Building Actor Model systems using Akka.NET

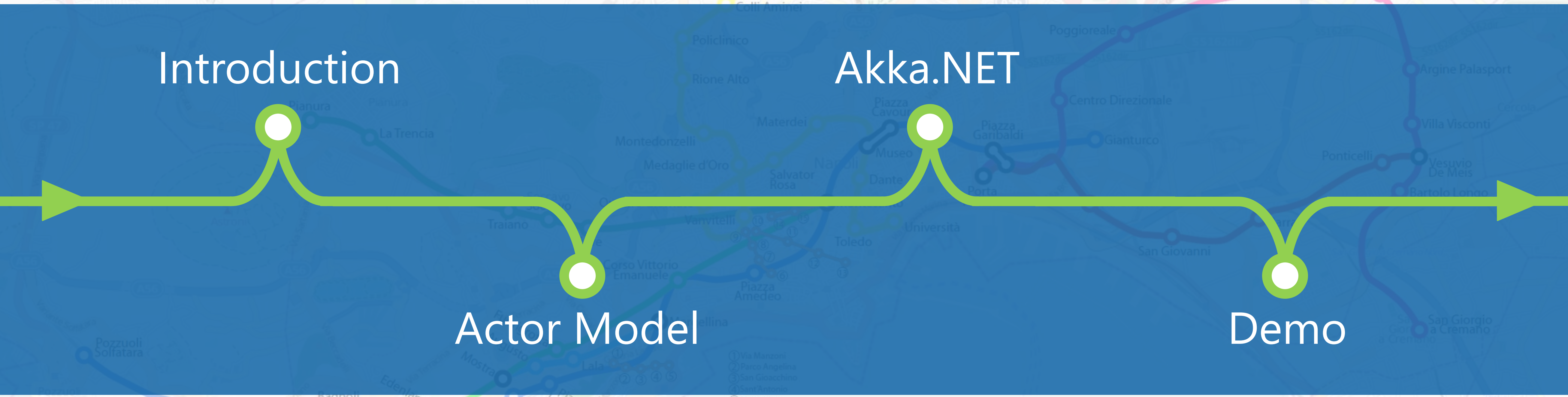
Edwin van Wijk
Principal Software Architect at Info Support

MVP Microsoft® Most Valuable Professional

edwinw@infosupport.com
@evanwijk
blog defaultconstructor.com

Info Support Solid Innovator

Session Roadmap



Building HA systems handling large concurrent work-loads is hard!

https://upload.wikimedia.org/wikipedia/commons/6/6d/Multiple_Server_.jpg

Threading

https://upload.wikimedia.org/wikipedia/commons/9/96/Colorful_Threads_(3965274345).jpg

Lock / Mutex / Semaphore

Actor Model paradigm

Introduced by Carl Hewitt in 1973

A model for building HA/scalable distributed systems

Primarily for building systems that need to handle large concurrent workloads

Battle-tested in several large scale systems

e.g. *Erlang* was created by Ericsson to handle large amounts of network packets in a telephony switch

