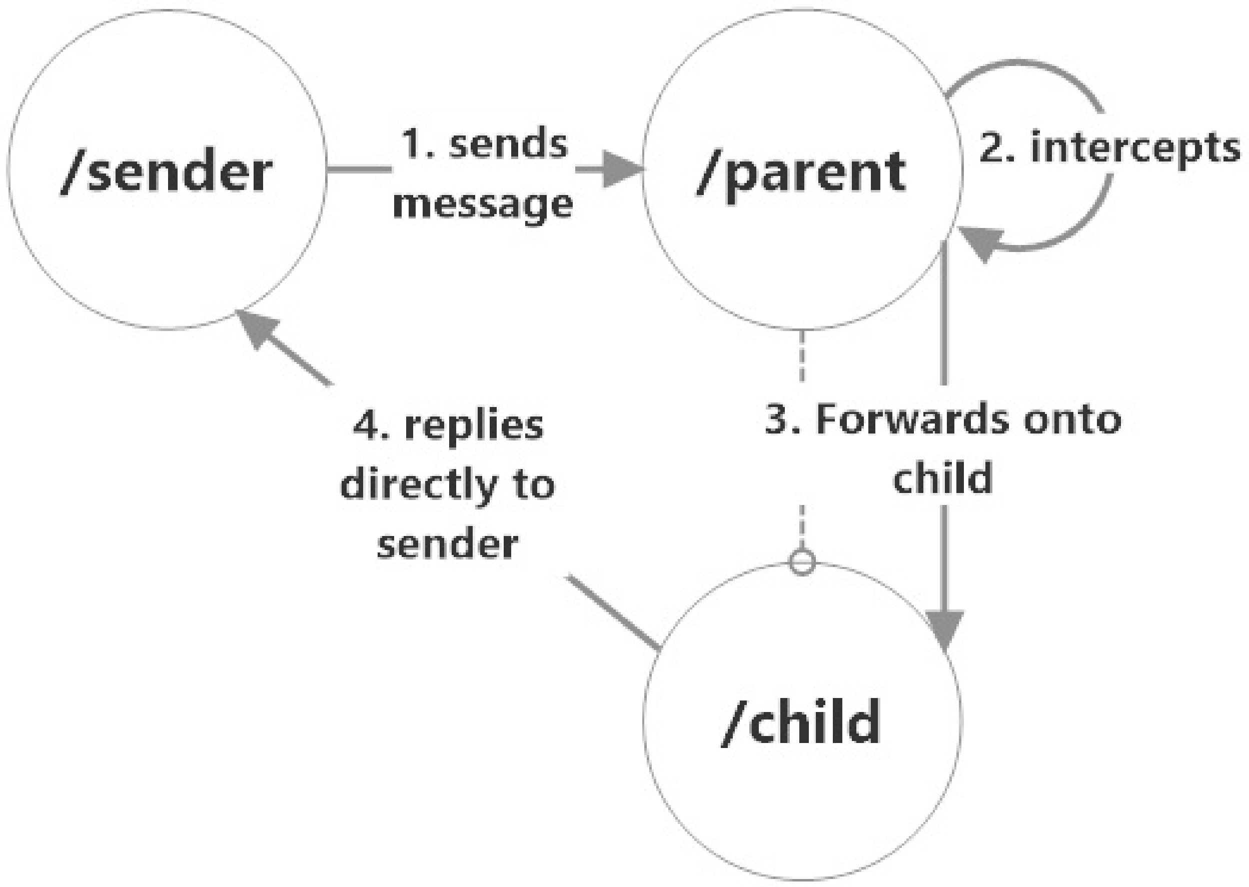
Or what if you want to be able to intercept messages before they reach a particular actor and perform some additional work or collaboration beforehand?

Enter the *Parent Proxy Pattern*!

Description: The Parent Proxy pattern occurs when an external actor ("the depender") needs to take a dependency on a particular actor, call it "child" in this case, but "child" can't be created right now or won't be available for work until after a period of time elapses. Instead of passing an actor reference of child to the depender right away, we'll instead create a new actor called "parent" who's responsible for simultaneously instantiating child and acting as a proxy for the depender. This allows child do complete its work without delaying the initialization of the depender.

Parent can also perform some additional interception, routing, or collaboration with other actors before finally forwarding messages to child.

> Child is always totally unaware of parent's involvement in messages; to child all messages appear as though they came directly from the depender.



Use Cases:

- Dynamic routing;
- Deferred creation of actors, usually due to sequencing or resource constraints;
- Message inspection, logging, or monitoring; and
- Distribution of messages and coordination to other actors.

Benefits:

- Fault tolerance and error isolation - insulates dependent actors from potential sources of failure and adds an injection point for restarting critical actors;
- Abstraction and loose coupling - offers injection points for being able to modify and extend the message flow between sender and child without either of those two actors knowing about it; and
- Routing - all `Pool` routers in Akka.NET work this way already; this pattern is very effective for introducing intelligent message routing and distribution.

Code Sample

All built-in `Pool` routers in Akka.NET are effectively parent proxies! But let's take a look at a specific `AkkaChat` example - the `IdentityManager`.

```
/// <summary>
/// PARENT PROXY PATTERN.
///
/// Creates <see cref="IdentityWriterActor"/> instances upon request but doesn't
/// really engage with them. Really serves as a broker between actors who need a user
/// created and the <see cref="IdentityWriterActor"/>.
/// </summary>
public class IdentityManager : ReceiveActor
{
    #region Message types

    /// <summary>
    /// Request an actor reference to the identity store
    /// </summary>
    public class GetIdentityStore { }

    #endregion

    private IActorRef _identityStore;

    public IdentityManager()
    {
        Receive<IdentityWriterActor.CreateUser>(create =>
        {
            var writer = Context.DI().ActorOf<IdentityWriterActor>();
            writer.Forward(create); //forward the current sender to the writer
        });

        //do nothing
        Receive<Terminated>(terminated => { });

        //pass the identity of the store directly to the caller
        Receive<GetIdentityStore>(store => Sender.Tell(_identityStore));

        //forward all other types of identity-related messages to the IdentityStoreActor
        ReceiveAny(m => _identityStore.Forward(m));
    }

    protected override void PreStart()
    {
        _identityStore = Context.DI().ActorOf<IdentityStoreActor>("identity");
    }

    protected override void PreRestart(Exception reason, object message)
    {
        //don't kill off our children
    }
}
```

Take a look at the `ReceiveAny` block - other than a set of specific messages the

`IdentityManager` reserves for creating users, everything else gets transparently forwarded to one of its children - the `IdentityStoreActor`.

The reason we did this is that a large number of other `AkkaChat` actors take a dependency on the `IdentityStoreActor`, so we introduced a proxy in order to help isolate those other actors from potential failures in the event that the `IdentityStoreActor` wasn't available right away or was recovering from a crash.

Next Step

We can now start to see some areas where actor composition patterns can help us made good choices with respect to our models for AkkaChat. Now lets explore some specific **Messaging Patterns.** to better understand how to create effective communication patterns between actors.

Messaging Patterns

One of the most crucial concepts to get right early is thinking through how actors will communicate with each other.

Here's the messaging patterns we'll cover in this section:

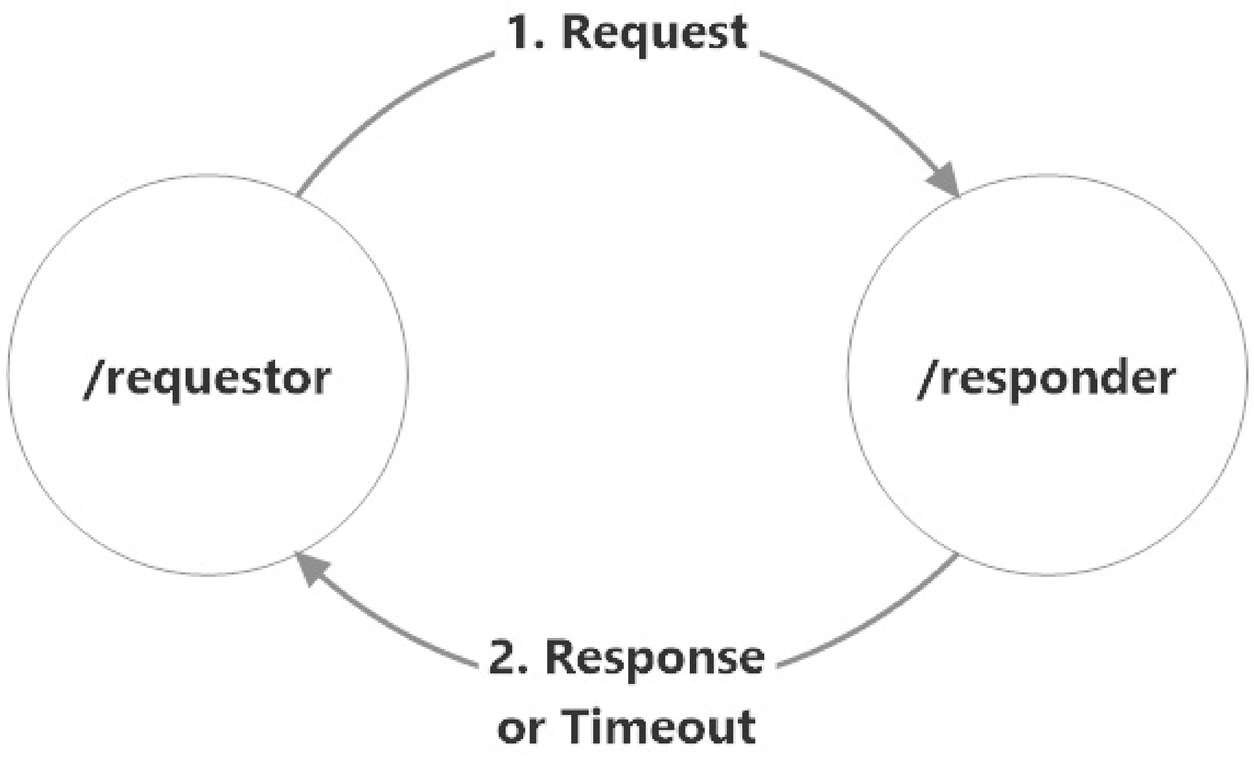
- Request-Response
- Push and Pull
- Publish-Subscribe (pub-sub)

There's only a small number of messaging patterns, but identifying the right one for each particular instance makes the difference between predictable, clear behavior and out of control nonsense.

Request-Response

Request / response is the most intuitive of all of the messaging patterns, because it's one that we're innately familiar with as web programmers. Request-response is exactly how HTTP works!

Description: For each request made to a given actor the sender expects exactly one message in response. In the event that no response arrives, the sender assumes that the operation fails and treats the timeout like an error.



Use Cases: Any scenario in which the `/requestor` cannot proceed without acknowledgment or fulfillment of its request is one in which request-response should be used.

Here's some examples:

- Authentication challenges;
- Determining if a particular resource is available;
- Guaranteed delivery of messages; and
- Transactions.

> Request-response is practically ubiquitous, given that everything we do on top of HTTP already uses it. However, this isn't the default way actors communicate with each other - in fact there *is no default*, so whenever we need request-response it has to be implemented ourselves.

Benefits: The benefit of request-response is that it is a high consistency messaging pattern. When done correctly it leaves very little to chance - you will know, with a high degree of confidence, what the state of your interaction is after receiving a response back from the `/responder`.

That's why request-response gets used in mission critical areas like authentication and transactions - it's makes the state of those operations explicit and well-known the majority of the time.

