

Akka Microservices Architecture And Design

Yaroslav Tkachenko

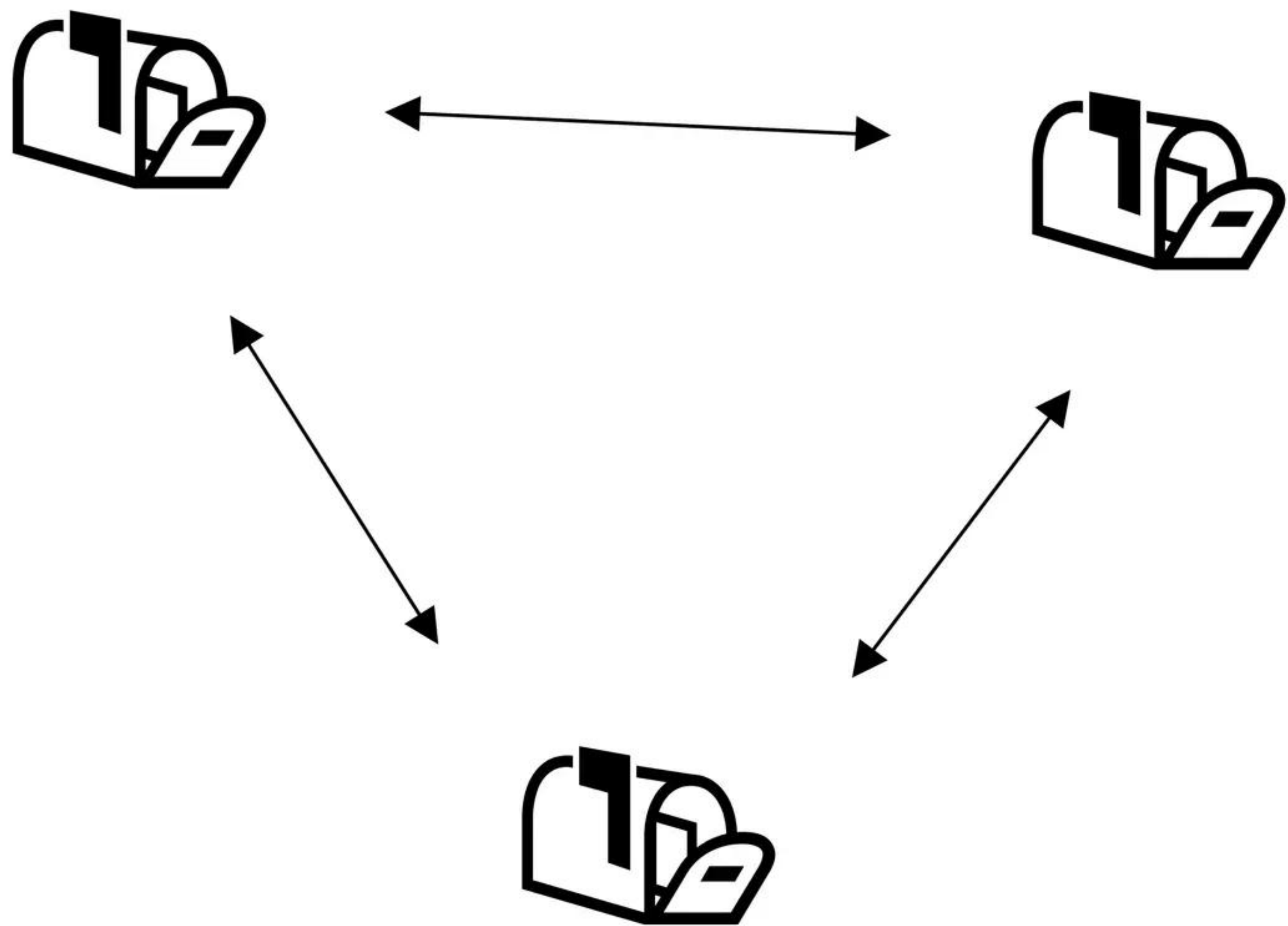
Senior Software Engineer at Demonware (Activision)

@sap1ens



Actors

- Communicate with asynchronous messages instead of method invocations
- Manage their own state
- When responding to a message, can:
 - Create other (child) actors
 - Send messages to other actors
 - Stop (child) actors or themselves



```
object MyActor {  
  case class Greeting(from: String)  
  case object Goodbye  
}  
  
class MyActor extends Actor with ActorLogging {  
  import MyActor._  
  def receive = {  
    case Greeting(greeter) => log.info(s"I was greeted by $greeter.")  
    case Goodbye           => log.info("Someone said goodbye to me.")  
  }  
}  
  
val system = ActorSystem("mySystem")  
val myActor = system.actorOf(Props[MyActor], "myactor")  
  
myActor ! Greeting("another actor")
```

Actor != Class

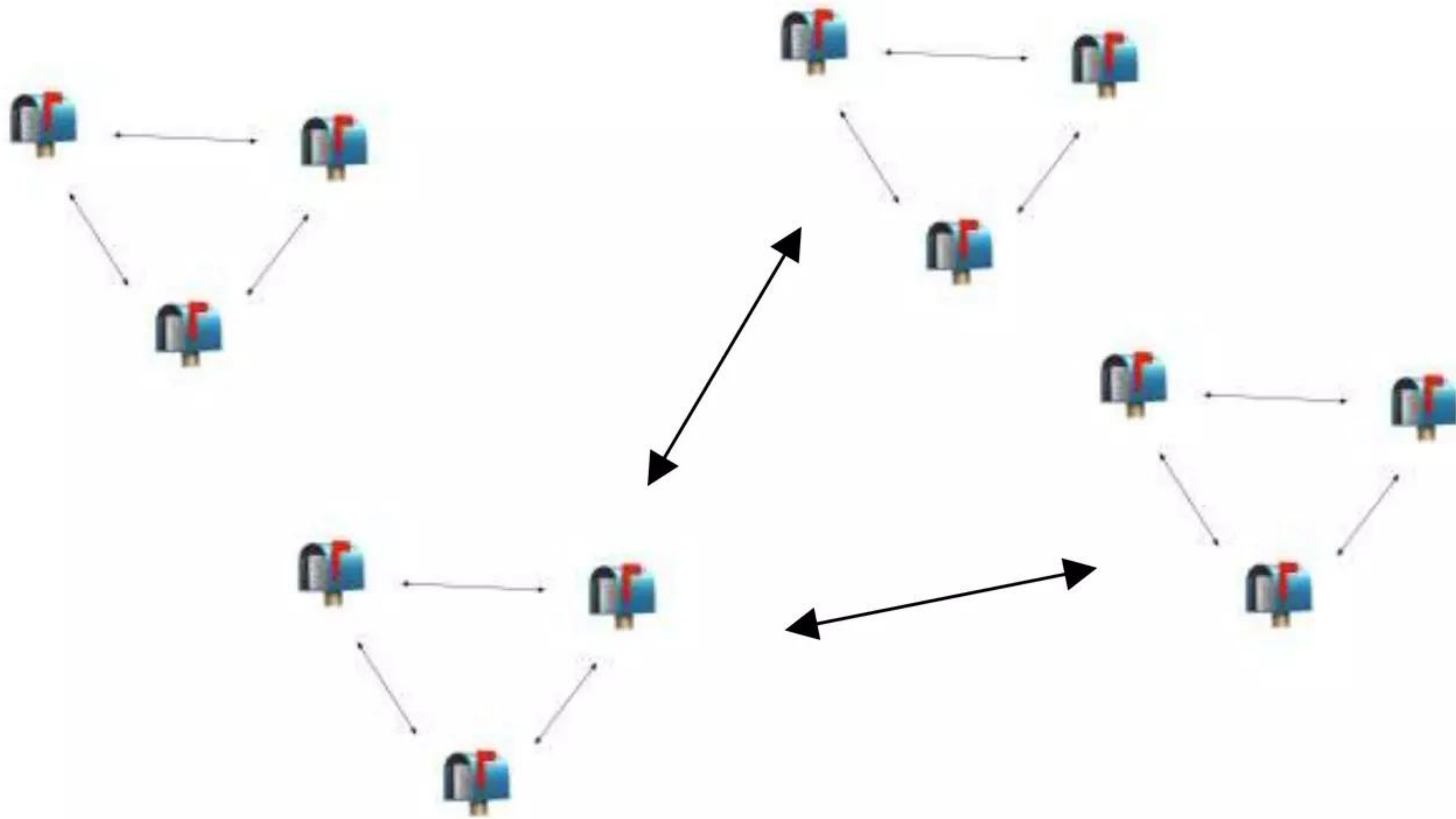
Erlang

```
loop() ->  
  receive  
    {From, Msg} ->  
      io:format("received ~p~n", [Msg]),  
  
      From ! "got it";  
  end.
```