Several frameworks for implementing Actor Model

Akka, **Akka.NET**, Erlang, Orleans, Quasar, PostSharp, Azure Service Fabric, …

Akka.NET

Port of the JVM Akka library to the CLR

Open Source project started by Aaron Stannard and Roger Alsing

Commercial support / training / consultancy available from Petabridge

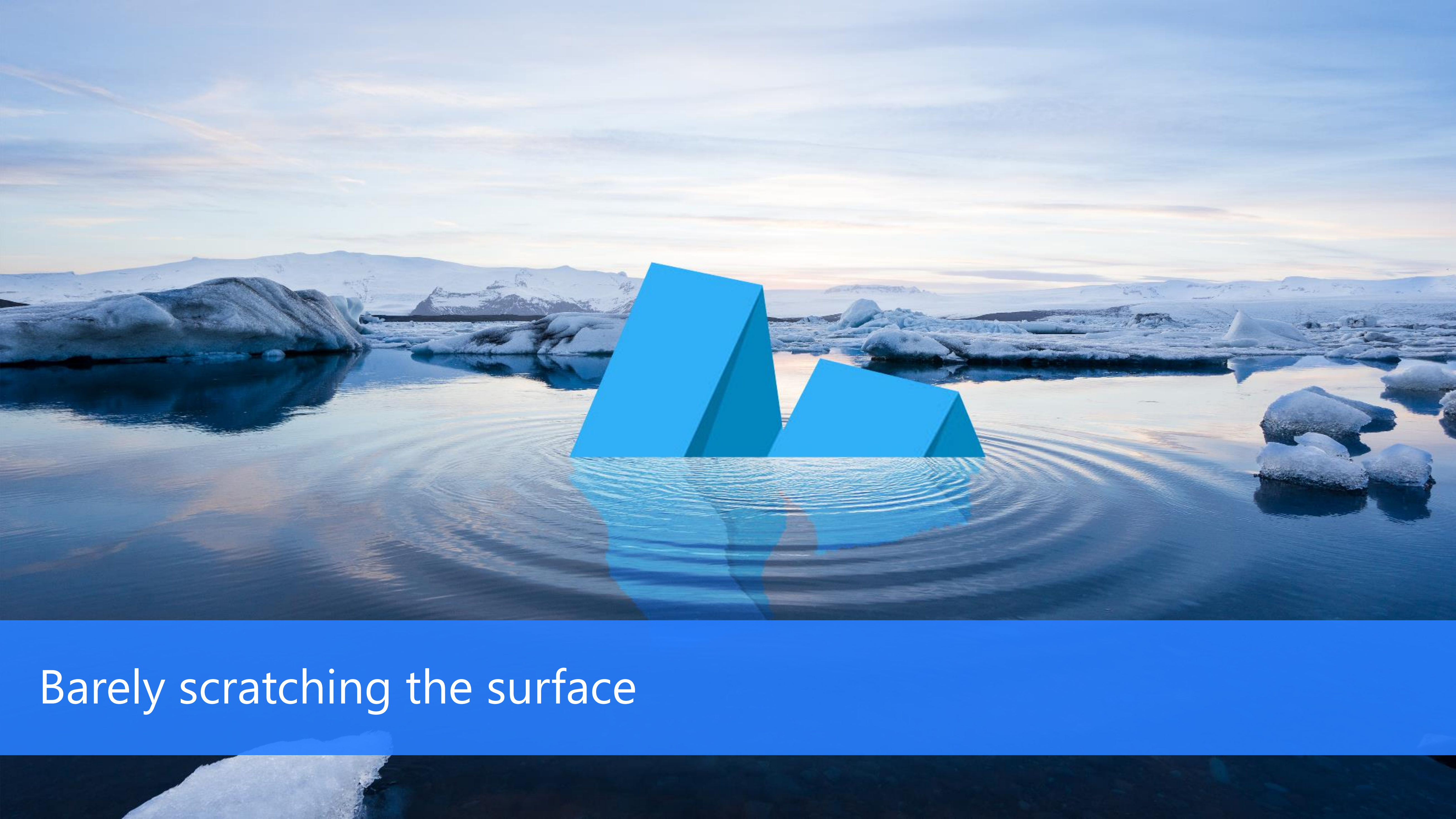
Framework and runtime

Using C# or F# on .NET or Mono

Hosted within a .NET process (Console App, Windows Service, ASP.NET Web App, Azure, …)

Highly extensible

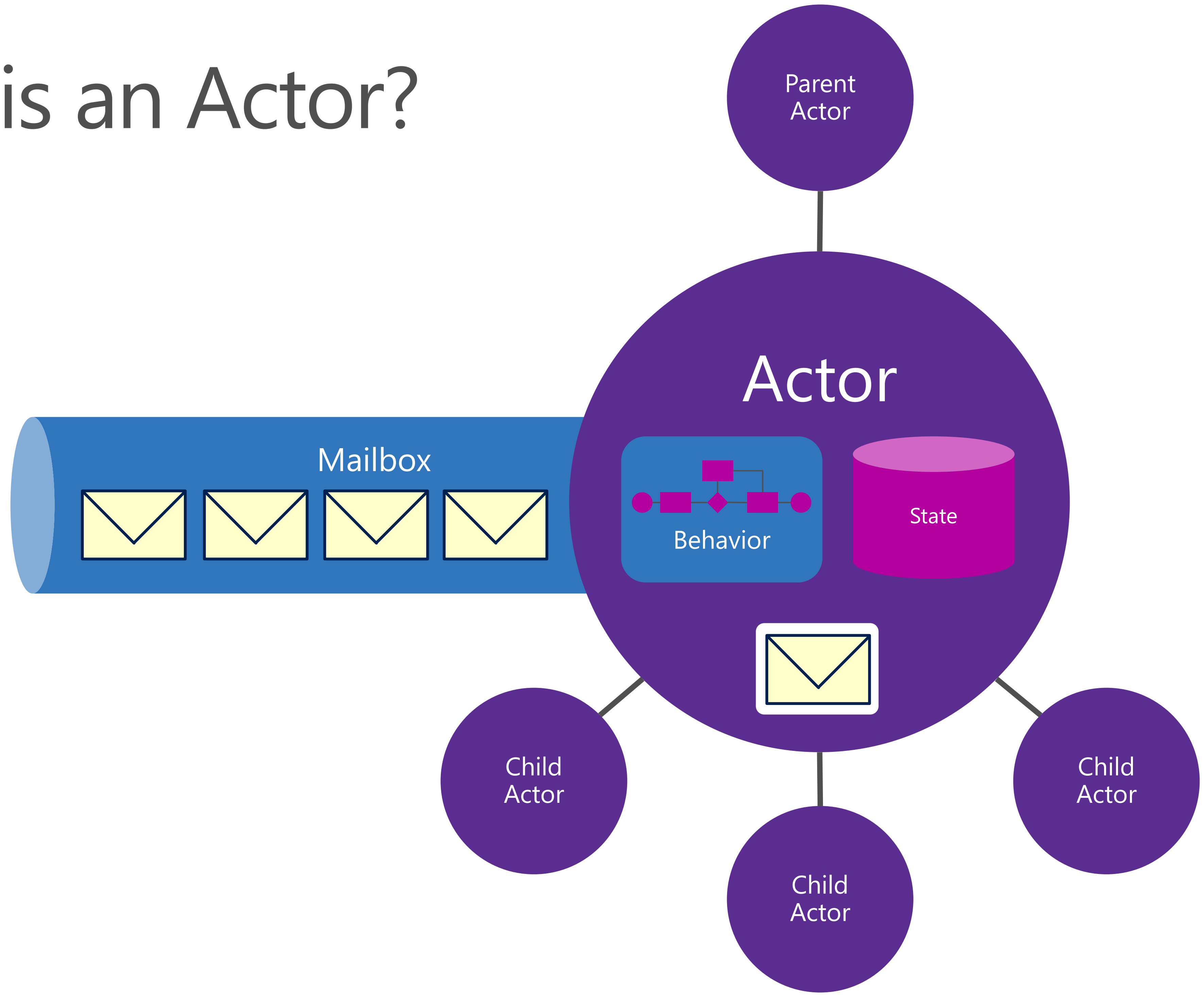
Distributed through NuGet



Barely scratching the surface

Actors

What is an Actor?