Code Sample

The `AuthenticationChallengeActor` in `AkkaChat` is a great example of an actor with a request-response message flow. In fact, it has this message flow in two directions - between itself and the `IdentityStoreActor` and its requestor.

```
/* Receive method inside the AuthenticationChallengeActor */
Receive<AuthenticationInformation>(info =>
{
    /* FOUND matching authentication information */
    var auth =
       _pendingAuthentications.FirstOrDefault(
           x => x.Challenge.DisplayName.Equals(info.DisplayName));
    if (auth == null)
    {
        _log.Warning(
            "Found authentication information for user {0}, but have no record of them in pe
            info.DisplayName);
    }
    else
    {
        var success = _crypto.CheckPassword(auth.Challenge.AttemptedPassword, info.HashedPas
        if (success) /* AUTHENTICATION SUCCESS */
        {
            _userIdentityStore.Ask<UserIdentity>(new IdentityStoreActor.FetchUserByName(info
                .ContinueWith<object>(tr =>
                {
                    if(tr.IsFaulted || tr.Result is MissingUserIdentity)
                        return new AuthenticationFailure(info.DisplayName,
                    "Authentication attempt was successful,
                    but connection to database failed during final handshake. Please retry!"
                    return new AuthenticationSuccess(tr.Result, tr.Result.CookieId); //creat
                }).PipeTo(auth.Requestor);
        }
        else //AUTHENTICATION FAILURE - PASSWORD MISMATCH
        {
            auth.Requestor.Tell(new AuthenticationFailure(info.DisplayName, "Login failed. W
        }

        _pendingAuthentications.Remove(auth);
        auth.Timeout.Cancel();
    }
});
```

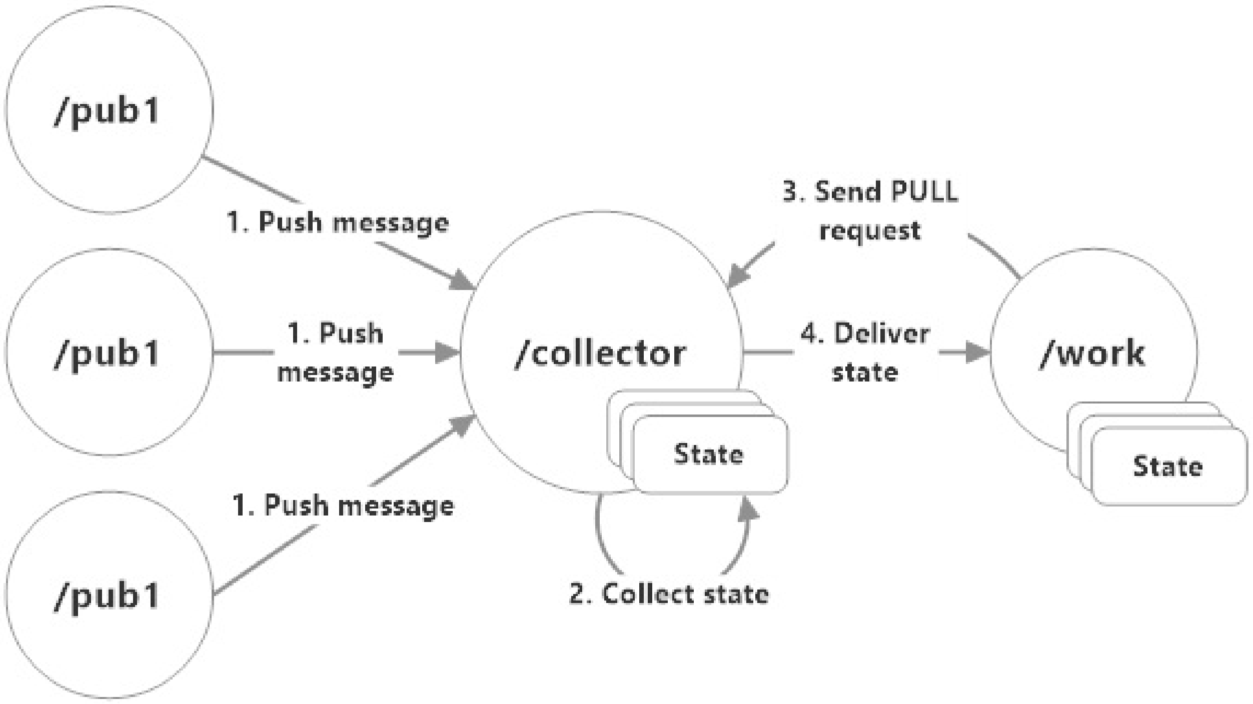
Notice the `AuthenticationSuccess` and `AuthenticationFailure` response messages? Those are what it guarantees to deliver back to its caller, even in the event that this `Task` fails (there's other timeout code not shown here that guarantees that.)

The `AuthenticationChallengeActor` is also guaranteed to receive either an `ExistingUserIdentity` message or a `MissingUserIdentity` response back from the `IdentityStoreActor` when it requests password information that it can use to perform its password challenge, so all around we have a strongly consistent messaging model built around this actor.

Push and Pull

Push and pull is an aggregation and accumulation messaging pattern that you typically find in stateful systems, and we use it inside `AkkaChat` for the message history of chat rooms.

Description: Multiple messages are pushed over time into a single accumulator inside an application, and periodically other subscribers will pull the accumulated messages or derived state from the accumulator in order to perform some additional work.



Use Cases: Push and pull is predominately used in stateful systems that accumulate or aggregate state over periods of time. The `AkkaChat` message history for each chat room is a good example!

In particular, it's important when you need to have a single source of truth on the state of some part of the system at any given point in time. The `/collector` can reset its state to empty or retain its current state until it's explicitly terminated, but during the windows between those types of events it will remain the single source of truth for a significant portion of the system.

Benefits: The benefits of push and pull are enormous, even though this pattern isn't as common as Request-response or publish-subscribe.

- Simplifies state management and provides clear indications as to where message persistence needs to be added;
- Makes no decisions about how its state gets used or read, which makes accumulators have a relatively small code base; and
- Works well across network calls - it's common to have `/collector` type actors sit in their own service inside an Akka.Cluster environment and simply observe and mutate state, only sharing it with other services upon request.

Code Sample

As mentioned earlier, the `RoomMessageHistoryActor` is a good example of a Push and Pull message flow, and it does this marvelously in combination with Akka.Persistence.

```
public class RoomMessageHistoryActor : ReceivePersistentActor
{
    private readonly ExistingRoom _room;
    private readonly IActorRef _signalRWriter;
    private MessageHistory _history;

    private string _cachedPersistenceId;
    public override string PersistenceId
    {
        get
        {
            if (string.IsNullOrEmpty(_cachedPersistenceId))
            {
                _cachedPersistenceId = string.Format("room-{0}-messages", _room.Id);
            }
            return _cachedPersistenceId;
        }
    }

    public RoomMessageHistoryActor(ExistingRoom room, IActorRef signalRWriter)
    {
        _signalRWriter = signalRWriter;
        _room = room;
```

```
            _history = new MessageHistory();
            ReadyCommands();
            ReadyRecovers();
        }

        private void ReadyCommands()
        {
            //SystemMessages don't get added to the message history
            Command<SystemMessage>(message =>
            {
                _signalRWriter.Forward(message);
            });

            //UserMessages do get persisted
            Command<UserMessage>(message => Persist(message, UpdateHistory));

            //Need to catch specific SignalR user up on the messages
            Command<FetchPreviousMessages>(previous =>
            {
                var queryTime = previous.Since ?? DateTime.MaxValue;
                var messages = _history.Messages.Where(x => x.When <= queryTime).OrderBy(x => x.
                _signalRWriter.Tell(new HistoricalMessageBatch(messages, previous.RoomName, prev
            });
        }

        private void UpdateHistory(ChatMessage message)
        {
            if (!_history.WillUpdate(message)) return;
            _signalRWriter.Forward(message);
            _history = _history.Update(message);
        }

        private void ReadyRecovers()
        {
            Recover<UserMessage>(message => RecoverHistory(message));
        }

        private void RecoverHistory(ChatMessage message)
        {
            _history = _history.Update(message);
        }
    }
```

The `MessageHistory` container effectively acts as a state bag for all of the `ChatMessage`s pushed into the `RoomMessageHistoryActor` by SignalR.

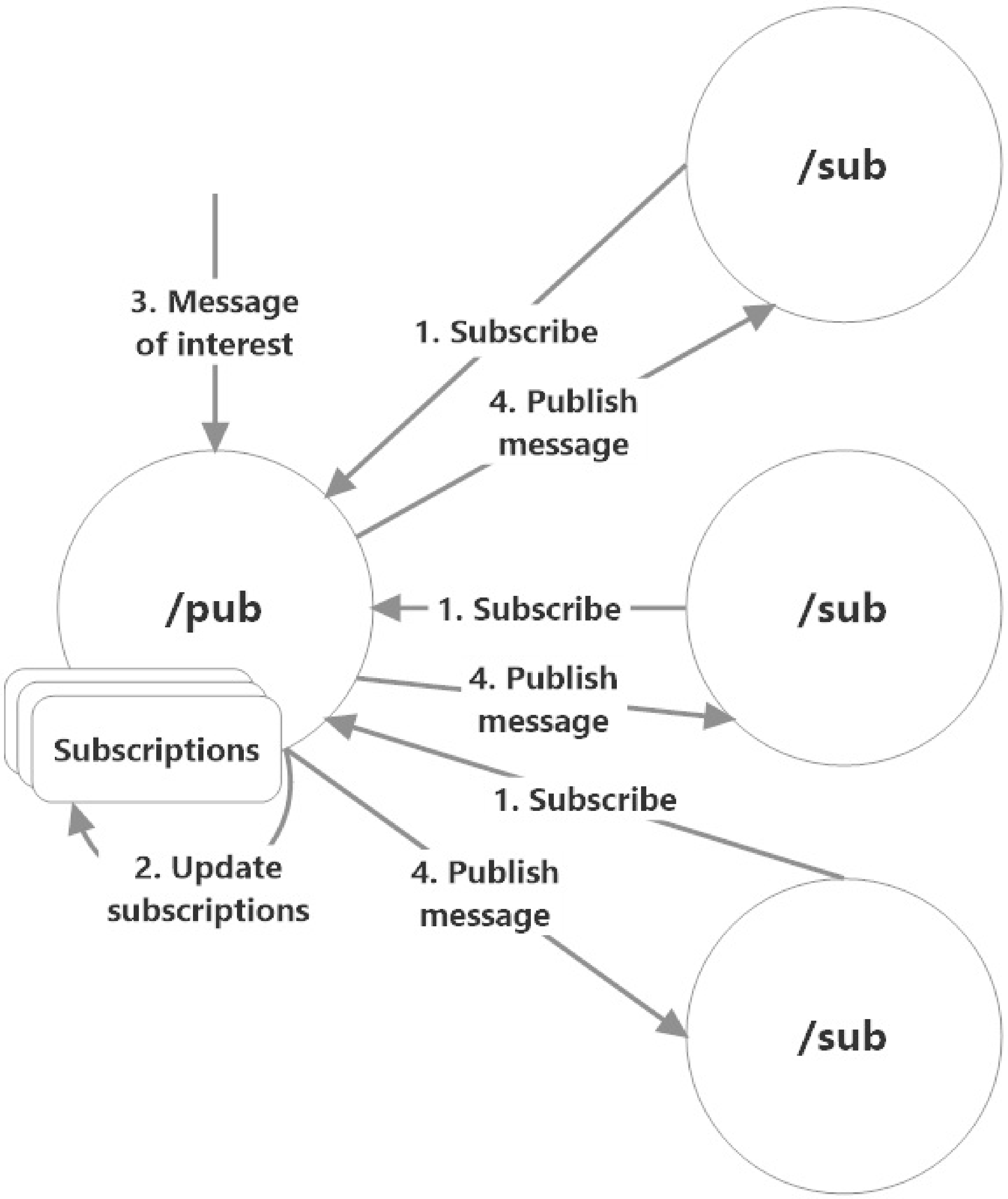
Whenever a new participant enters the chatroom this actor will receive a `FetchPreviousMessages`. some portion of that accumulated state will be pulled out of the `RoomMessageHistoryActor` and into the new SignalR user's session.

Publish-Subscribe (Pub-Sub)

Publish-Subscribe (pub-sub) is delightfully simple with Akka.NET and it's a natural extension of the reactive programming style inherent to actors.

Definition: Pub-sub occurs when a subscriber actor notifies a publisher that it would like to periodically receive updates on specific topics, types of messages, when the publisher becomes aware of them. The publisher keeps track of its subscribers and follows through on its promise to push messages to them when it has updates on topics of interest to subscribers.

Subscribers also have the option of unsubscribing from publishers at any time.



Use Cases: Pub-sub is used in stateful systems that deal in high frequencies and variety of messages, especially event-driven systems. It's a staple of any distributed message-based system also.

Benefits: The biggest benefit of pub-sub is that is consolidates all of the logic in topic subscription and publication into one place, which puts all of the actors who need to do actual work into a position where they can just sit back and wait for messages to arrive. In a pub-sub environment no one really needs to "seek out" work - work finds you instead!