Akka Typed

```
object HelloWorld {  
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])  
  final case class Greeted(whom: String)  
  
  val greeter = Actor.immutable[Greet] { (_, msg) =>  
    println(s"Hello ${msg.whom}!")  
    msg.replyTo ! Greeted(msg.whom)  
    Actor.same  
  }  
}
```

Actor Systems



??? What the ...

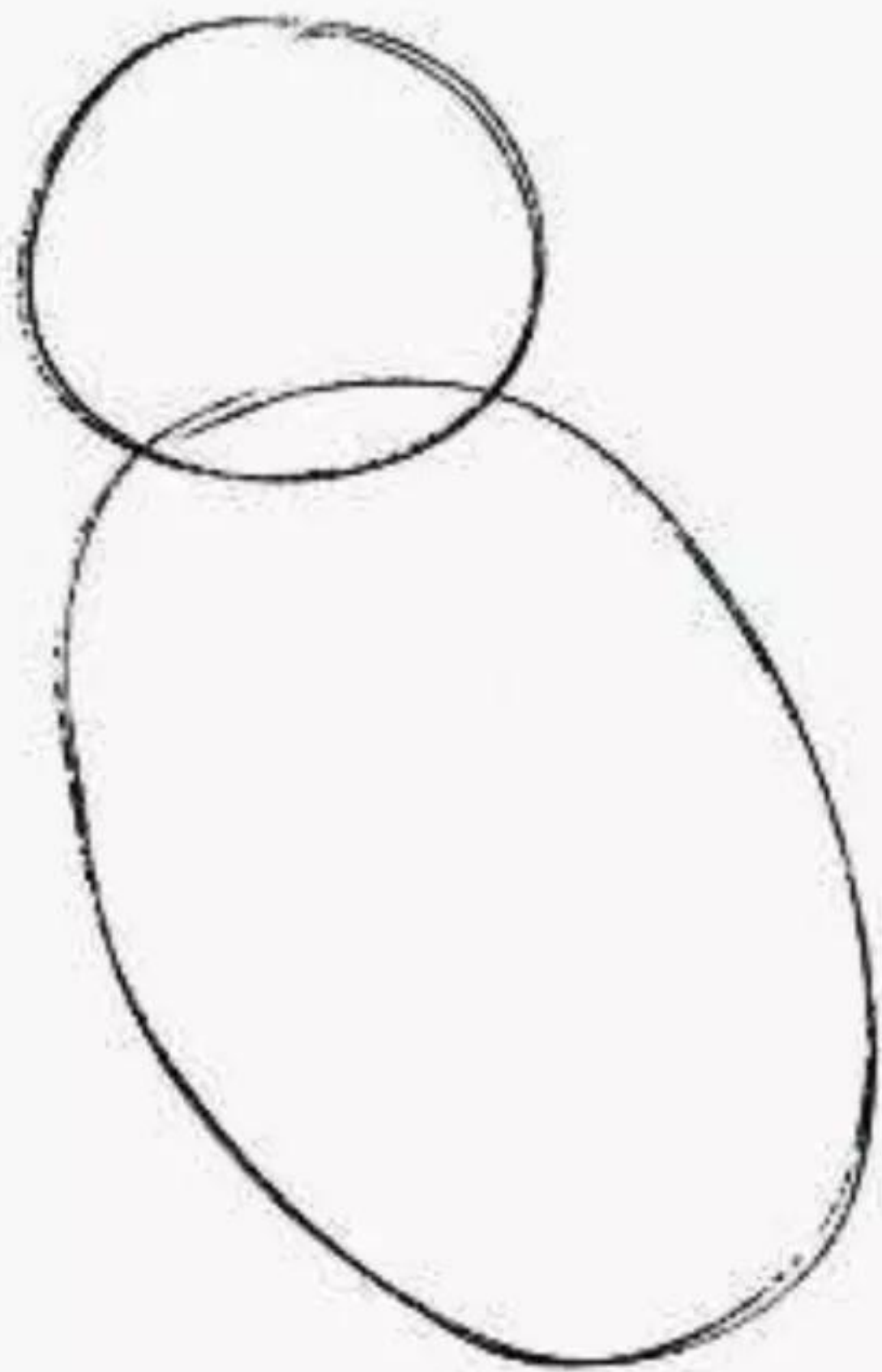


Fig 1. Draw two circles



Fig 2. Draw the rest of the damn Owl

- I know MVC!
- How do I do DI?
- What about that pattern X?
- Does it follow SOA?
- Microservices?

Domain-Driven Design

- Focus on core domain and business logic
- Base complex designs on the domain model
- Collaborate with domain experts (ubiquitous language)
- Strategic and tactical patterns

Tactical Patterns

Entity (Case Class)

- Account
- User
- Transaction

Value object (Trait)

- Checking | Savings | Credit Card
- Active | Inactive
- Debit | Credit

Entities contain business logic *only* involving their own **state**.
We can also call this logic **pure**.

- Account.addBalance
- User.isDisabled
- Transaction.hasLedger

Event (Actor Message - Case Class) is simply a fact that something happened.

- AccountDeactivated(account)
- UserCreated(user)
- TransactionUpdated(transaction, transaction)

Command (Actor Message - Case Class) is a direct instruction for data manipulation, retrieval or a side-effect.