Actor implementation

```
public class ActorA : ReceiveActor
{
    public ActorA()
    {
        Receive<SomeMessage>(msg => Console.WriteLine(msg.Payload));
        Receive<SomeOtherMessage>(msg => Handle(msg));
    }

    private void Handle(SomeOtherMessage message)
    {
        if (message.IsEnabled)
        {
            Console.WriteLine(message.Payload);
        }
    }
}
```

Actor implementation

```
public class ActorB : TypedActor, IHandle<SomeMessage>, IHandle<SomeOtherMessage>
{
    public void Handle(SomeMessage message)
    {
        Console.WriteLine(message.Payload);
    }

    public void Handle(SomeOtherMessage message)
    {
        if (message.IsEnabled)
        {
            Console.WriteLine(message.Payload);
        }
    }
}
```

Actors form a hierarchy

Work is off-loaded to child actors

Especially “dangerous” work

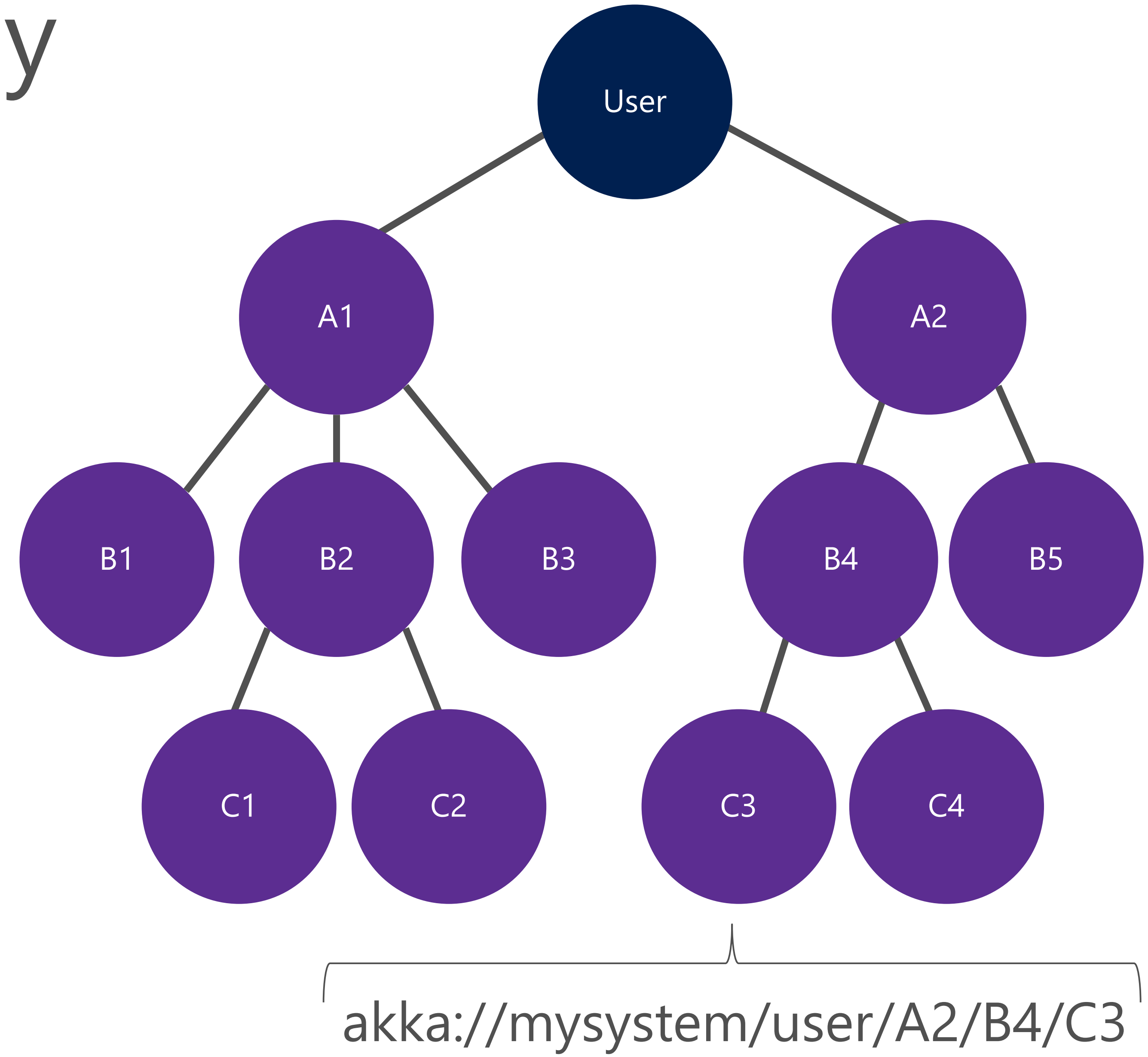
Routers can also be used to divide work

Each actor has a unique address within the hierarchy

Like a file-system folder structure

Every actor can communicate with every other actor directly

By absolute or relative address



Communication

Communicating with Actors

Creating an Actor yields an *IActorRef* instance

Consider this the proxy for your actor

You never get a direct reference to the actor instance

You can *Tell* or *Ask* an Actor something

An immutable message (POCO) is sent as payload

Avoid Ask (request – response) as much as you can

You can find an Actor using an *ActorSelection*