Code Sample

The code sample that best illustrates this concept actually comes from Petabridge's Cluster WebCrawl sample - it's uses pub-sub to ensure that clients who start long-running crawl jobs automatically receive updates from all of the nodes in the cluster participating in the job.

```
/// <summary>
/// Actor responsible for individual <see cref="CrawlJob"/>
/// </summary>
public class CrawlMaster : ReceiveActor, IWithUnboundedStash
{
    public const string CoordinatorRouterName = "coordinators";
    protected readonly CrawlJob Job;

    /// <summary>
    /// All of the actors subscribed to updates for <see cref="Job"/>
    /// </summary>
    protected HashSet<IActorRef> Subscribers = new HashSet<IActorRef>();

    protected JobStatusUpdate RunningStatus;

    protected CrawlJobStats TotalStats
    {
        get { return RunningStatus.Stats; }
        set { RunningStatus.Stats = value; }
    }

    protected IActorRef CoordinatorRouter;
    protected IActorRef DownloadTracker;

    public IStash Stash { get; set; }

    public CrawlMaster(CrawlJob job)
    {
        Job = job;
        RunningStatus = new JobStatusUpdate(Job);
        TotalStats = new CrawlJobStats(Job);
        Context.SetReceiveTimeout(TimeSpan.FromSeconds(5));
           Ready();
    }

    protected override void PreStart()
    {
        /* Request a download tracker instance from the downloads master */
        Context.ActorSelection("/user/downloads").Tell(new DownloadsMaster.RequestDownloadTr
    }

    private void Ready()
```

```
        {
            // kick off the job
            Receive<StartJob>(start =>
            {
                Subscribers.Add(start.Requestor);
                var downloadRootDocument = new DownloadWorker.DownloadHtmlDocument(new CrawlDocu

                //should kick off the initial downloads and parsing
                CoordinatorRouter.Tell(downloadRootDocument);

                Become(Started);
                Stash.UnstashAll();
            });

            ReceiveAny(o => Stash.Stash());

        }

        private void Started()
        {
            Receive<StartJob>(start =>
            {
                //treat the additional StartJob like a subscription
                if (start.Job.Equals(Job))
                    Subscribers.Add(start.Requestor);
            });

            Receive<SubscribeToJob>(subscribe =>
            {
                if (subscribe.Job.Equals(Job))
                    Subscribers.Add(subscribe.Subscriber);
            });

            Receive<UnsubscribeFromJob>(unsubscribe =>
            {
                if (unsubscribe.Job.Equals(Job))
                    Subscribers.Remove(unsubscribe.Subscriber);
            });

            Receive<CrawlJobStats>(stats =>
            {
                TotalStats = TotalStats.Merge(stats);
                PublishJobStatus();
            });

            Receive<StopJob>(stop => EndJob(JobStatus.Stopped));

            //job is finished
            Receive<ReceiveTimeout>(timeout => EndJob(JobStatus.Finished));
        }

        private void EndJob(JobStatus finalStatus)
        {
            RunningStatus.Status = finalStatus;
            RunningStatus.EndTime = DateTime.UtcNow;
            PublishJobStatus();
            Self.Tell(PoisonPill.Instance);
        }


        private void PublishJobStatus()
        {
            foreach (var sub in Subscribers)
                sub.Tell(RunningStatus);
        }
    }
```

These jobs can run for hours, so the `CrawlMaster` receives tens of thousands of updates which it will push to anyone who's interested in listening to the results of the crawl operation.

Next Step

We have a good feel for some of the common messaging and interaction patterns between actors now. So what can we do to make sure that our actors can perform their functions well?

Let's apply some **Reliability Patterns.** and find out!

Reliability Patterns

One of the fundamental benefits of using the actor model is its ability to form self-healing networked applications - and the way this is accomplished is through taking advantage of its supervision and restart capabilities using reliability patterns!

Here are the patterns we'll be covering in this section:

- Character Actor Pattern (Error Kernel Pattern)
- Supervisor Strategy Pattern
- Consensus Pattern

Character Actor Pattern (Error Kernel Pattern)

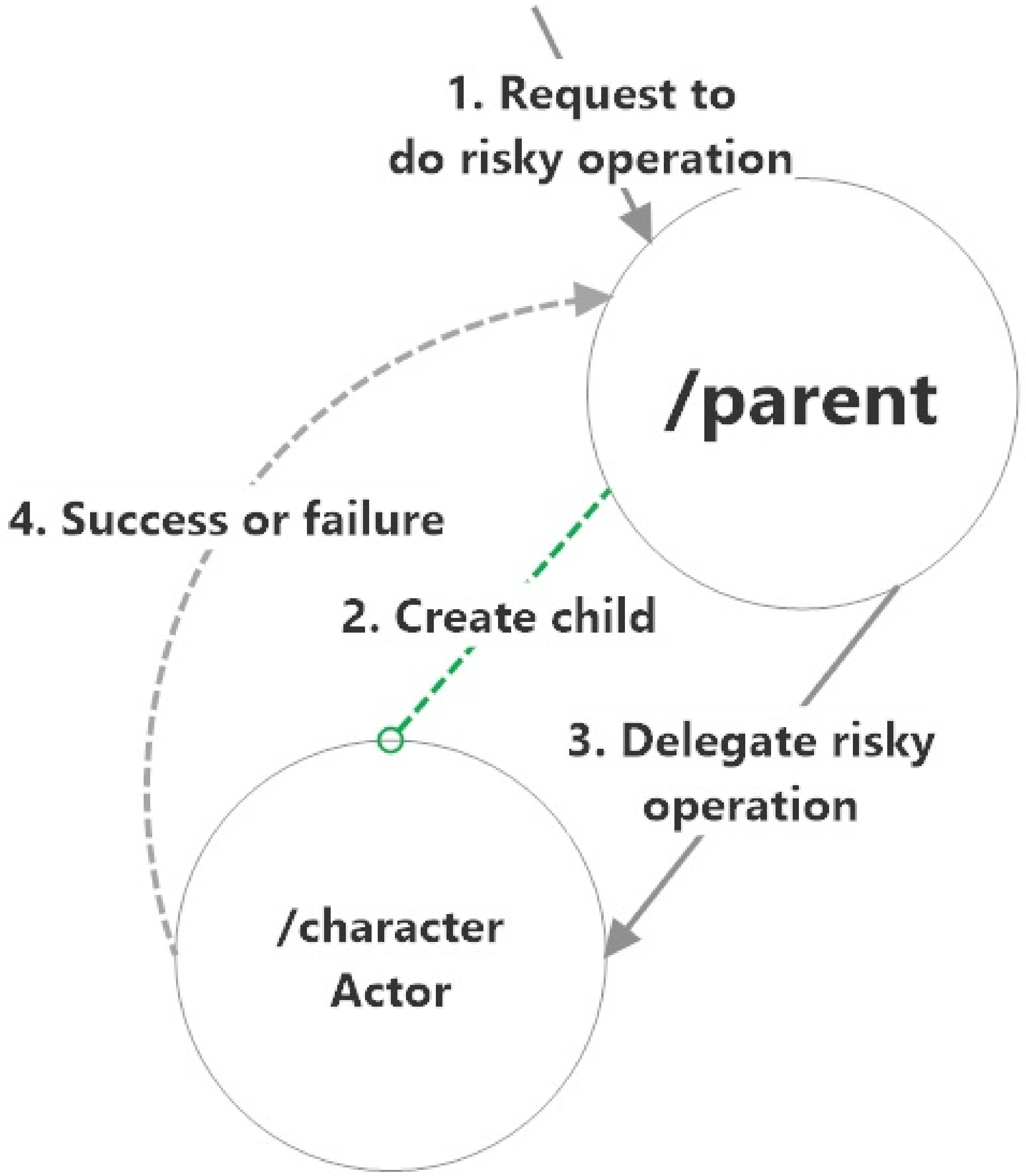
The name *Character Actor Pattern* was coined by Scott Hanselman during an interview with Aaron Stannard about Akka.NET, and it refers to a specific actor composition pattern that's used to ensure reliability through the use of disposable children.

Definition: The Character Actor Pattern is an instance where an application has some risky, but critical operation that needs to be safely executed without any negative side effects. In many circumstances it's often cheaper, faster, and more reliable to simply delegate these risky operations to a purpose-built, but trivially disposable actor whose sole responsibility is to carry out the operation successfully or die trying.

These brave, disposable actors are *Character actors*.

Character actors can have be temporary or long-running actors, but typically they're designed to carry out only one specific type of risky operation. Often times character actors can be re-used throughout an application, belonging to many different types of parents.

> This pattern has a less fun name: the **Error kernel pattern**, which is the name given to it by Erlang.



Use Cases: The Character Actor pattern is broadly applicable - any time you need to do something risky such as network calls, file I/O, parsing malformed content, and so on makes for a good candidate for a character actor. However, character actors are most effective when used to provide protection and fault isolation to some other important type of actor, typically one containing some important state.

Benefits:

- Insulates stateful and critical actors from failures and risky operations;
- Makes it easy to cleanly introduce retry / backoff / undo semantics specific to each type of risky operation; and
- Reduces code - let's the `SupervisionStrategy` and actor lifecycle do most of the heavy lifting.

Code Sample

The `IdentityWriterActor` inside `AkkaChat` is a *very* short-lived character actor designed to carry out exactly one task: register new users to SQL Server.

We don't want to disrupt the operations of the `IdentityStoreActor`, who's responsible for the read-side of everything `User`-related, so we encapsulate each individual write operation inside its own `IdentityStoreActor` due to how infrequently they occur.

Take a look at the code inside the `IdenityManager`, the actor who's ultimately the parent of both the `IdentityStoreActor` and all of the `IdentityWriterActor` instances. Here's what the character actor pattern actually looks like in practice:

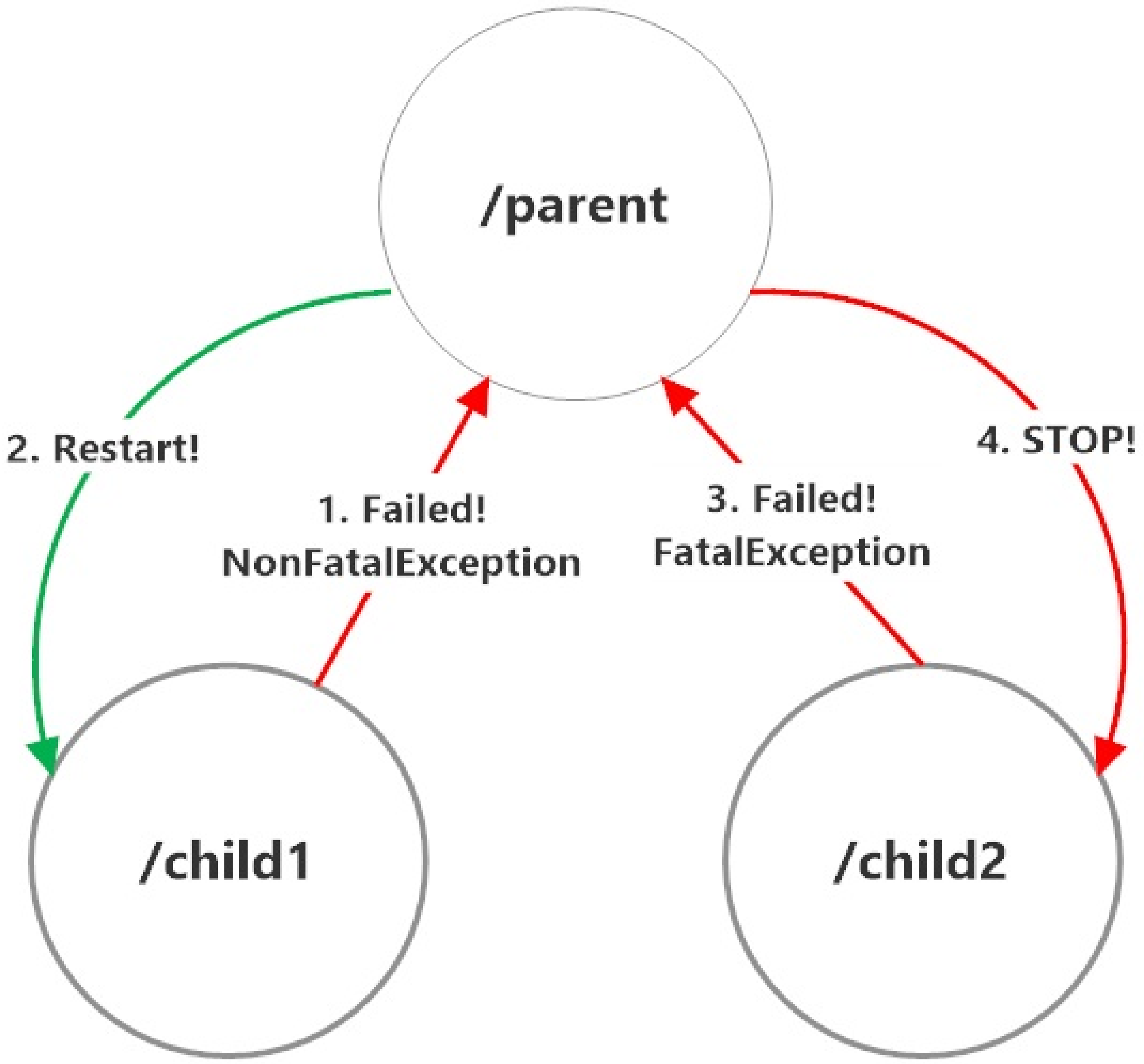
```
/// <summary>
/// PARENT PROXY PATTERN.
///
/// Creates <see cref="IdentityWriterActor"/> instances upon request but doesn't
/// really engage with them. Really serves as a broker between actors who need a user
/// created and the <see cref="IdentityWriterActor"/>.
/// </summary>
public class IdentityManager : ReceiveActor
{
    private IActorRef _identityStore;

    public IdentityManager()
    {
        Receive<IdentityWriterActor.CreateUser>(create =>
        {
            var writer = Context.DI().ActorOf<IdentityWriterActor>();
            writer.Forward(create); //forward the current sender to the writer
        });
    }
}
```

For each `CreateUser` request, we create an `IdentityWriterActor` who will carry out this operation **and** shut itself down afterwards so we don't waste resources.