- DeactivateAccount(account)
- CreateUser(user)
- UpdateTransaction(transaction)

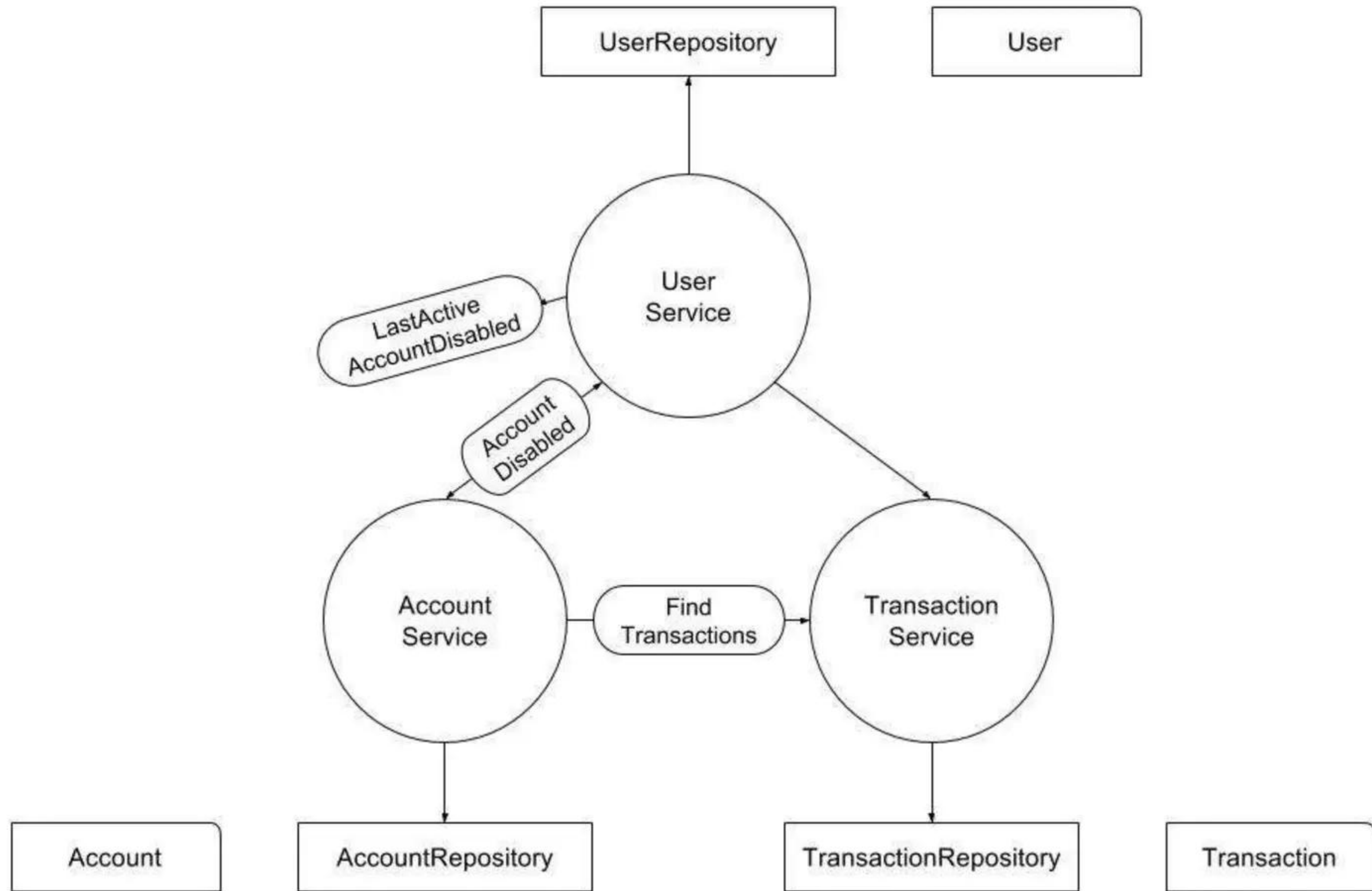
Repository (Actor OR Object/Class) is a way to access or modify domain data and produce a set of entities.

- accountRepository ! FindAccountById(id)
- userRepository ! UpdateUserName(id, name)
- transactionRepository !
FindAllTransactionsBetween(date, date)

Domain Services (Actors) manage multiple entities and/or communicate with other services and repositories to implement complex logic.
They're usually **stateless**.

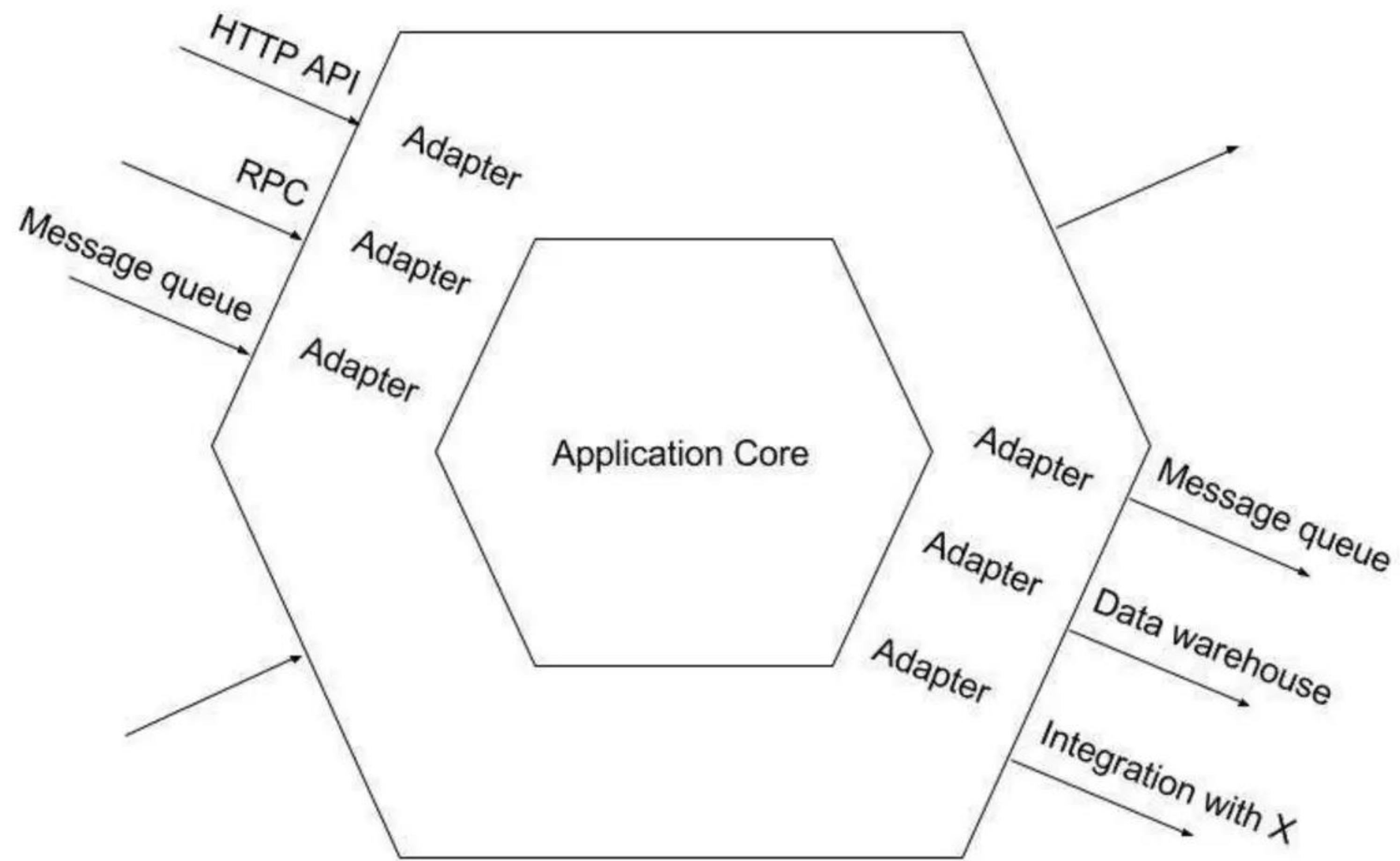
- accountService ! DeactivateAccount(account)
- userService ! RegisterUser(name, email, type)
- transactionService ! MergeTransactions(transaction, transaction)