Use the Actor's path (absolute or relative) to locate it

Wildcards can be used

Communicating with Actors

```
using (ActorSystem mySystem = ActorSystem.Create("mySystem"))
{
   IActorRef a1 = mySystem.ActorOf<ActorA>("A1");
   a1.Tell(new SomeMessage("payload"));
}
```

```
// inside actor
IActorRef actorB = Context.ActorOf<ActorB>("B1");
actorB.Tell(new SomeMessage("payload"));
```

```
// inside actor
ActorSelection actorB = Context.ActorSelection("/user/A1/B1");
actorB.Tell(new SomeOtherMessage("payload"));
```

Communicating with Actors

```
// inside actor
Sender.Tell(new SomeMessage("payload"));

Context.Parent.Tell(new SomeMessage("payload"));

Self.Tell(new SomeMessage("payload"));

Context.Child("B1").Tell(new SomeMessage("payload"));
```

Supervision

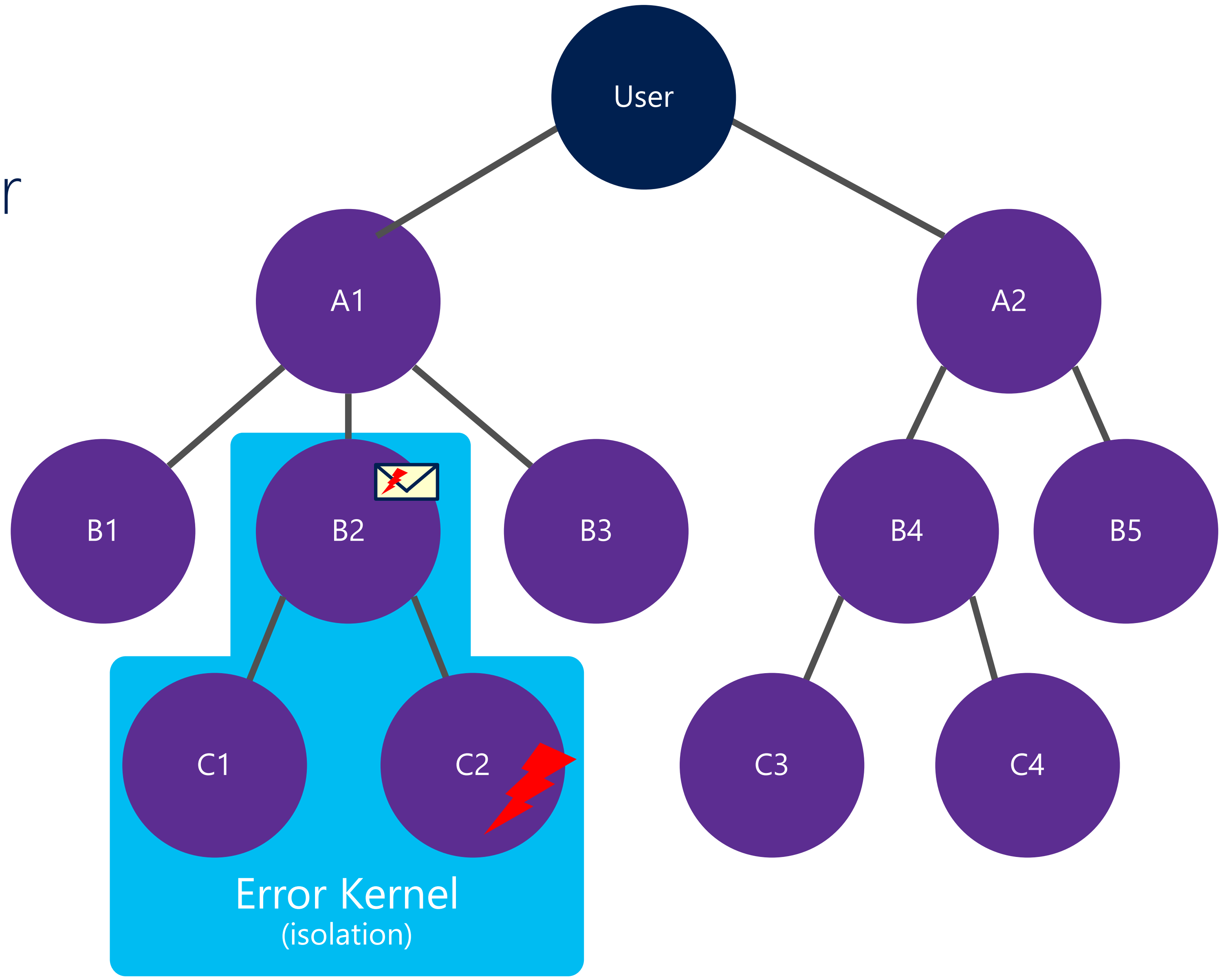
Supervision

Every actor is the supervisor of its children

Errors are communicated using system messages

The parent decides how to handle the error

Resume, Restart, Stop, Escalate



Supervision

```
// inside parent actor
protected override SupervisorStrategy SupervisorStrategy()
{
    return new OneForOneStrategy(
        5, // max. 5 exceptions ...
        TimeSpan.FromMinutes(1), // ... during a 1 minute period
        (ex) => {
            return
                ex is NotImplementedException ? Directive.Resume :
                ex is ArgumentException ? Directive.Restart :
                ex is NullReferenceException ? Directive.Stop :
                Directive.Escalate;
        });
}
```