Supervisor Strategy Pattern

If you've gone through the Akka.NET Bootcamp, you'll have at least some understanding of how each actor's built-in `SupervisorStrategy` works. In this case we're going to take a look at how you can write your own `SupervisorStrategy` and custom `IDecider` to make decisions to enhance the reliability of your applications.

Description: the supervisor strategy pattern occurs when a parent actor discriminates against the types of exceptions its children can throw and takes different actions according to the severity of the exception.



Use Cases: the supervisor strategy pattern is typically used when the parent has some knowledge about the work of its children and the type of failures they can occur, and specific types of failures provide a stronger indication as to the recoverability of an actor and its operations. In these cases the parent can make more specific decisions in order to provide greater reliability guarantees.

This also usually comes up when the parent needs to protect or minimize access to some resource that's shared by the child actors, such as database connections. If a parent encounters a child actor that holds onto a database connection and keeps experiencing fatal errors, it's possible that the child's database connection itself is the problem and a restart won't cut it - escalation of the error might be required.

Benefits: The biggest benefit is that local supervision decisions decrease the amount of decisions that need to be made in top-level actors - they effectively prevent small errors from rolling uphill to big actors with large hierarchies, where the costs of restart can be tremendous.

Code Sample

Inside `AkkaChat` we use a handful of custom exceptions inside our `UserSessionActor` to provide signals as to whether or not a particular session is in a verifiable state. If we can't verify a logged-in users session, then the `UserSessionActor` is in a state where it cannot recover.

So rather than silently stop, we can send a detailed exception back to the `UserSessionManager` to let it know that this particular session is borked.

```
/// <summary>
/// Actor session that corresponds 1:1 with an active user
/// </summary>
public class UserSessionActor : ReceiveActor, IWithUnboundedStash
{
    private readonly Guid _cookieId;
    private UserIdentity _userIdentity;

    protected UserIdentity Identity
    {
        get { return _userIdentity; }
        set
        {
            _userIdentity = value;
            _akkaChatIdentity = new AkkaChatIdentity(_userIdentity);
            _sessionAck = new ValidSession(_akkaChatIdentity);
        }
    }

    private ValidSession _sessionAck;
```

```
        private AkkaChatIdentity _akkaChatIdentity;
        private readonly ILoggingAdapter _log = Context.GetLogger();
        private readonly IActorRef _identityActor;

        public IStash Stash { get; set; }

        public UserSessionActor(Guid cookieId, IActorRef identityActor)
        {
            _cookieId = cookieId;
            _identityActor = identityActor;
            WaitingForIdentity();
        }

        private void WaitingForIdentity()
        {
            Receive<MissingUserIdentity>(missing => PublishAndThrow(_cookieId));

            Receive<UserIdentity>(identity =>
            {
                Identity = identity;
                Context.Parent.Tell(identity);
                BecomeReady();
            });

            //Timed out - didn't hear back from identity actor
            Receive<ReceiveTimeout>(timeout => PublishAndThrow(_cookieId));

            //stash any other messages
            ReceiveAny(o => Stash.Stash());
        }

         private void PublishAndThrow(Guid cookieId)
            {
                _log.Error("Not able to locate user for cookie ID {0}", cookieId);
                throw new UnidentifiedAkkaSessionException(string.Format("Could not locate user 
                cookieId), cookieId);
            }
    }
```

Take note of the `PublishAndThrow` method and the `UnidentifiedAkkaSessionException` - notice how the `cookieId` that could not be verified is incorporated into the exception? The `UserSessionManager` can use this information inside its `SupervisorStrategy`!

```
    public class UserSessionManager : ReceiveActor
    {
        #region Message classes
        #endregion

        #region SupervisorStrategy
        /// <summary>
        /// STOP any children who throw an <see cref="UnidentifiedAkkaSessionException"/> messag
        /// </summary>
        protected override SupervisorStrategy SupervisorStrategy()
        {
            return new OneForOneStrategy(Decider.From(Directive.Restart,
                new KeyValuePair<Type, Directive>(typeof(UnidentifiedAkkaSessionException), Dire
        }

        #endregion
    }
```

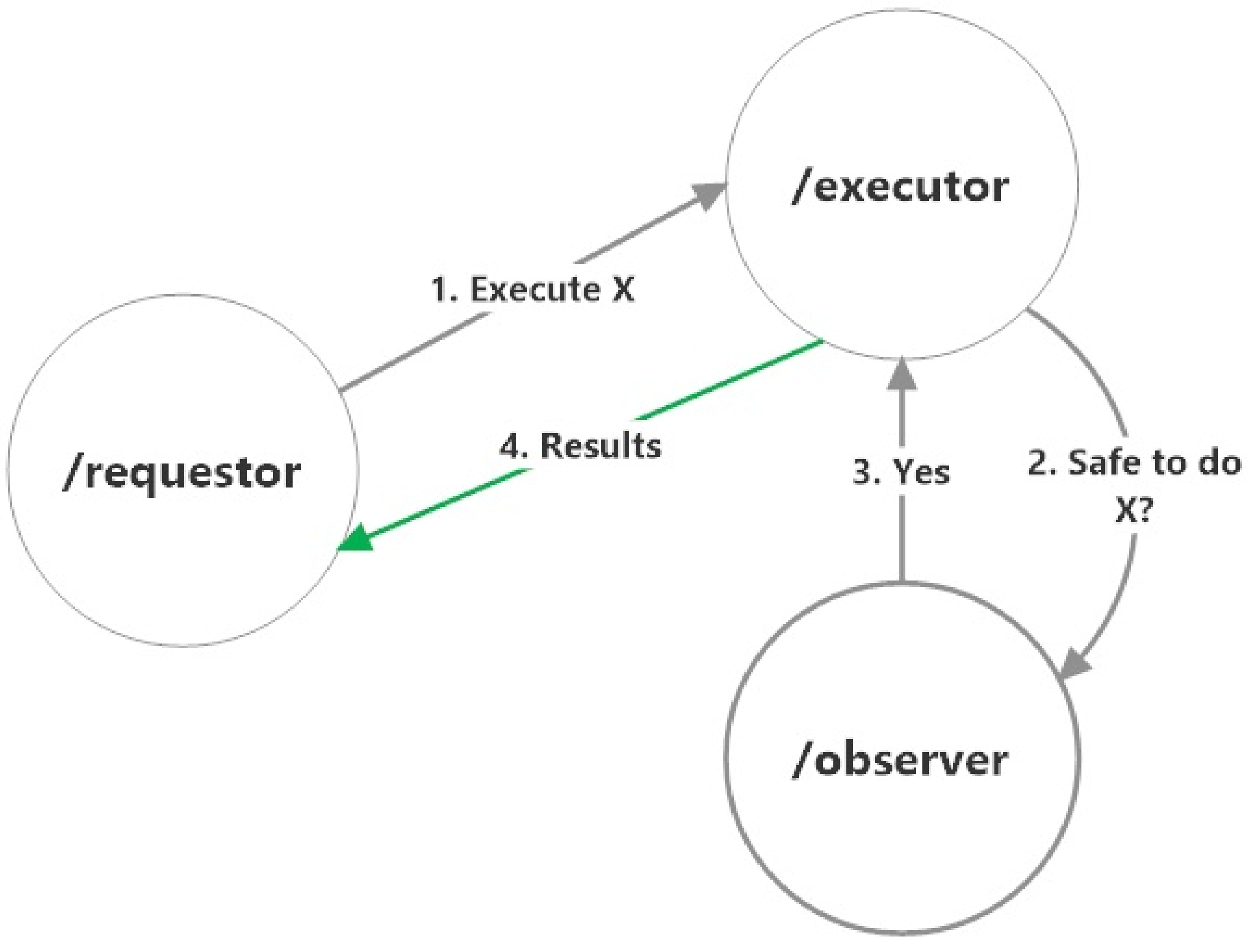
The `UserSessionManager` issues a default directive of `Restart` to any actors who fail, but we know that this particular exception is irrecoverable - so we explicitly send a `Stop` directive to any actor who throws this error. This logs the user out! Pretty cool!

Consensus Pattern

The consensus pattern is really a conversation pattern between actors to determine whether or not the system is in a state where a particular operation can be executed.

Description: When two or more actors have to establish agreement about the state of the system as a set of explicitly stated requirements before executing the operation. Each operation is identified to all involve parties and actors will explicitly send messages to each other referring to it before carrying on.

If consensus is reached, the operation shall be executed. If the consensus operation times out or achieves a negative result, the operation will not be carried out.



Use Cases: Used in operations that require higher consistency or in scenarios where access to a shared resource has to be monitored by some actor before other actors can access it.

Benefits:

- Increased consistency and data integrity;
- Fewer attempted operations in conditions that will produce failure; and
- Improved coordination between actors.

Code Sample

In `AkkaChat` we use a special message class called `CanProceed<T>` to wrap all operations that need the consensus pattern in a contract.

```
/// <summary>
/// Implements part of the CONSENSUS pattern.
///
/// Used as part of flow-control where consensus or permission is
/// needed to proceed with some operation.
```

```
///
/// <remarks>
/// This illustrates a naive implementation.
///
/// Full-blown distributed consensus is often much more complicated
/// and is an area of intense academic research.
/// </remarks>
/// </summary>
/// <typeparam name="T"></typeparam>
public class CanProceed<T>
{
    public CanProceed(T originalMessage, IActorRef intendedDestination, bool canExecute)
    {
        CanExecute = canExecute;
        IntendedDestination = intendedDestination;
        OriginalMessage = originalMessage;
    }

    public T OriginalMessage { get; private set; }

    public IActorRef IntendedDestination { get; private set; }

    public bool CanExecute { get; private set; }
}
```

Where it gets used is inside the `IdentityWriterActor`, who uses it to determine if a user's name is unique before we attempt to create the record in SQL Server.

```
/// <summary>
/// Actor responsible for creating new users
/// </summary>
public class IdentityWriterActor : ReceiveActor
{
    private IUserRepository _repository;

    public IdentityWriterActor(IUserRepository repository)
    {
        _repository = repository;

        Receive<CreateUser>(user =>
        {
            var sender = Sender;
            repository.UserNameIsAvailable(user.DisplayName)
                .ContinueWith(tr =>
                {
                    return new CanProceed<CreateUser>(user, sender, tr.Result.Payload);
                })
                .PipeTo(Self);

            // time this operation out if we don't hear back within three seconds
            SetReceiveTimeout(TimeSpan.FromSeconds(3));
        });
```

```
            //shut down in the event of a timeout
            Receive<ReceiveTimeout>(timeout => Context.Stop(Self));

            Receive<CanProceed<CreateUser>>(proceed =>
            {
                SetReceiveTimeout(null); //disable ReceiveTimeout
                if (proceed.CanExecute)
                {
                    var createUser = proceed.OriginalMessage;

                    // create the user
                    // TODO: could use AutoMapper here instead of manually doing the mapping.
                    var result = repository.CreateUser(new User()
                    {
                        DateJoined = createUser.DateJoined,
                        DisplayName = createUser.DisplayName,
                        Email = createUser.Email,
                        PasswordHash = createUser.PasswordHash,
                        PasswordSalt = createUser.PasswordSalt,
                        CookieId = createUser.CookeId
                    });

                    if (result.Status == ResultCode.Success)
                    {
                        //notify the original sender that we successfully created the user
                        proceed.IntendedDestination.Tell(new UserCreateResult(createUser.Display
                        Context.Stop(Self); //shut ourselves down
                        return; //done working
                    }
                }

                //let the caller know that we failed
                proceed.IntendedDestination.Tell(new UserCreateResult(proceed.OriginalMessage.Di
            });
        }
    }
```

This wrapper would allow us to maintain a level of consistency over our writes even without SQL Server's constraints - but as it stands right now we're using the `CanProceed` pattern in a very weak manner.

Exercise: Consensus Pattern

Why did we describe the current way the `IdentityWriterActor` uses the consensus pattern as "weak"? What could be a stronger way of doing this?

Think about your answer and then discuss with the instructor.

Next Steps

Now that we've covered reliability patterns, we can take our practice of making our actor-based applications reliable and resilient in production one step further by making them *testable*. Onto the next section: **Testability Patterns.**.

Testability Patterns

One of the keys to building successful large-scale applications on top of Akka.NET is learning how to leverage the Akka.TestKit and designs actors who are capable of using it.

In this section we'll learn some testability patterns that we can use to design individual actor classes that can be tested easily using Akka.TestKit.