How about Aggregate Roots?



We defined an Application Core, containing our Domain model and business logic. We haven't yet discussed:

- APIs
- Integrations
- Security
- etc.

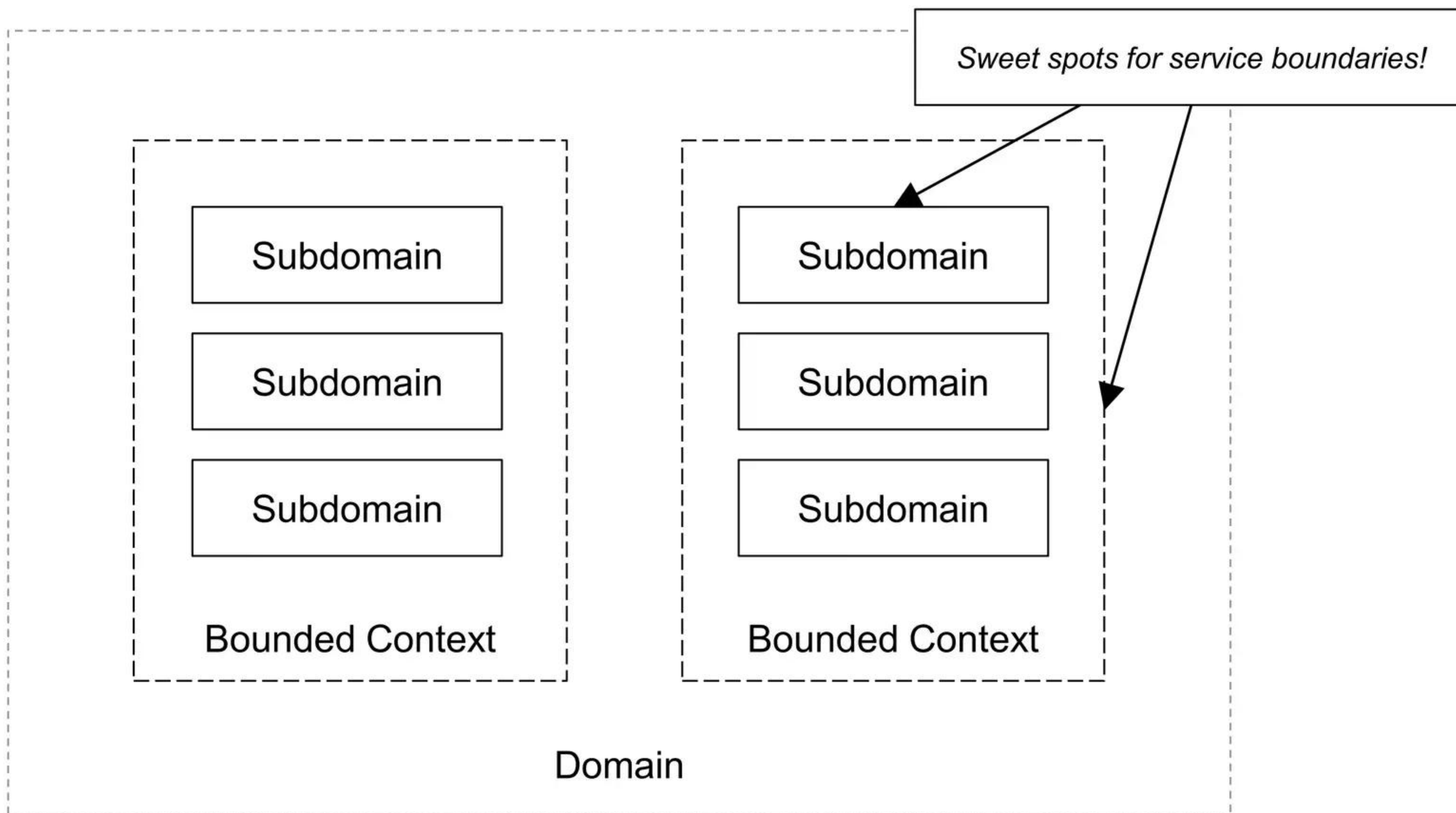


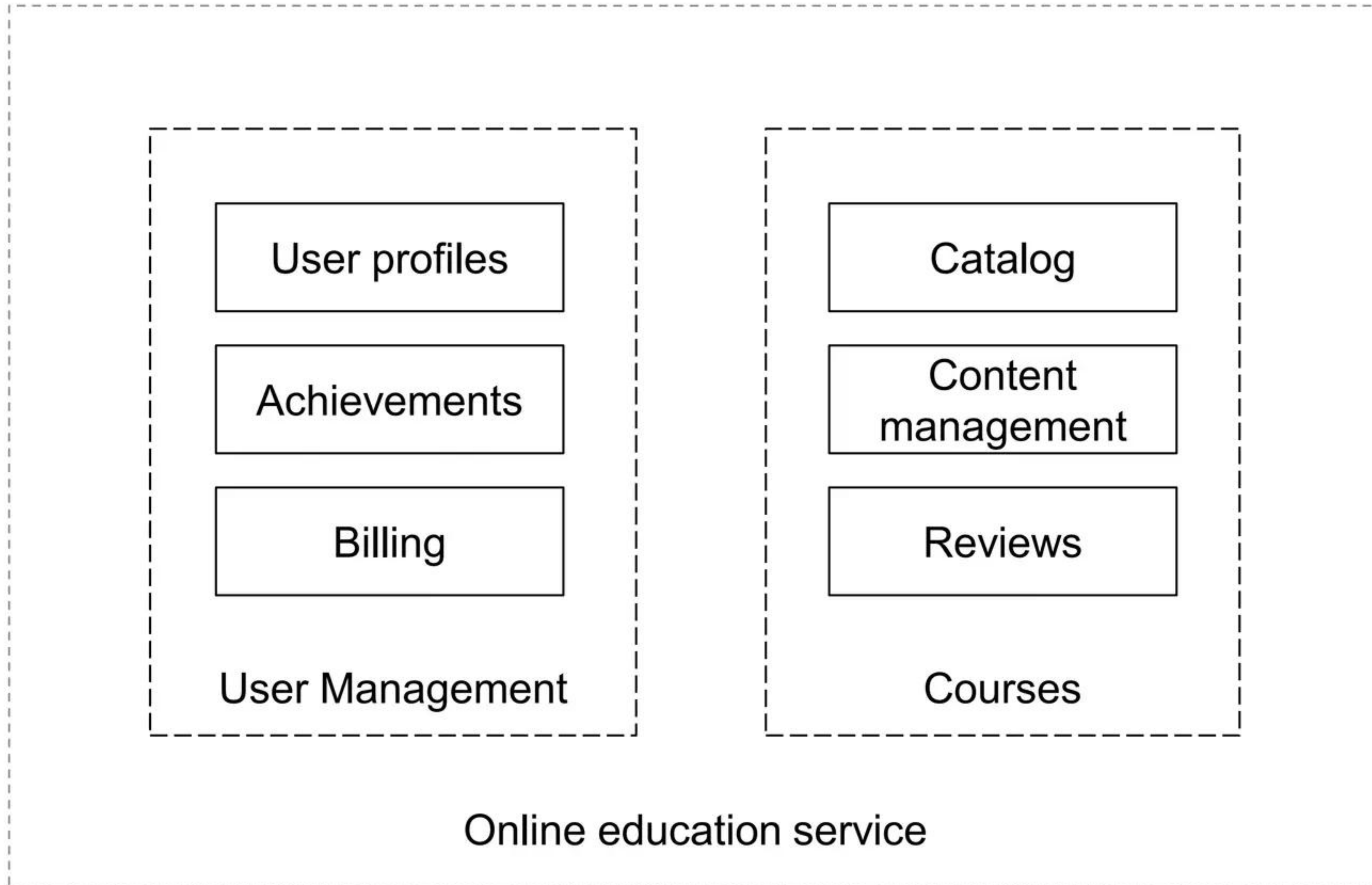
Application Services implement Adapter pattern and they're explicitly located outside of the Application Core. Usually represented as Actors. They can be responsible for:

- Communication via HTTP, RPC APIs, messaging queues
- Integrations with 3rd-party systems
- Security, caching, etc.

Strategical Patterns

How (*assuming we should*) do we split our Domain Model into multiple [micro]services?





So, our [micro]services are:

- Completely isolated
- Deployed separately
- Have different actor systems
- Have clear boundaries from DDD perspective