Location Transparency

Akka.Remote

Akka.Remote offers location transparency

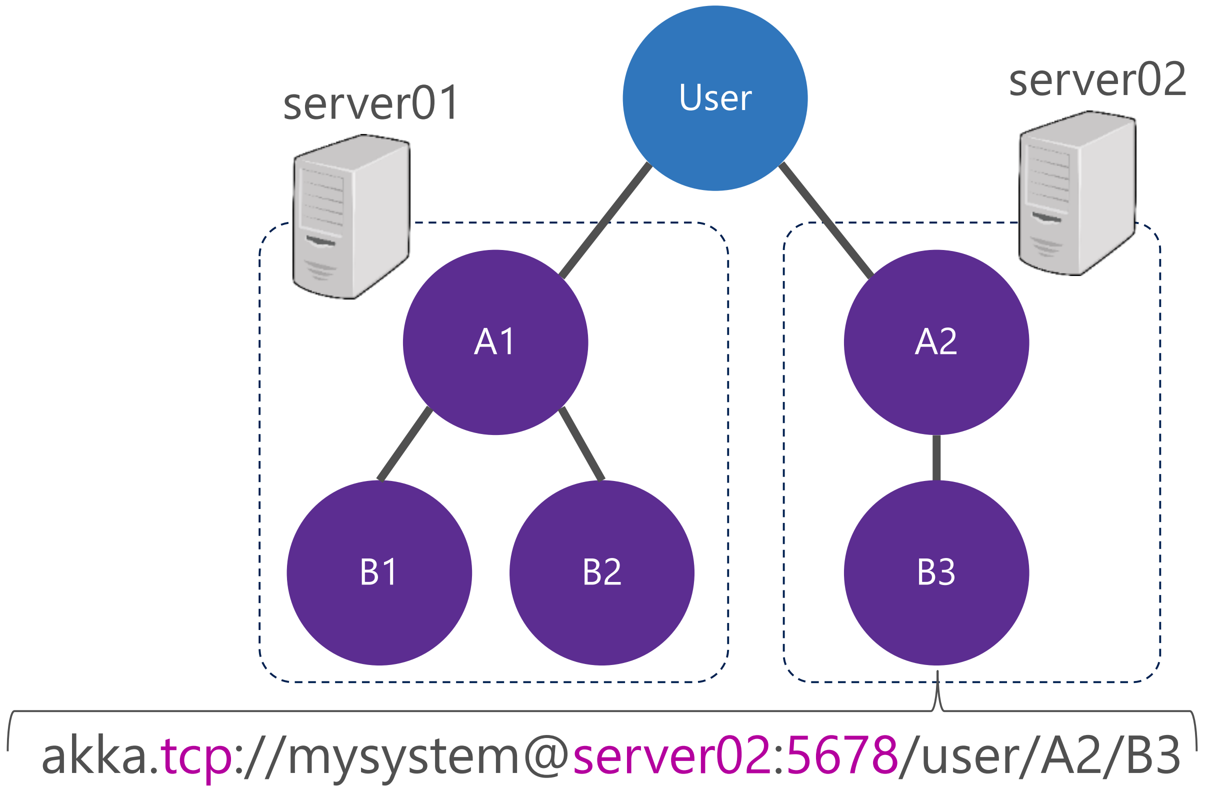
RemoteActorRef acts as proxy

For the developer there's no difference with working with local actors

Address contains transport, host and port

Actors can be deployed in a remote ActorSystem

Akka.Cluster adds another abstraction layer offering fault-tolerant elastic scaling over multiple nodes