Here are the patterns we're going to learn:

- Explicit Target
- Functional Props
- Dependency Injection
- Reply-to-Sender

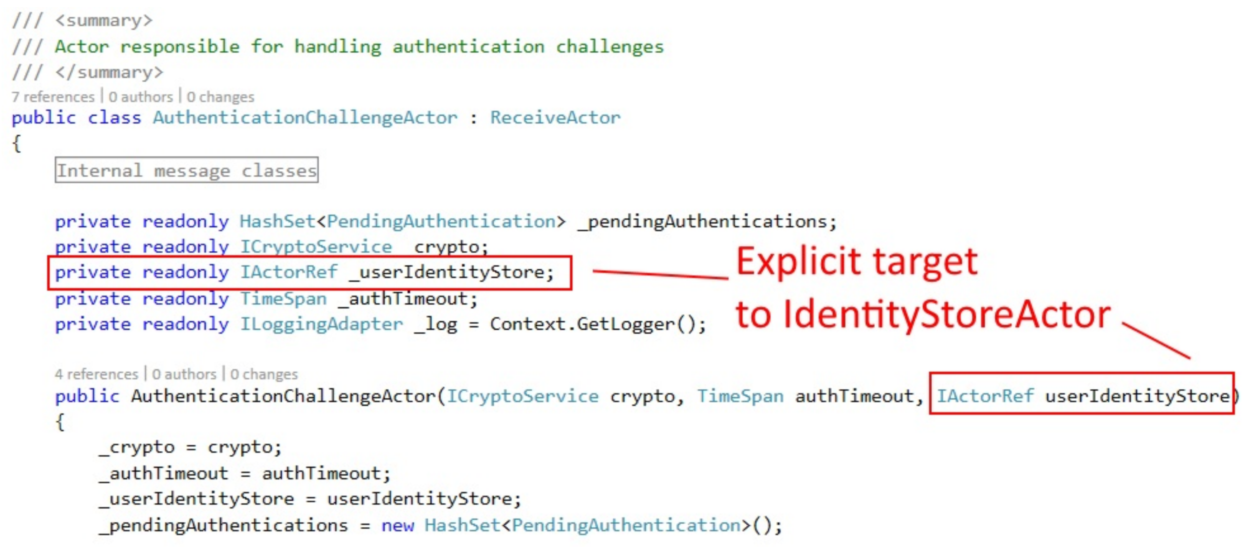
Explicit Target Pattern

When it comes to testing relationships between actors, one very useful pattern for being able to test these relationships involves passing the `IActorRef` of a dependent actor in as an argument to your primary actor.

This is called the *Explicit Target Pattern*.

Definition: The explicit target pattern occurs when `actorA`, who regularly communicates with `actorB` as part of its normal operation, holds an explicit `IActorRef` to `actorB` passed in either via constructor argument or an explicit message (such as a pub-sub subscription.)

By passing this "actor dependency" into the constructor of `actorA` we can easily mock or fake the responses of `actorB` in order to unit test the behavior of `actorA`.



```
/// <summary>
/// Actor responsible for handling authentication challenges
/// </summary>
7 references | 0 authors | 0 changes
public class AuthenticationChallengeActor : ReceiveActor
{
    Internal message classes

    private readonly HashSet<PendingAuthentication> _pendingAuthentications;
    private readonly ICryptoService _crypto;
    private readonly IActorRef _userIdentityStore;
    private readonly TimeSpan _authTimeout;
    private readonly ILoggingAdapter _log = Context.GetLogger();

    4 references | 0 authors | 0 changes
    public AuthenticationChallengeActor(ICryptoService crypto, TimeSpan authTimeout, IActorRef userIdentityStore)
    {
        _crypto = crypto;
        _authTimeout = authTimeout;
        _userIdentityStore = userIdentityStore;
        _pendingAuthentications = new HashSet<PendingAuthentication>();
```

Use Cases: The explicit target pattern is used when one actor, such as the `AuthenticationChallengeActor` in `AkkaChat`, is paired with some other actor to perform cooperative work over the course of its lifecycle. In the example above the `AuthenticationChallengeActor` is paired with an instance of `IdentityStoreActor`, whom the `AuthenticationChallengeActor` communicates with in order to verify the identity of users attempting to log into the system.

Benefits: The ultimate benefit of this pattern is that it makes it very easy to unit test the behaviors of individual actors in isolation from their dependencies.

By taking a simple `IActorRef` as a constructor argument or by accepting a message that sets the explicit target to a specific `IActorRef`, we can easily mock or fake this dependency and test actor implementations independently.

Code Sample

So we saw this chunk of code for the `AuthenticationChallengeActor` in the annotated image above.

```
public class AuthenticationChallengeActor : ReceiveActor
{
    private readonly HashSet<PendingAuthentication> _pendingAuthentications;
    private readonly ICryptoService _crypto;
    private readonly IActorRef _userIdentityStore;
    private readonly TimeSpan _authTimeout;
    private readonly ILoggingAdapter _log = Context.GetLogger();

    public AuthenticationChallengeActor(ICryptoService crypto, TimeSpan authTimeout, IActorR
    {
        _crypto = crypto;
        _authTimeout = authTimeout;
        _userIdentityStore = userIdentityStore;
        _pendingAuthentications = new HashSet<PendingAuthentication>();

        /* RECEIVE methods ... */
    }
}
```

Inside of the code of a unit test using the Akka.TestKit, we can inject an `IActorRef` to *any other type of actor* instance and test the behaviors of the `AuthenticationChallengeActor` - which is exactly what we do inside the `AkkaChat.Web.Tests` project.

```
[Test] //Using Akka.Testkit.NUnit
public void AuthChallengeActor_should_fail_challenge_on_timeout()
{
    var blackhole = Sys.ActorOf(BlackHoleActor.Props); //blackhole actor never responds to m
    var auth = Sys.ActorOf(Props.Create(() =>
        new AuthenticationChallengeActor(_cryptoService, TimeSpan.FromSeconds(3), blackhole)
    auth.Tell(new AuthenticationChallenge("Test", "test"));
    var fail = ExpectMsg<AuthenticationFailure>();
    Assert.AreEqual("Test", fail.DisplayName);
}
```

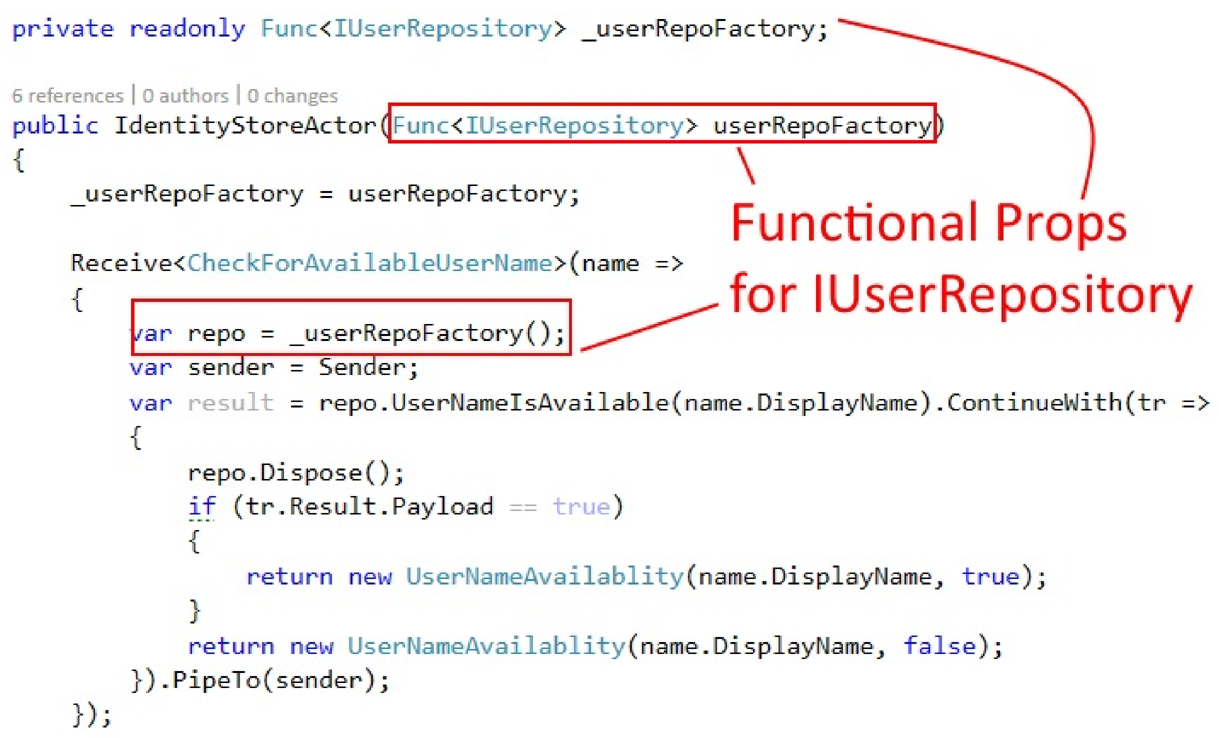
In `AuthChallengeActor_should_fail_challenge_on_timeout` we're testing to see if the `AuthenticationChallengeActor` responds as expected in the event of a timeout.

Thus, instead of sending a reply back to the `AuthenticationChallengeActor` using a normal `IdentityStoreActor` instance we use an `IActorRef` created from the the built-in `Akka.Testkit.BlackholeActor` - an actor who never replies to messages, which will create a timeout for the `AuthenticationChallengeActor`.

Functional Props Pattern

This is a pattern often used in combination with dependency injection and, in particular, instances where actors have longer lifecycles than the `IDisposable` resources they need to use when processing messages.

Description: The *Functional `Props` Pattern* occurs when an actor class, say `MyActor` in this instance, takes a dependency of type `IThingMyActorDependsOn` through its constructor. Rather than pass an instance of `IThingMyActorDependsOn` into `MyActor` directly, we pass a **generator function** that can be used to produce a *new instance* of `IThingMyActorDependsOn` each time the actor restarts or processes a message.



```
private readonly Func<IUserRepository> _userRepoFactory;

6 references | 0 authors | 0 changes
public IdentityStoreActor(Func<IUserRepository> userRepoFactory)
{
    _userRepoFactory = userRepoFactory;

    Receive<CheckForAvailableUserName>(name =>
    {
        var repo = _userRepoFactory();
        var sender = Sender;
        var result = repo.UserNameIsAvailable(name.DisplayName).ContinueWith(tr =>
        {
            repo.Dispose();
            if (tr.Result.Payload == true)
            {
                return new UserNameAvailablity(name.DisplayName, true);
            }
            return new UserNameAvailablity(name.DisplayName, false);
        }).PipeTo(sender);
    });
```

Use Cases: The functional `Props` pattern is generally used when all of the following conditions are met for actor of type `MyActor`:

- When `MyActor` takes a constructor dependency on any object which implements `IDisposable` or any object that manages its own resource lifecycle (such as acquiring or releasing temporary connections to a database.)
- When the lifecycle of `MyActor` exceeds that of the `IDisposable` resource mentioned above; and
- When `MyActor` is allowed to restart by its parent's `SupervisorStrategy`. Note: all actors by default are allowed to restart.

The idea is to ensure that an actor can safely re-acquire new instances of its `IDisposable` dependency in the event of a failure or lifecycle mismatch (it's extremely common to have `MyActor` live longer than the database connection it uses to do its work.)

Benefits: One of the tremendous benefits of functional `Props` is its ability to mock, intercept, and add additional test triggers in the course of testing the internals of an actor. You can count the number of calls to the generator function, fail the Nth call to it, and so forth.

This makes it possible to test all sorts of internal behaviors for an individual actor in ways that would normally require special testing tools or frameworks.

Code Sample

In this case we've used the functional `Props` pattern to provide new `IUserRepository` instances on-demand to the `IdentityStoreActor`.

```
/// <summary>
/// Actor reponsible for fetching users from the local system
/// </summary>
public class IdentityStoreActor : ReceiveActor
{
    private readonly Func<IUserRepository> _userRepoFactory;

    public IdentityStoreActor(Func<IUserRepository> userRepoFactory)
    {
        _userRepoFactory = userRepoFactory;

        Receive<CheckForAvailableUserName>(name =>
        {
            var repo = _userRepoFactory();
            var sender = Sender;
            var result = repo.UserNameIsAvailable(name.DisplayName).ContinueWith(tr =>
            {
                repo.Dispose();
                if (tr.Result.Payload == true)
                {
                    return new UserNameAvailablity(name.DisplayName, true);
                }
                return new UserNameAvailablity(name.DisplayName, false);
            }).PipeTo(sender);
        });
    }
}
```

From this code we can determine that for each `CheckForAvailableUserName` message received by the `IdentityStoreActor` it will call `_userRepoFactory`, the generator function, to produce a new `IUserRepository` instance.