Now we have multiple [micro]services running multiple actor systems. How do they communicate?

- Synchronously, over HTTP/RPC
- Asynchronously, over messaging

Bla bla microservices bla bla

It is unfortunate that synchronous HTTP is widely considered as the go-to Microservice communication protocol. **Its synchronous nature introduces strong coupling between services** which makes it a very bad default protocol for inter-service communication.

Instead, **communication between Microservices needs to be based on Asynchronous Message-Passing**. Having an asynchronous boundary between services is necessary in order to decouple them, and their communication flow:

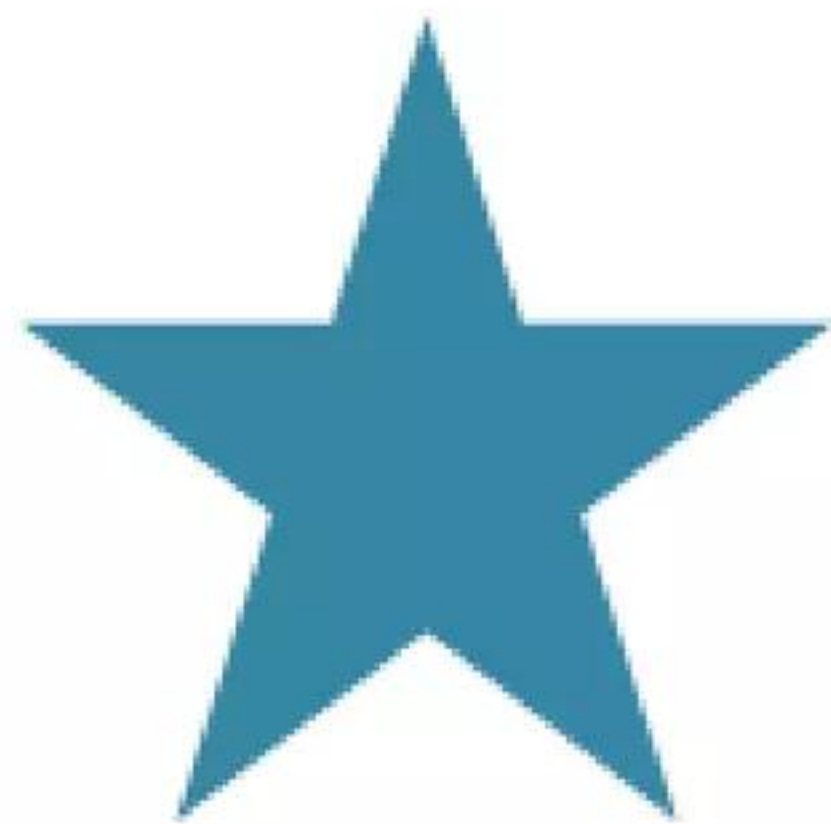
- in time: for concurrency, and
- in space: for distribution and mobility