Persistence

Akka.Persistence

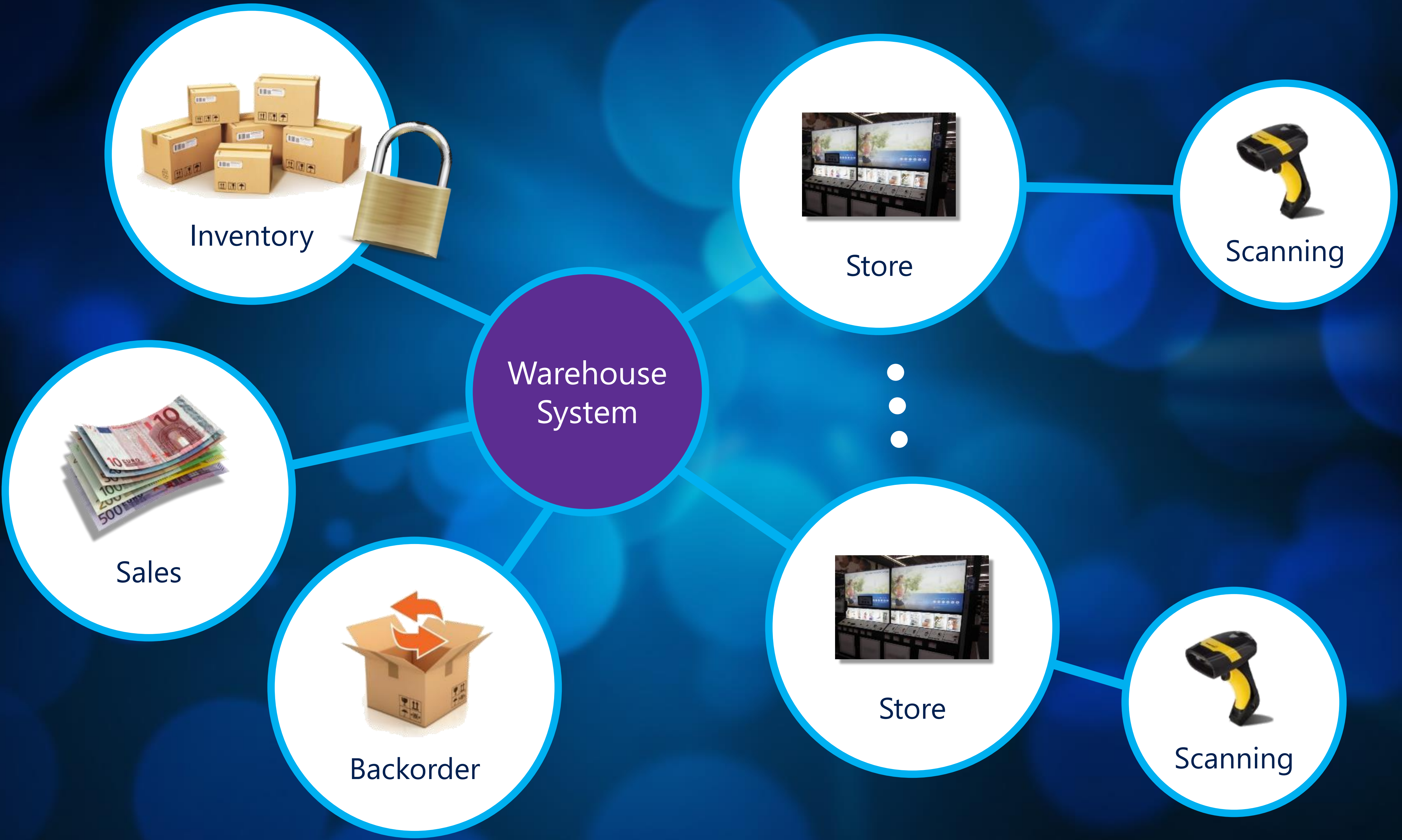
- Offers event-sourced persistence of actor state
  - Works by persisting the events that have occurred (read-only)
- Rebuild state by replaying events in chronological order
  - Specific overloads exist for handling replay
  - Snapshot are supported
- Persistence store is pluggable
  - Sql Server, MongoDB, Redis, Cassandra, Azure Table Storage, …

Akka.Persistence

```
public class CounterActor : ReceivePersistentActor
{
  private int _total = 0;

  public override string PersistenceId { get; } = "unique-counter-id" ;

  public CounterActor()
  {
    Command<Add>(cmd => Persist(new AmountAdded(cmd.Amount), Add));
    Command<Subtract>(cmd => Persist(new AmountSubtracted(cmd.Amount), Subtract));
    Recover<AmountAdded>(evt => Add(evt));
    Recover<AmountSubtracted>(evt => Subtract(evt));
  }

  private void Add(AmountAdded evt) { _total += evt.Amount; }

  private void Subtract(AmountSubtracted evt) { _total -= evt.Amount; }
}
```

Demo application

Wearable Fitness Technology
Wearable Fitness Tech
Get in Sync to Get Fit
Get in Sync to Get Fit
Q
14


Inventory
Warehouse System
Sales
Backorder
Store
Scanning
Store
Scanning

Demo application - first try

Store

Per store a single actor is created
Customer actors are created to simulate a customer
Customer actor creates and activates a scanner