By using a generator function, we can really easily test this behavior.

```
[Test]
```

```
public void IdentityStoreActor_should_use_new_repository_for_each_request()
{
    var callCount = 0;
    Func<IUserRepository> generatorWithCallCount = () =>
    {
        callCount++; //wrap the original _userRepoFunction with call count behavior
        return _userRepoFunction();
    };
    var identityActor = Sys.ActorOf(Props.Create(() => new IdentityStoreActor(generatorWithC
    var userNameRequest = _checkUserNameFunc();
    var expectedCalls = 4;
    foreach (var call in Enumerable.Repeat(0, expectedCalls))
    {
        identityActor.Tell(userNameRequest);
        Assert.True(ExpectMsg<IdentityStoreActor.UserNameAvailablity>().IsAvailable);
    }

    Assert.AreEqual(expectedCalls, callCount);
}
```

The `_userRepoFunction` is defined in the setup of the test fixture class, and it generates new instances of an `IUserRepository` implementation. In this case we wanted to test its instance-per-message expected behavior, so we wrapped it inside another nested function and measured the call count.

Dependency Injection

.NET developers are already overwhelmingly familiar with the practice and benefits of Dependency Injection (DI) inside their ASP.NET applications, but what's special about doing it with actors?

Description: Dependency Injection is a process by which you inject instances of dependent objects into the constructor of an actor, rather than have the actor `new()` up instances of its own dependencies. DI is often used in combination with Inversion of Control (IOC), by which a DI container gets used to resolve dependencies in a way that's reusable and generic.

Use Cases: The easiest way to break down the matter of dependency injection with Akka.NET actors is to turn it into this question:

> When is it appropriate for me to use a Dependency Injection framework like Ninject, Castle Windsor, or Autofac with my Akka.NET actors?

The following answers are valid:

1. If you are already using one of those DI frameworks inside your Windows Service, ASP.NET, or WCF application where you plan to use Akka.NET. You can just re-use them inside Akka.NET.
2. If you have dependencies, such as repositories or remote services, where the sophistication and familiarity of tools like Ninject is preferable over using your own static factory classes to inject dependencies explicitly into your actors.

Benefits: The benefits of DI are pretty straightforward - it gives you a separation of concerns and the ability to mock / fake dependencies in order to test the behaviors of individual actor classes under a variety of circumstances.

Code Sample

The purpose of spelling out this pattern inside this course isn't to teach you how DI works - it's to show you how to combine the DI tools and practices you've learned in the course of doing ASP.NET and elsewhere with Akka.NET.

So consider the following Ninject DI container we defined in `AkkaChat` for our Nancy MVC web application:

```
public partial class Startup
{
    private static IKernel SetupNinject()
    {
        var kernel = new StandardKernel(new INinjectModule[] {new FactoryModule()});

        //SQL Schema information
        var connectionString = ConfigurationManager.ConnectionStrings["akkaChat"];
        var schemaName = ConfigurationManager.AppSettings["SqlSchemaName"];
        var chatTableName = ConfigurationManager.AppSettings["ChatTableName"];
        var userTableName = ConfigurationManager.AppSettings["UsersTableName"];

        //Reuse the same instance
        kernel.Bind<SqlQueries>()
            .ToMethod(context1 => new SqlQueries(schemaName, userTableName, chatTableName))
            .InSingletonScope();

        kernel.Bind<IDbConnection>()
            .To<SqlConnection>()
            .WithConstructorArgument("connectionString", connectionString.ConnectionString);

        kernel.Bind<IUserRepository>().To<SqlUserRepository>();
        kernel.Bind<IRoomRepository>().To<SqlRoomRepository>();
        kernel.Bind<Func<IUserRepository>>().ToMethod(context1 => () => context1.Kernel.Get<
        kernel.Bind<Func<IRoomRepository>>().ToMethod(context1 => () => context1.Kernel.Get<

        //more dependencies...
        return kernel;
    }
}
```

We can use this very same `IKernel` that Nancy uses inside our Akka.NET actors by installing the `Akka.DI.Ninject` NuGet package and doing the following:

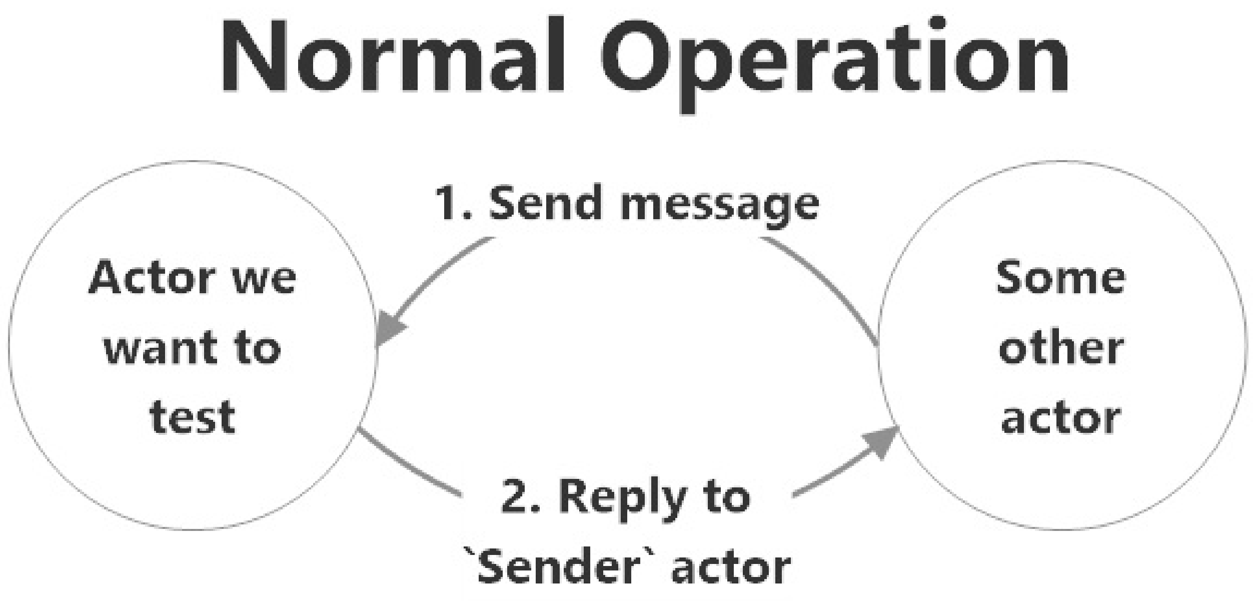
```
 public partial class Startup
{
    private static void StartActorSystem(IKernel container)
    {
        ActorSystemRefs.Kernel = container;

        /*
        * Initialize ActorSystem and essential system actors
        */
        var sys = ActorSystemRefs.System = ActorSystem.Create("AkkaChat");

        /*
       * Enable SQL Server Akka.Persistence
       */
        var resolver = ActorSystemRefs.DiResolver = new NinjectDependencyResolver(ActorSyste
        var identityProps = sys.DI().Props<IdentityManager>();
        var identity = ActorSystemRefs.Identity = sys.ActorOf(identityProps, "identity");
        container.Bind<AuthenticationChallengeActor>().ToSelf()
            .WithConstructorArgument("authTimeout", TimeSpan.FromSeconds(3.0))
            .WithConstructorArgument("userIdentityStore", identity);
        var authChallenge = ActorSystemRefs.Authenticator = sys.ActorOf(sys.DI().Props<Authe
        var sessionManager =
            ActorSystemRefs.SessionManager =
                sys.ActorOf(Props.Create(() => new UserSessionManager(identity, authChalleng

    }
}
```

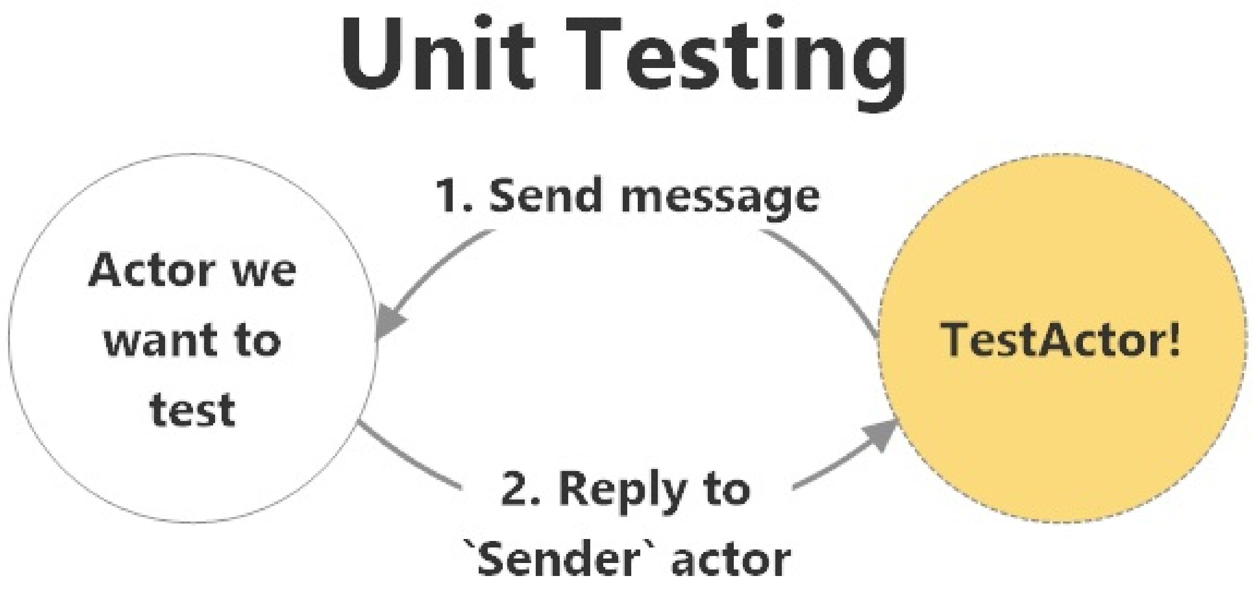
We create a `NinjectDependencyResolver` instance and stash it inside our static user-defined `ActorSystemRefs` class, so we can refer to it later. And then we create an instance of `Props` for our `IdentityManager` actor using the DI extension method: `sys.DI().Props<IdentityManager>()` , and we can pass those `Props` to the tried-and-true `ActorOf` method to create it with Ninject dependency injection.

Reply-to-Sender

Reply-to-Sender is one of those patterns that we use so naturally that we oftentimes don't think about the benefits - however, that's what we're going to explicitly spell out here.

Description: the *Reply-to-Sender Pattern* occurs when an actor (`MyActor`) is programmed to reply directly to its `Sender` field on receiving messages which merit a response, versus replying back to a fixed `ActorSelection`, to `Context.Parent`, or any other field aside from an *Explicit Target*. This pattern makes it very easy to intercept message responses `MyActor` from the outside, by simply *sending the message whose response we need to test* from an actor capable of intercepting and asserting the message.





Use Cases: this pattern is globally applicable for testing. It should be used in any instance where you want to be able to explicitly test the message replies from a specific actor.

> **This pattern works excellently in conjunction with the Akka.TestKit.TestActor, which acts as the implicit sender of all messages sent to any of your actors during unit tests.**

Benefits: The benefit of this case is ease with which you can intercept response messages from an actor which we want to test - all you have to do is send the messages you want to test from an actor capable of intercepting the replies and making them accessible for test assertions.

The built-in Akka.TestKit `TestActor` does this *automatically*.

Code Sample

The `IdentityStoreActor` uses the reply-to-sender pattern in all of its `Receive` methods, because the actor assumes that the sender of all of its supported message types is asking on behalf of itself and not some other actor. This makes it easy to test!

```
public IdentityStoreActor(Func<IUserRepository> userRepoFactory)
{
    _userRepoFactory = userRepoFactory;

    Receive<CheckForAvailableUserName>(name =>
    {
        var repo = _userRepoFactory();
        var sender = Sender;
        var result = repo.UserNameIsAvailable(name.DisplayName).ContinueWith(tr =>
        {
            repo.Dispose();
            if (tr.Result.Payload == true)
            {
                return new UserNameAvailablity(name.DisplayName, true);
            }
            return new UserNameAvailablity(name.DisplayName, false);
        }).PipeTo(sender); //REPLY TO SENDER
    });

    //other receive methods
}
```

So given that `IdentityStoreActor` will always reply back to the original sender of the actor who sent it a `CheckForAvailableUserName` message, we can use this information to test this actor's behavior from the outside by inspecting the message `IdentityStoreActor` sends in response to different types of `CheckForAvailableUserName` messages.

```
[Test]
public void IdentityStoreActor_should_not_find_nonexistent_user()
{
    var identityActor = Sys.ActorOf(Props.Create(() => new IdentityStoreActor(_userRepoFunct

    //generate a random username request
    var userNameRequest = _checkUserNameFunc();
    identityActor.Tell(userNameRequest); //TestActor is the implicit sender
    var available = ExpectMsg<IdentityStoreActor.UserNameAvailablity>().IsAvailable;
    Assert.True(available);
}
```

In this case, we send a `CheckForAvailableUserName` message for a user who doesn't exist inside the database - therefore we expect a `IdentityStoreActor.UserNameAvailablity` message in response with an `IsAvailable` property equal to `true`.