Jonas Bonér

CTO of Lightbend

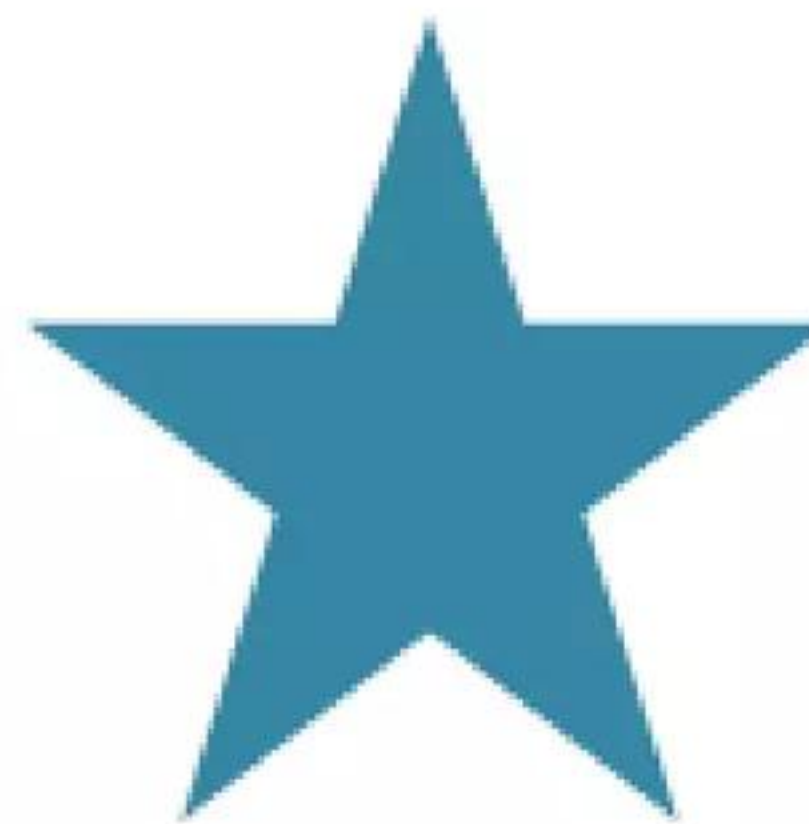
Messaging with actors



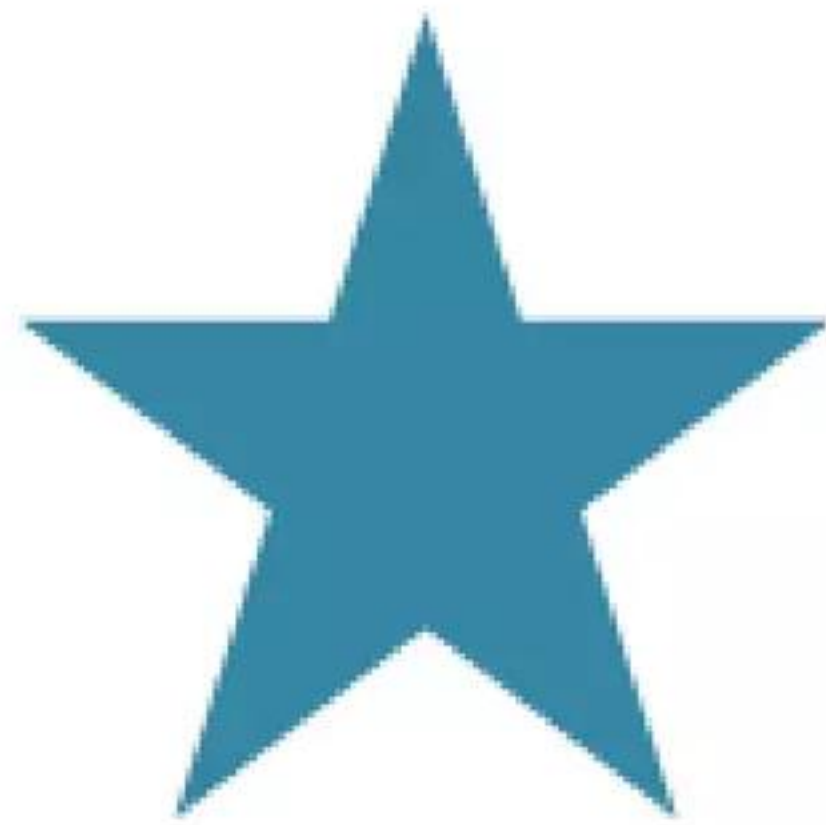
Actor



Message



Actor



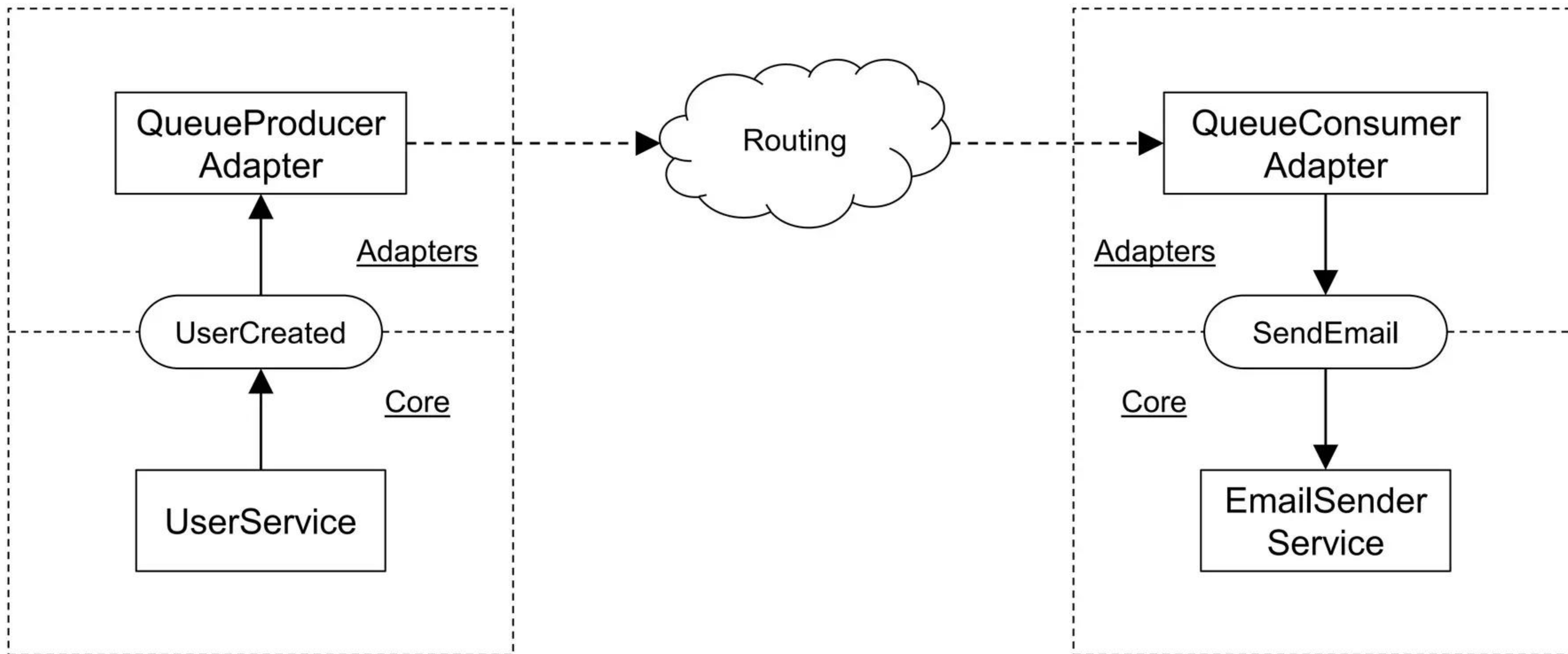
Actor



Message queue

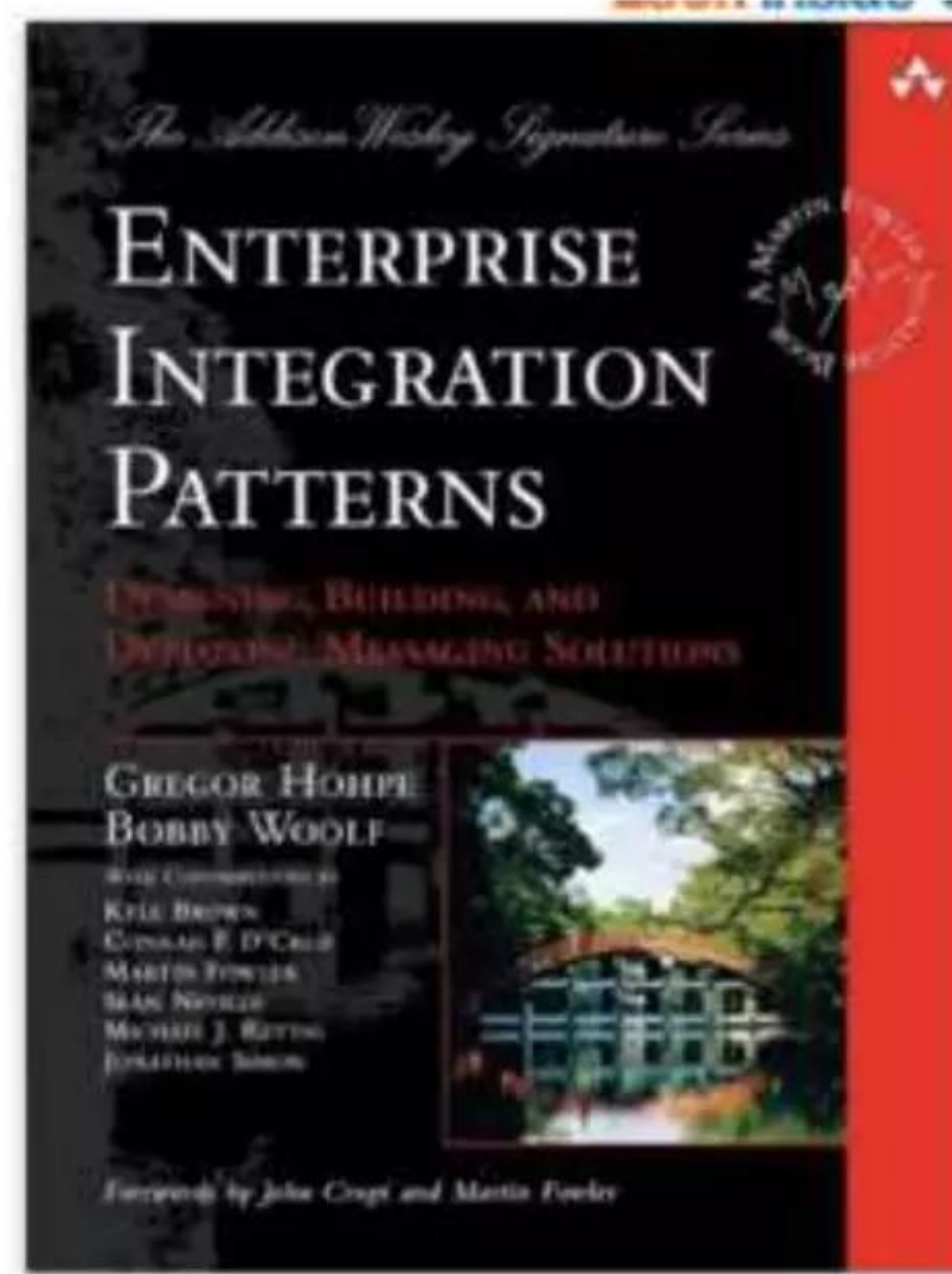


Actor

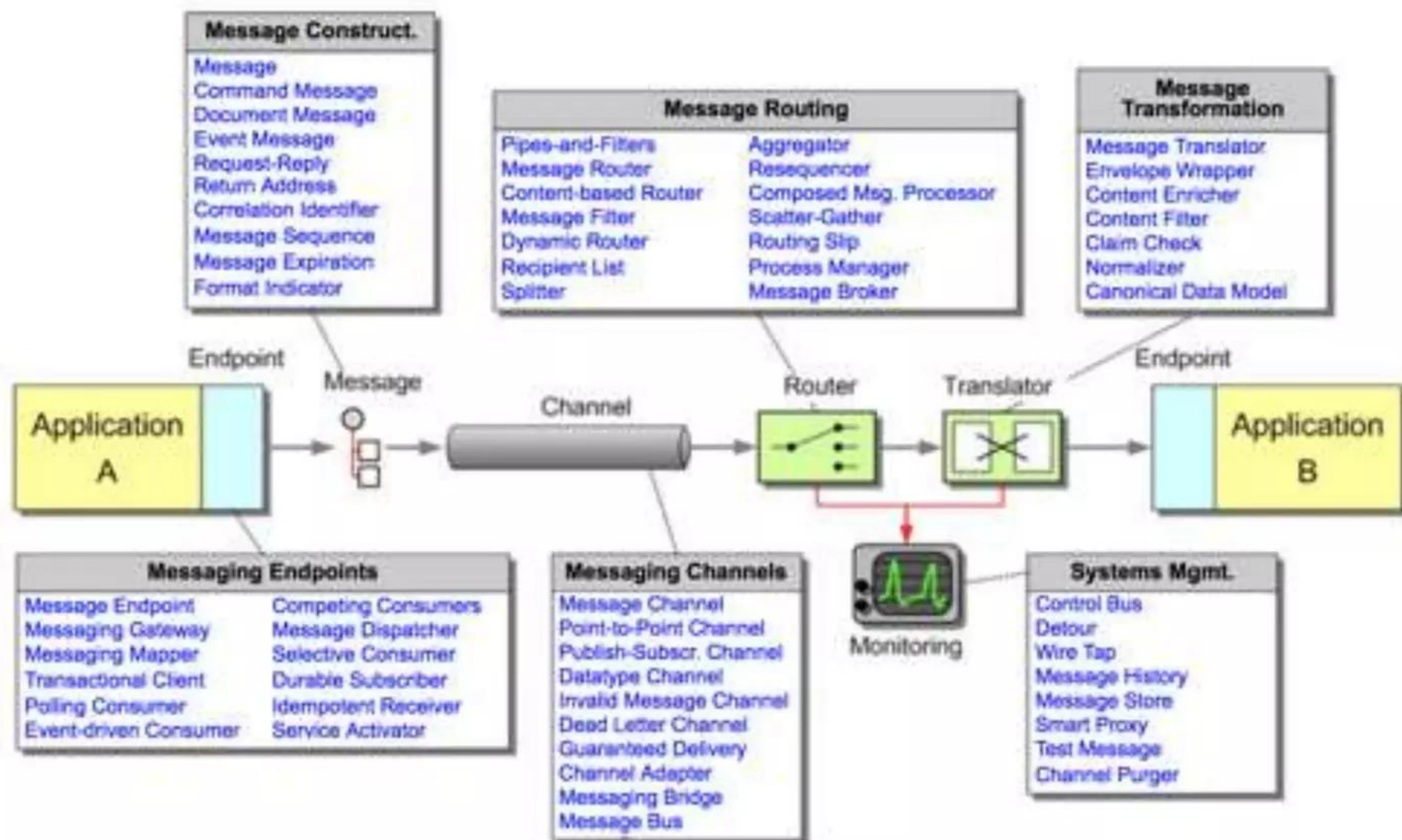


Message routing is very specific to your domain:

- Use ActiveMQ/RabbitMQ by default and see if you can survive with static topics/queues
- Add custom middleware layer for routing (consumer and producer at the same time)
- Use Kafka with Kafka Streams / Akka Streams



EIP was published in **2003** and it contains **65 patterns**.