Sales

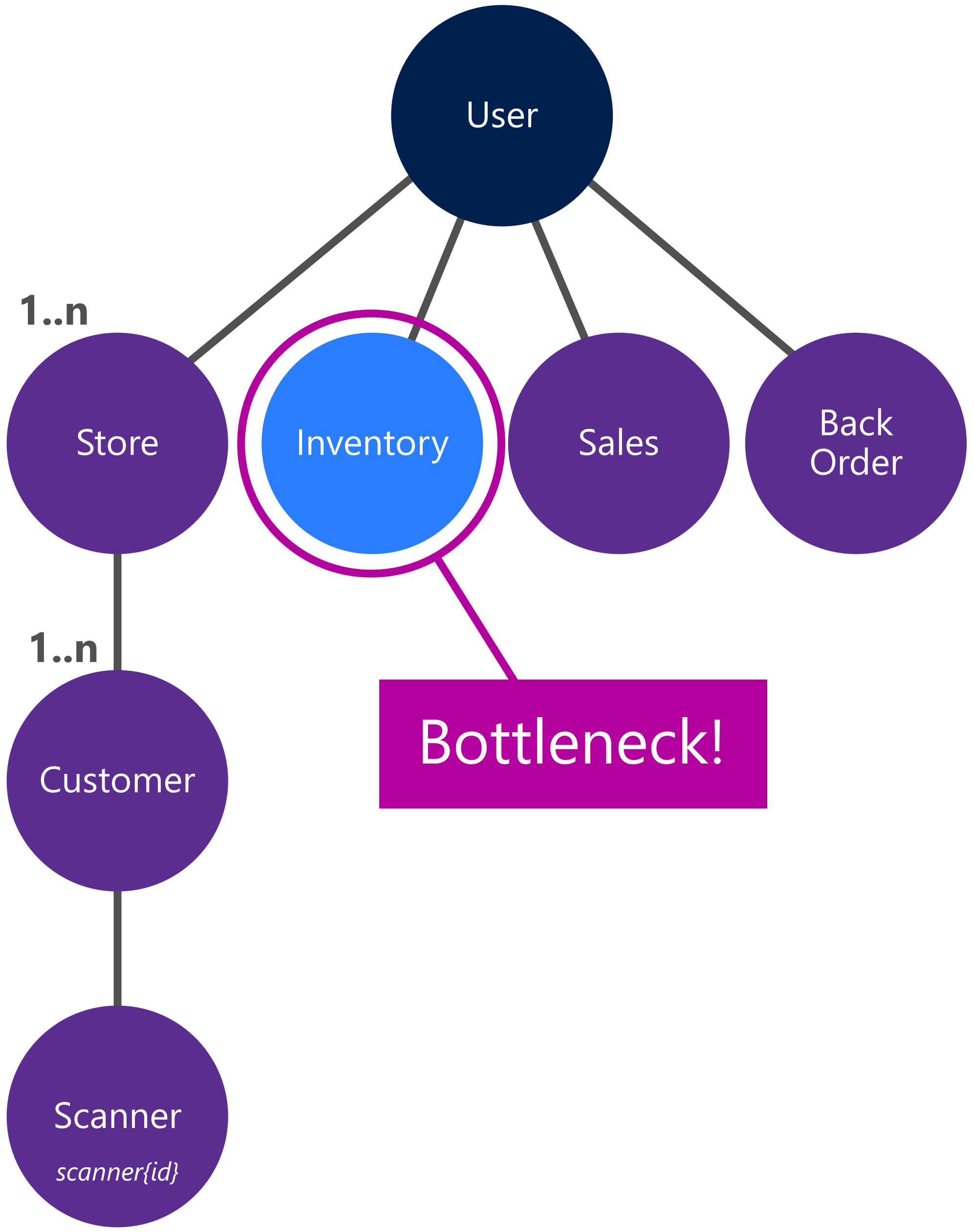
Accumulates sales numbers

Back Order

Back-orders products that are out of stock

Inventory

Handles product purchases
Having 1 inventory actor handles concurrency



Demo application - refactored

Store

Per store a single actor is created
Customer actors are created to simulate a customer
Customer actor creates and activates a scanner