> One important thing to note here - the `ExpectMsg<IdentityStoreActor.UserNameAvailablity>()` works by checking a buffer built into the `TestActor`, and the `TestActor` is the implicit sender during this call: `identityActor.Tell(userNameRequest);`.

By replying to the sender inside the `IdentityStoreActor`, we can easily test its behaviors from the outside using special testing actors as the sender of messages during unit tests.

Next Steps

Now that we have some testability patterns covered, let's close out the *Akka.NET Design & Architecture Patterns* course by learning some **Persistence Patterns.**

Persistence Patterns

Akka.NET actors need to be able to persist messages and state to durable stores for a multitude of reasons, so in this section we present some patterns for being able to do that correctly.

Here are the patterns we will cover in this section:

- Disposable Repository Actor
- Reusable Repository Actor
- Event Sourcing with Akka.Persistence

Repository Pattern and Actor Lifecycles

Many .NET developers use the Repository Pattern on a daily basis, defining interfaces like this:

```
 /// <summary>
/// Repository for looking up <see cref="User"/> entities
/// </summary>
public interface IUserRepository : IDisposable
{
    /// <summary>
    /// Create a new <see cref="User"/>
    /// </summary>
    RepositoryResult CreateUser(User newUser);

    /// <summary>
    /// Determine if an existing username is available.
    /// </summary>
    Task<RepositoryResult<bool>> UserNameIsAvailable(string userName);

    /// <summary>
    /// Fetch an existing user from the database
    /// </summary>
    Task<RepositoryResult<ExistingUser>> FetchUser(string userName);

    /// <summary>
    /// Fetch an existing user from the database
    /// </summary>
    Task<RepositoryResult<ExistingUser>> FetchUser(Guid cookieId);
}
```

And concrete implementations built on top of SQL Server, MongoDB, Azure Table Storage, and any other implementation that you might need to run your applications in production.

This pattern is pervasive in .NET already and we're not going to spend much time talking about the merits of it. However, we do need to talk about *how to use the repository pattern with Akka.NET actors*.

Repositories Have Lifecycles

All of your repository implementations have lifecycles, dependent upon the database driver used in their implementation.

There are only three types of possible lifecycles for repositories:

1. **Per-request** - the lifecycle of a database connection or context is designed to last for exactly one request. If you're an Entity Framework user, you belong to this group.
2. **Time-delimited** - in this case a database connection will remain open a **maximum** of some explicit amount of time, such as 60 seconds, before being closed / recycled or something else. If you attempt to use a repository after this time window has elapsed then your operation will fail. *Most database drivers work this way.*
3. **Perpetual** - in this case a database connection remains open permanently until explicitly closed. These are rare in the wild, but they do exist. The Azure Table Storage driver is a good example, since it creates new HTTP connections / requests in a manner that's transparent to the caller.

In order to use repositories correctly inside your Akka.NET actors, **you must have this information** or assume the safest option: per-request lifecycle.

Disposable Repository Actor vs. Reusable Repository Actor

Once you know the lifecycle of your repositories, you can figure out how to make your actors work in concert with your repositories.

There are two strategies for doing this:

1. **Disposable Repository Actor** - temporary character actors who use a repository

instance to carry out a single operation before self-terminating. Used for infrequent, but mission-critical write operations.

2. **Reusable Repository Actor** - a long-lived character actor who manages the lifecycle of its repository directly, typically using the functional `Props` pattern from earlier.

When to Use Disposable Repository Actor vs. Reusable Repository Actor

Use Disposable Repository Actor when the lifecycle of your actor very closely follows the lifecycle of your repository, and the work is sufficiently complicated that it merits its own actor.

Use the Reusable Repository Actor by default otherwise, but be sure to manage the lifecycle of your repository within the actor correctly. When in doubt, assume that the lifecycle is per-request.

Code Example: Disposable Repository Actor

The `IdentityWriterActor` in `AkkaChat` is a perfect example of a disposable repository actor - the `IdentityManager` creates one `IdentityWriterActor` per `CreateUser` request.

```
public class IdentityWriterActor : ReceiveActor
{
    private IUserRepository _repository;

    public IdentityWriterActor(IUserRepository repository)
    {
        _repository = repository;

        Receive<CreateUser>(user =>
        {
            var sender = Sender;
            repository.UserNameIsAvailable(user.DisplayName)
                .ContinueWith(tr =>
                {
                    return new CanProceed<CreateUser>(user, sender, tr.Result.Payload);
                })
                .PipeTo(Self);

            // time this operation out if we don't hear back within three seconds
            SetReceiveTimeout(TimeSpan.FromSeconds(3));
        });

        //shut down in the event of a timeout
```

```
            Receive<ReceiveTimeout>(timeout => Context.Stop(Self));

            Receive<CanProceed<CreateUser>>(proceed =>
            {
                SetReceiveTimeout(null); //disable ReceiveTimeout
                if (proceed.CanExecute)
                {
                    var createUser = proceed.OriginalMessage;

                    // create the user
                    // TODO: could use AutoMapper here instead of manually doing the mapping.
                    var result = repository.CreateUser(new User()
                    {
                        DateJoined = createUser.DateJoined,
                        DisplayName = createUser.DisplayName,
                        Email = createUser.Email,
                        PasswordHash = createUser.PasswordHash,
                        PasswordSalt = createUser.PasswordSalt,
                        CookieId = createUser.CookeId
                    });

                    if (result.Status == ResultCode.Success)
                    {
                        //notify the original sender that we successfully created the user
                        proceed.IntendedDestination.Tell(new UserCreateResult(createUser.Display
                        Context.Stop(Self); //shut ourselves down
                        return; //done working
                    }
                }

                //let the caller know that we failed
                proceed.IntendedDestination.Tell(new UserCreateResult(proceed.OriginalMessage.Di
            });
        }

        protected override void PostStop()
        {
            //cleanup any repositories or disposable resources we might have used
            try
            {
                _repository.Dispose();
            }
            catch { }

            base.PostStop();
        }
    }
```

Once the `IdentityWriterActor` is done working it will call `Context.Stop` on itself to begin the shutdown process, which will cause its `PostStop` method to be called where the `IUserRepository` instance injected into the constructor will be disposed.

We used the disposable repository actor pattern here because:

- `CreateUser` operations are infrequent, but mission critical;
- There's a lot of plumbing involved in making sure the `CreateUser` operation can be completed successfully; and
- The lifecycle of a SQL Server connection, which `AkkaChat` uses, is time-delimited.

Code Example: Reusable Repository Actor

The `IdentityStoreActor` inside `AkkaChat` lives for the entire length of the `AkkaChat` application and is responsible for handling all of the database-read queries related to users, so it has to be able to manage the lifecycle of its own `IUserRepository` instances.

```
public IdentityStoreActor(Func<IUserRepository> userRepoFactory)
{
    _userRepoFactory = userRepoFactory;

    Receive<CheckForAvailableUserName>(name =>
    {
        var repo = _userRepoFactory();
        var sender = Sender;
        var result = repo.UserNameIsAvailable(name.DisplayName).ContinueWith(tr =>
        {
            repo.Dispose();
            if (tr.Result.Payload == true)
            {
                return new UserNameAvailablity(name.DisplayName, true);
            }
            return new UserNameAvailablity(name.DisplayName, false);
        }).PipeTo(sender); //REPLY TO SENDER
    });

    //other receive methods
}
```

The `IdentityStoreActor` assumes per-request lifecycle behavior from its `IUserRepository` instances, thus it creates and disposes its own `IUserRepository` instance on *every message*. This is perfectly acceptable to do in production - it's what your ASP.NET MVC controllers do already.

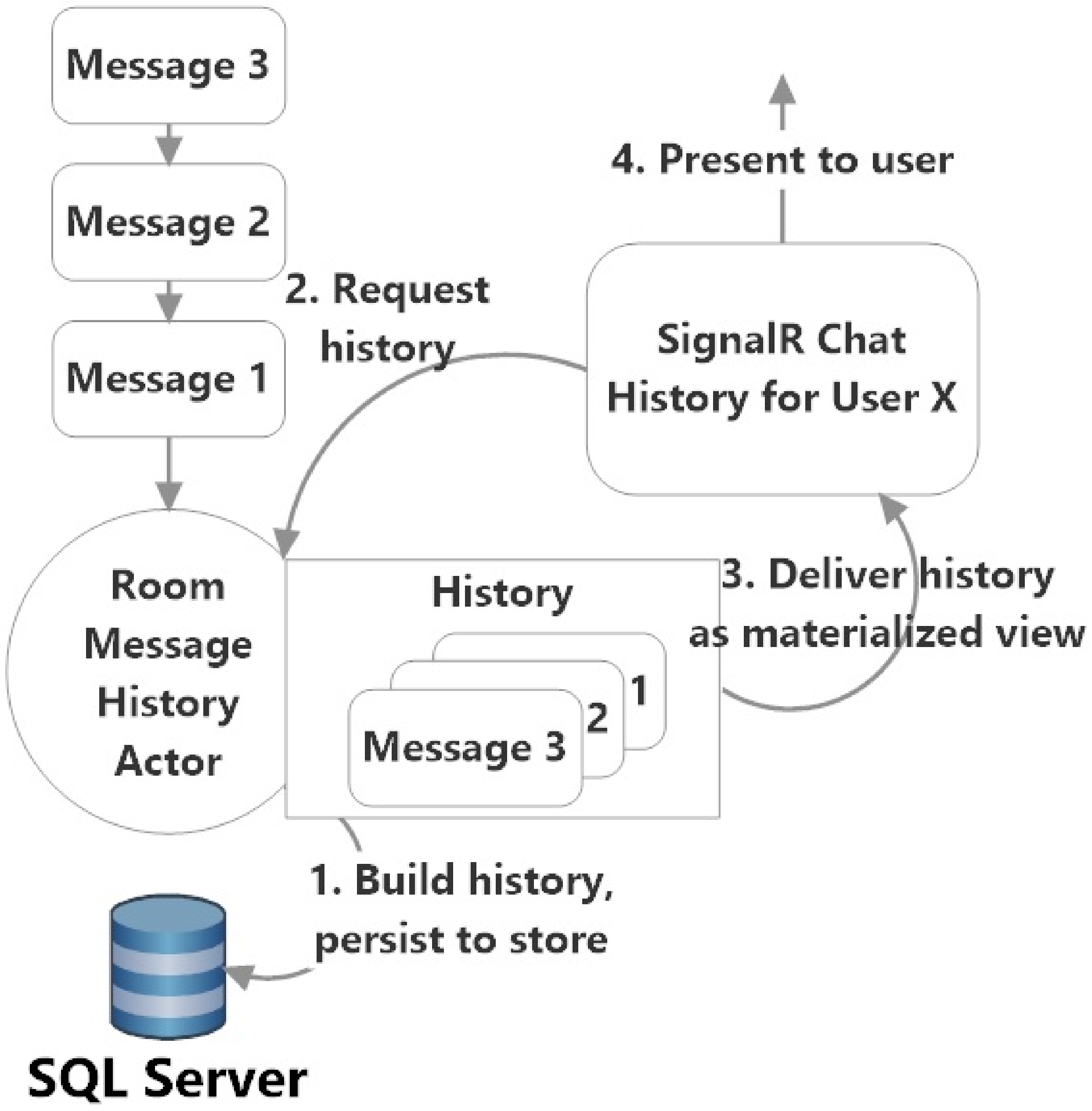
Event Sourcing with Akka.Persistence

One of the exciting new packages to come to Akka.NET is Akka.Persistence, a library which gives Akka.NET actors the ability to easily persist messages and actor state to a durable store such as SQL Server, Postgres, EventStore, Cassandra, and more.

This allows to us to create some immensely powerful delivery and persistence patterns, but the one we're going to focus on here is the *Event Sourcing Pattern*.