Apache Camel:

- Integration framework based on EIP
- akka-camel is an official Akka library
- Can be used with any JVM language
- “The most unknown coolest library out there” (©) JM

```
class CustomerService extends Actor with ActorLogging with Consumer {  
  val camel = CamelExtension(system)  
  camel.context.addComponent("activemq", ActiveMQComponent.activeMQComponent(  
    "tcp://localhost:61616"  
  ))  
  
  def endpointUri = "activemq:topic:events"  
  
  def receive = {  
    case e: CamelMessage => {  
      sender() ! "SomeMessage"  
    }  
  }  
}
```

~~Apache Camel~~

Alpakka - *Enterprise Integration Done Right* ©

- Built in top of Akka Streams
- Back-pressure all the way
- Still in active development

Stateful services

Stateful *application* service keeps all data inside the application instead of an external storage (like database or cache)

Why discuss stateful services?

Example, service requirements:

- Need to be really fast (low latency)
- Should have “strong” consistency (at least within one entity), no stale data
- Should be highly available (HA)
- Lots of data
- [Maybe] very complex queries

Which means:

- Might be very challenging to use a database
- Can't use caching
- Solution: stateful service. Keeping data and application logic together

Naive approach:

- Just keeping data in memory. Ok...
- Need to make service HA, running at least 2 nodes. Ok...
- Now we have duplicated data. How do we route requests? Sticky LB? Some partitioning?
- Or what if our data volume is too big for one node?
- We need to shard it. How?
- Akka Cluster Sharding + Akka Persistence FTW!

Akka Clustering is a cluster membership service:

- Gossip-based
- No SPOF
- Eventually consistent
- Failure detection built-in

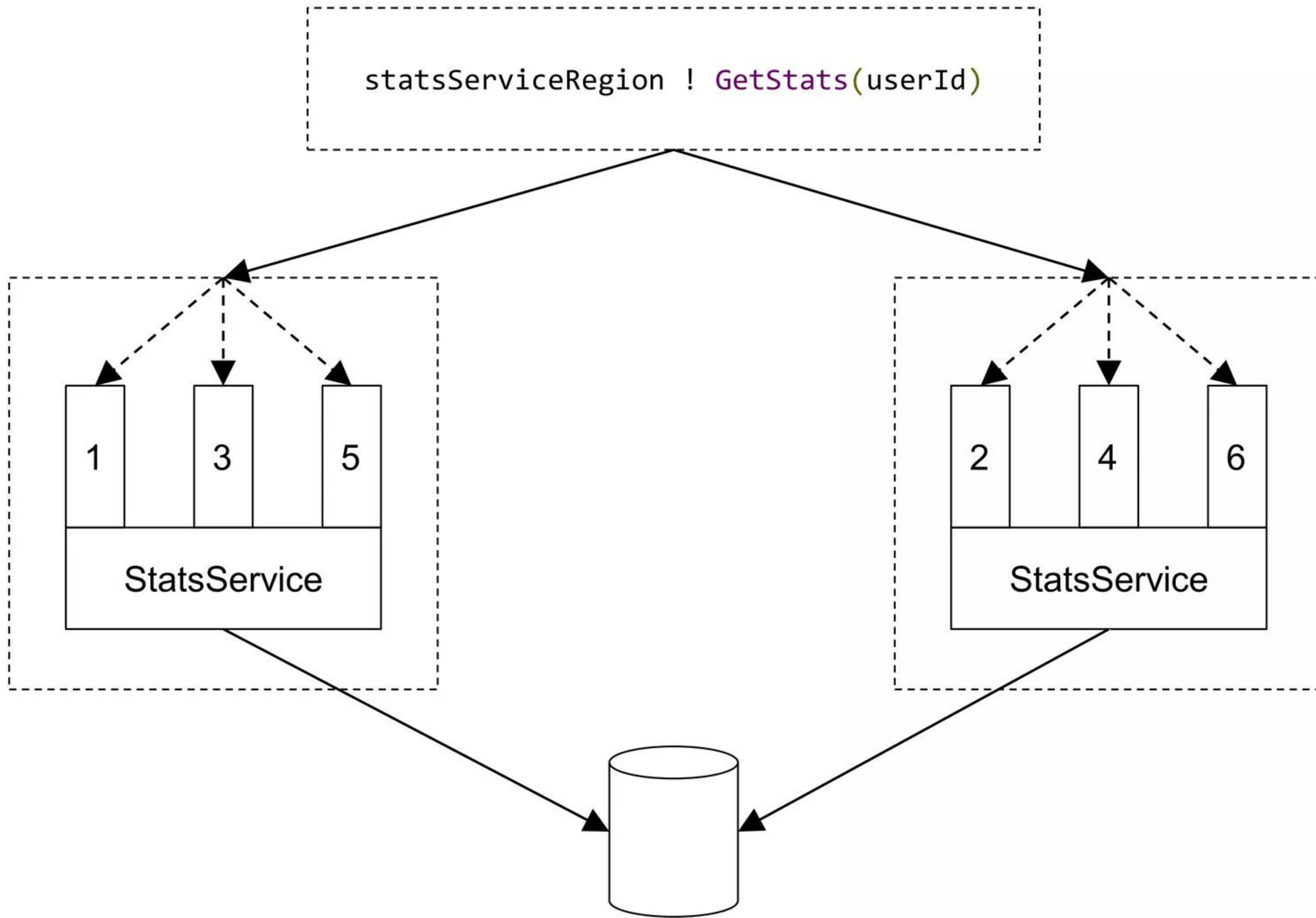
Powerful collection of patterns for building distributed applications:

- Cluster Routing
- Cluster Pub/Sub
- Cluster Singleton
- Cluster Sharding

Akka Persistence allows actors to persist internal state to an external storage and recover after restarts and crashes:

- Event sourcing applied to an internal actor state
- RDBMS, NoSQL, caches and and queues as storage
- Snapshotting for fast recovery

Akka Cluster Sharding + Akka Persistence = ❤️



Stateful services with Akka:

- Routing, sharding and recovery built-in
- You still just deal with actors!
- Akka Distributed Data can be used instead of external storage

It's important to remember that Akka Clustering application is still restricted by the *bounded context (or subdomain)* and it should be used within the *microservice* boundary.

Summary

- Akka is great!
- Use Domain-Driven Design for modelling your application core and understanding service boundaries
- Use Enterprise Integration Patterns for developing your asynchronous messaging design
- Look at Akka Clustering and Persistence for complex problems with distributed services

Questions?

@sap1ens
sap1ens.com