Pragmatic Event-Driven Microservices

Allard Buijze – @allardbz – allard@axoniq.io
Layered architecture

- **User Interface**
  - Web Cache
  - Session replication

- **Service Layer**
  - Method invocation Cache
  - Worker pools

- **Data Access Layer**
  - Distributed 2nd level cache
  - Query Cache

- **Domain Model**
Command Query Responsibility Segregation

Events

Command model

Client

Projections
Monoliths

St Breock Downs Monolith - www.cornwalls.co.uk
Microservices vs Monoliths

Monoliths
Almost all the successful microservice stories have started with a monolith that got too big and was broken up.

Microservices system
Almost all the cases where I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.

Martin Fowler

Source: http://martinfowler.com/bliki/MonolithFirst.html
Are you tall enough?

You must be this tall to use microservices

Source: martinfowler.com/bliki/MicroservicePrerequisites.html
Noun Driven Design

Noun? → Service!
Noun Driven Design

OrderService
Noun Driven Design

CustomerService
Noun Driven Design

ProductService
Noun Driven Design

InventoryService
Noun Driven Design
Location transparency

A component should neither be aware of nor make any assumptions about the location of components it interacts with.

Location transparency starts with good API design

*(but doesn’t end there)*
Reasons to send a message

• Event
  Something has happened
‘Event-Driven’ Microservices

Order service

Need to know ordered items

- OrderCreated
- ItemAdded
- ItemRemoved
- OrderConfirmed
Reasons to send a message

• Event       Something has happened
• Command     I want the system to do something
• Query       I want to know something
Something has happened - Event

• Data change
• Deadline passing
• Or anything else that’s relevant in the domain
Publish-subscribe

Something happened!

Want to know

Want to know

Want to know
Exclusive consumers

Something happened!

One of us wants to know

Want to know

One of us wants to know (crossed out)
Competing consumers

Something happened!

One of us wants to know

One of us wants to know

Want to know
One of us wants to know

Something happened!

One of us wants to know

Want to know

Balanced consumers
I want something done - Command

• Request-a-side-effect
  • Change data / application state
  • Send email
• Exactly 1 destination
• OK / NOK reply
  • Maybe some data
Command Routing

Do something!

Can do something

Can do something

Can do something
Command Routing

Do something! if (%) Can do something
Can do something
Can do something
I want to know something -
Query

• Desire for information
  • The response has more value than the question
  • (Usually) side-effect free

• Different messaging patterns
  • Single destination
  • Scatter – gather query
  • Subscription
Query – point to point

Result = Price

Price = 49
Query – scatter-gather

Result = Min(Price) (wait 100ms)

Price = 199

Price = 149
Query – scatter-gather

Result = Min(Price) (wait 100ms)

If (…)
Price = 199

If (…)
Price = 99

If (…)
Price = 149
Query – subscription

Result = Price + \sum(\Delta)

\Delta = -10

Price = 39
‘Event-Driven’ Microservices

OrderCreated →
ItemAdded →
← GetOrderDetails
OrderDetails →

Order service

Need to know ordered items

AxonIQ
Events retain value

Event Sourcing is an Architectural pattern in which Events are considered the “source of truth”, based on which components (re)build their internal state.
Event Store

An Event Store stores the published events to be retrieved both by consumers as well as the publishing component itself.
Event Sourcing

Order service ➔ OrderCreated ➔ OrderConfirmed ➔ ItemRemoved ➔ OrderDetails ➔ Some smart analytics
Event Store operations

- Append
- Validate ‘sequence’
Event Store operations

• Full sequential read
Event Store operations

- Read aggregate’s events
1. Define which routing patterns to apply

2. Choose technology/protocol accordingly
At scale, different rules apply
How do you route all these events to all components?

How will this scale?
You Don't!

It Won't!
Unmanageable mess

As shipping module, I want to know when an order is placed

Order Created
Item Added to Order
Shipping Address Added
Billing Address Added
Order Confirmed
Bounded context

Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don’t be distracted or confused by issues outside.
Within a context, share ‘everything’
Between contexts, share ‘consciously’

As shipping module, I want to know when an order is placed

Order Created
+ Item Added
+ Order Confirmed
→ Order Placed

Order Created
Item Added to Order
Shipping Address Added
Billing Address Added
Order Confirmed
Between contexts, share ‘consciously’

As shipping module, I want to know when an order is placed.
Where does AxonFramework fit?

- Inside each component in bounded context
- Axon provides the Java APIs towards platform
  - EventBus, CommandBus, QueryBus
- Separation of business logic and infrastructure logic
Microservices Messaging
“Just enough” intelligence

Message Broker
Sends messages. Main value add is reliability.

AxonHub
Understands difference between Commands, Events, Queries and their routing patterns. Does not care about the content of these messages.

Enterprise Service Bus
Understands message content. Hard to configure and maintain.

dumb

AxonIQ

smart
Our mission

Provide the APIs and implementations necessary for event-driven microservices to cooperate harmoniously, allowing each of them to focus on the business logic.