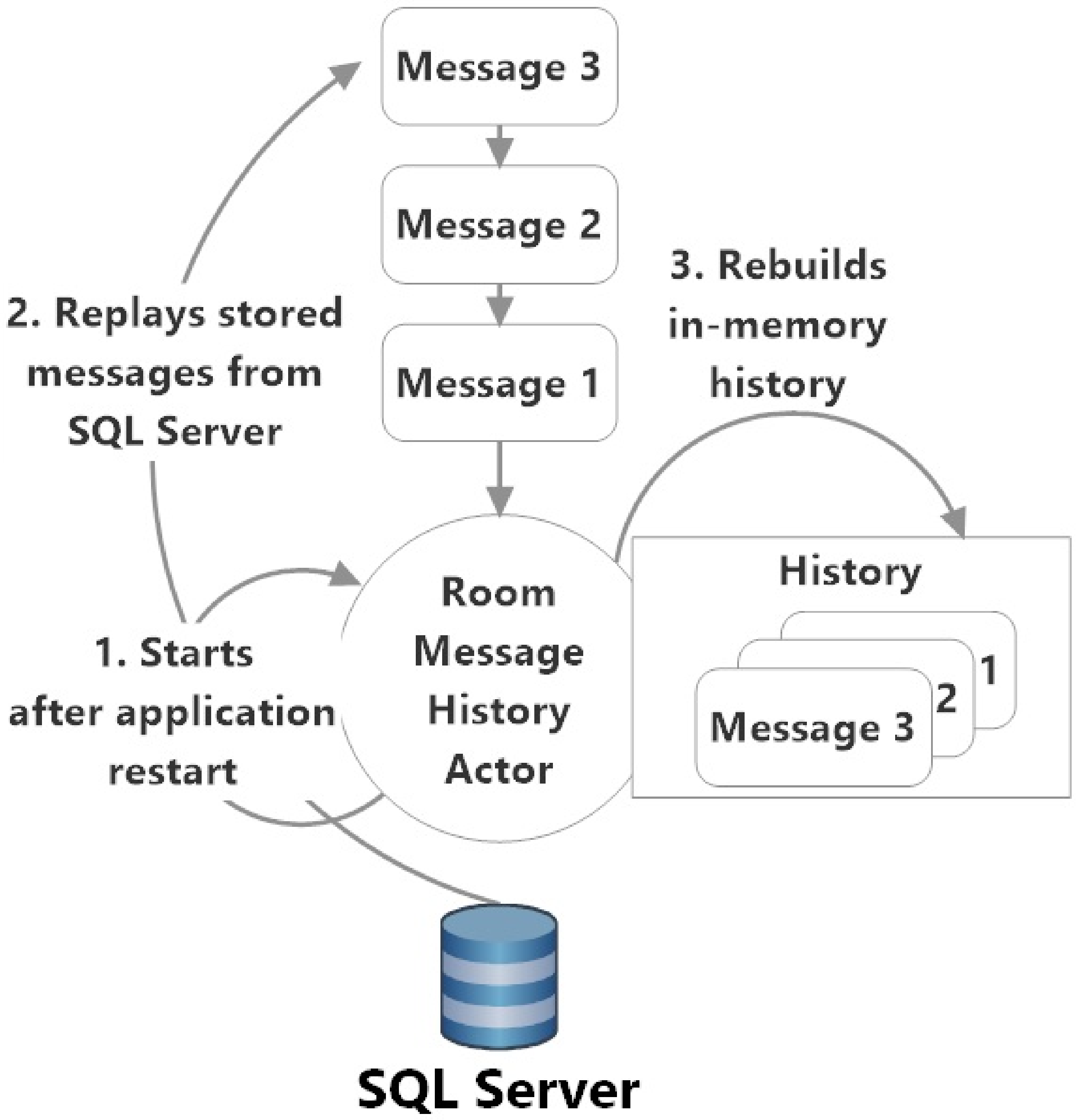
Definition: We're going to borrow the Event Sourcing definition from Microsoft's Cloud Design Patterns content.

> The Event Sourcing pattern defines an approach to handling operations on data that is driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they are persisted. Each event represents a set of changes to the data (such as AddedItemToOrder).
>
> The events are persisted in an event store that acts as the source of truth or system of record (the authoritative data source for a given data element or piece of information) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that is required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.


Message 3
Message 2
Message 1
2. Request history
4. Present to user
SignalR Chat History for User X
Room Message History Actor
History
Message 3
2
1
3. Deliver history as materialized view
1. Build history, persist to store
SQL Server

In the case of our `RoomMessageHistoryActor`, which uses Akka.Persistence's journaling capabilities to build an event source on top of SQL Server, we present the entire history of a particular chatroom as an object composed of the 30 most recent previous messages, and this information is sent to each new user as soon as they connect.

But what makes Akka.Persistence's capabilities really cool is the ability to recover `RoomMessageHistoryActor`'s state in the event of restarting the `AkkaChat` application OR restarting the actor!


Message 3
Message 2
Message 1
2. Replays stored messages from SQL Server
3. Rebuilds in-memory history
1. Starts after application restart
Room Message History Actor
History
Message 3
2
1
SQL Server

In Akka.Persistence, all `PersistentActors` can recover their state automatically from their persistent store on restarts.

Use Cases: When would you want to use event sourcing? When the following conditions are true:

1. You don't care about the queryability of your persisted events and messages - they're not going to be used in analytics or ad-hoc queries any time soon.
2. When it's easy to model changes in your state as per-message mutations - works particularly well with the chat history in the case of `AkkaChat`.

When you want to use the `AtLeastOnceDeliveryActor` for guaranteed delivery of messages, since the persisted messages are effectively transient anyway.

Benefits: The benefits of event sourcing are enormous:

1. Plays nicely with journaling and snapshots in Akka.Persistence;
2. Plays nicely with the message-passing model used by all Akka.NET actors;
3. Can "undo" events to mutate back into an earlier state; and
4. Naturally provides CQRS behavior to your actors who use it.

Code Sample

The first thing to look at is our Akka.Persistence configuration for `AkkaChat`:

```
<akka>
<hocon>
  <![CDATA[
      akka {
        actor{
          deployment {
            /signalr-writer{
              router = broadcast-pool
              nr-of-instances = 1
            }

            /signalr-reader{
              router = broadcast-pool
              nr-of-instances = 1
            }
          }
        }
        persistence{
          journal {
            plugin = "akka.persistence.journal.sql-server"
            sql-server {
                class = "Akka.Persistence.SqlServer.Journal.SqlServerJournal, Akka.Persisten
                schema-name = dbo
                auto-initialize = on
                connection-string = "Data Source=(LocalDB)\\v11.0;AttachDbFilename=|DataDire
            }
          }
          snapshot-store{
            plugin = "akka.persistence.snapshot-store.sql-server"
            sql-server {
                class = "Akka.Persistence.SqlServer.Snapshot.SqlServerSnapshotStore, Akka.Pe
                schema-name = dbo
                auto-initialize = on
                connection-string = "Data Source=(LocalDB)\\v11.0;AttachDbFilename=|DataDire
            }
          }
        }
      }
  ]]>
</hocon>
</akka>
```

We're going to be using SQL Server for both Akka.Persistence journals and snapshots.

Snapshots vs. Journals

Akka.Persistence offers two varieties of storage mediums:

- **Snapshots** - a snapshot is the representation of the entire internal state of a `PersistentActor` taken at any given point in time. When a `PersistentActor` recovers it will load only its most recent snapshot.
- **Journals** - journals represent an ordered set of messages belonging to a particular `PersistentActor`. Each individual message is saved as its own record. When a `PersistentActor` recovers it will load **ALL** of its journaled messages.

Journals are what we'll be using to create an event source inside our `RoomMessageHistoryActor` inside `AkkaChat`.

Creating Our `PersistentActor` Implementations

We're going to define the `RoomMessageHistoryActor` as a `ReceivePersistentActor` - a `PersistentActor` that works with `ReceiveActor` capabilities.

The first thing we need to do is define a state container to act as our in-memory event source. We'll use the following `MessageHistory` class for that.

```
 /// <summary>
/// In-memory EventStore for the message history for a given chatroom
/// </summary>
public class MessageHistory
{
    public MessageHistory(IEnumerable<ChatMessage> messages = null)
    {
        Messages = new SortedSet<ChatMessage>(messages ?? new List<ChatMessage>());
    }

    public IEnumerable<ChatMessage> Messages { get; private set; }
    public bool WillUpdate(ChatMessage message)
    {
        return !Messages.Contains(message);
    }

    public MessageHistory Update(ChatMessage message)
    {
        return new MessageHistory(Messages.Concat(message));
    }
}
```

The `ChatMessage` class implements `IComparable`, which is what allows us to time-order the chat messages so easily.

```
public class ChatMessage : IComparable<ChatMessage>, IRouteToRoom
{
    public ChatMessage(string id, string message, DateTime when, string roomName, string use
    {
        UserName = userName;
        RoomName = roomName;
        When = when;
        Message = message;
        Id = id;
    }

    public string Id { get; private set; }

    public string UserName { get; private set; }

    public string RoomName { get; private set; }

    public string Message { get; private set; }

    public DateTime When { get; private set; }

    public int CompareTo(ChatMessage other)
    {
        return When.CompareTo(other.When);
    }

    protected bool Equals(ChatMessage other)
    {
        return string.Equals(Id, other.Id) && When.Equals(other.When) && string.Equals(RoomN
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;
        if (obj.GetType() != this.GetType()) return false;
        return Equals((ChatMessage) obj);
    }
    public override int GetHashCode()
    {
        unchecked
        {
            var hashCode = (Id != null ? Id.GetHashCode() : 0);
            hashCode = (hashCode*397) ^ When.GetHashCode();
            hashCode = (hashCode*397) ^ (RoomName != null ? RoomName.GetHashCode() : 0);
            return hashCode;
        }
    }
}
```

When we journal the messages our `RoomMessageHistoryActor`, Akka.Persistence will be serializing and persisting `ChatMessage` objects to SQL Server.

Now let's take a look at what the `RoomMessageHistoryActor` actually does!

```
/// <summary>
/// PERSISTENT ACTOR.
///
/// Responsible for managing a single chatroom's message history.
/// </summary>
public class RoomMessageHistoryActor : ReceivePersistentActor
{
    private readonly ExistingRoom _room;

    private MessageHistory _history;

    private string _cachedPersistenceId;
    public override string PersistenceId
    {
        get
        {
            if (string.IsNullOrEmpty(_cachedPersistenceId))
            {
                _cachedPersistenceId = string.Format("room-{0}-messages", _room.Id);
            }
            return _cachedPersistenceId;
        }
    }

    private readonly IActorRef _signalRWriter;

    public RoomMessageHistoryActor(ExistingRoom room, IActorRef signalRWriter)
    {
        _signalRWriter = signalRWriter;
        _room = room;
        _history = new MessageHistory();
        ReadyCommands();
```

```
            ReadyRecovers();
        }

        private void ReadyCommands()
        {
            //SystemMessages don't get added to the message history
            Command<SystemMessage>(message =>
            {
                _signalRWriter.Forward(message);
            });

            //UserMessages do get persisted
            Command<UserMessage>(message => Persist(message, UpdateHistory));

            //Need to catch specific SignalR user up on the messages
            Command<FetchPreviousMessages>(previous =>
            {
                var queryTime = previous.Since ?? DateTime.MaxValue;
                var messages = _history.Messages.Where(x => x.When <= queryTime).OrderBy(x => x.
                _signalRWriter.Tell(new HistoricalMessageBatch(messages, previous.RoomName, prev
            });
        }

        private void UpdateHistory(ChatMessage message)
        {
            if (!_history.WillUpdate(message)) return;
            _signalRWriter.Forward(message);
            _history = _history.Update(message);
        }

        private void ReadyRecovers()
        {
            Recover<UserMessage>(message => RecoverHistory(message));
        }

        private void RecoverHistory(ChatMessage message)
        {
            _history = _history.Update(message);
        }
    }
```

The first thing to notice is the overridden `PersistenceId` property - all `PersistenceActor` implementations are required to override this property and return a valid string. **This property is effectively the unique row key for all messages that belong to this actor instance. So make sure no two persistent actors can accidentally use the same PersistenceId!**

In this case we set the `PersistenceId` to be some function of the chat room ID that this `RoomMessageHistoryActor` instance is recording messages for.

Next, notice that we don't have any `Receive` methods inside this `ReceiveActor`. Instead we have `Recover<TMessage>` and `Command<TMessage>` - what's that about?

Recovers vs. Commands

All `PersistentActor`s have to distinguish between two different types of received messages:

1. **Commands** - these are the equivalent of `Receive` methods; they receive new messages from other actors and the `PersistentActor` can choose to save those commands into a journal or snapshot or both.
2. **Recovers** - these are *stored messages* that are being pushed into a `PersistentActor` by the durable store. Recovers happen when the actor restarts and needs to recover its state from its durable store.

So when the `RoomMessageHistoryActor` receives stored `ChatMessage` objects via its `Recover<UserMessage>(message => RecoverHistory(message));` hook, it just pushes those messages back onto its in-memory `MessageHistory` object.

When the `RoomMessageHistoryActor` receives `UserMessage` objects via its `Command<UserMessage>(message => Persist(message, UpdateHistory));` command, it'll first save the `UserMessage` to SQL Server and then run the `UpdateHistory` method afterwards, which looks like this:

```
private void UpdateHistory(ChatMessage message)
{
    if (!_history.WillUpdate(message)) return;
    _signalRWriter.Forward(message);
    _history = _history.Update(message);
}
```

This is the correct way to journal messages - guarantee that the message has been written to the journal before "handling" the rest of the message and updating the actor's in-memory state.

Conclusion

This marks the end of Petabridge's *Akka.NET Design & Architecture Patterns* course! We hope you enjoyed it!

You should consider taking our other training courses:

- **Akka.NET Remoting** - learn how to connect remote actor systems together, do remote deployments, and pass messages remotely between actors.
- **Akka.NET Clustering** - learn how to build resilient self-healing Akka.NET clusters using Akka.Cluster. Perfect for developers interested in building distributed systems with Akka.NET.