Sales

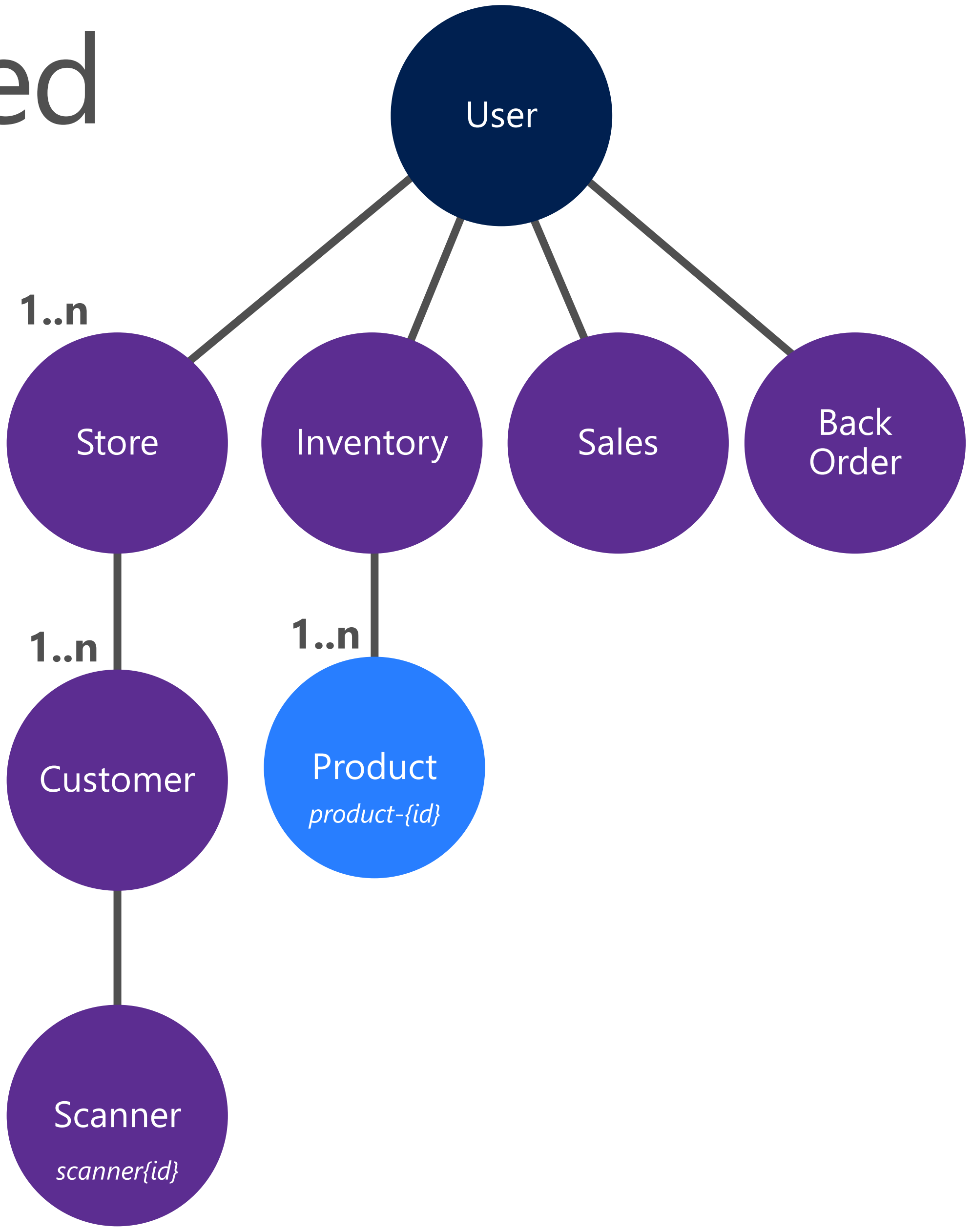
Accumulates sales numbers

Back Order

Back-orders products that are out of stock

Inventory

Handles product purchases
Having a Product actor per product <u>handles concurrency</u>



Demo

https://github.com/EdwinVW/akka.net-warehouse-sample

Wrap Up

Actor Model is great for building HA/scalable systems

Specialized in handling large concurrent workloads

Don’t use Actor model for every LOB application

Think outside the box when designing your system

Partitioning your problem differently can make it easier to solve

Start “on a whiteboard”

Draw and reason about a hierarchy before you start coding

Resources

Akka.NET website and documentation

http://getakka.net

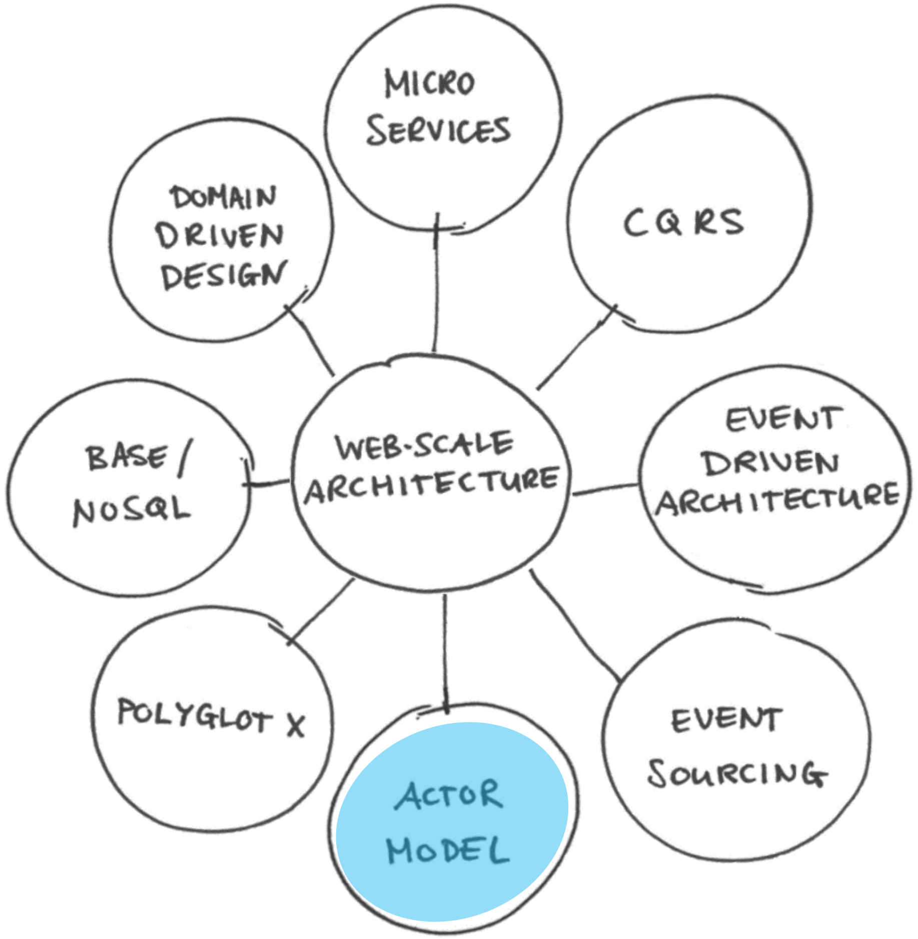
Akka.NET bootcamp

https://github.com/petabridge/akka-bootcamp

Demo code

https://github.com/EdwinVW/akka.net-warehouse-sample

Web-scale Architecture



http://wsa.infosupport.com

Thank you!

edwinw@infosupport.com

@evanwijk

blog defaultconstructor.com

Info Support
Solid Innovator