@evanwijk

</DevSum>
May 19th 2022

Lessons learned implementing a cloud-native architecture in .NET (Core)

Edwin van Wijk
Principal Architect
InfoSupport
Solid Innovator

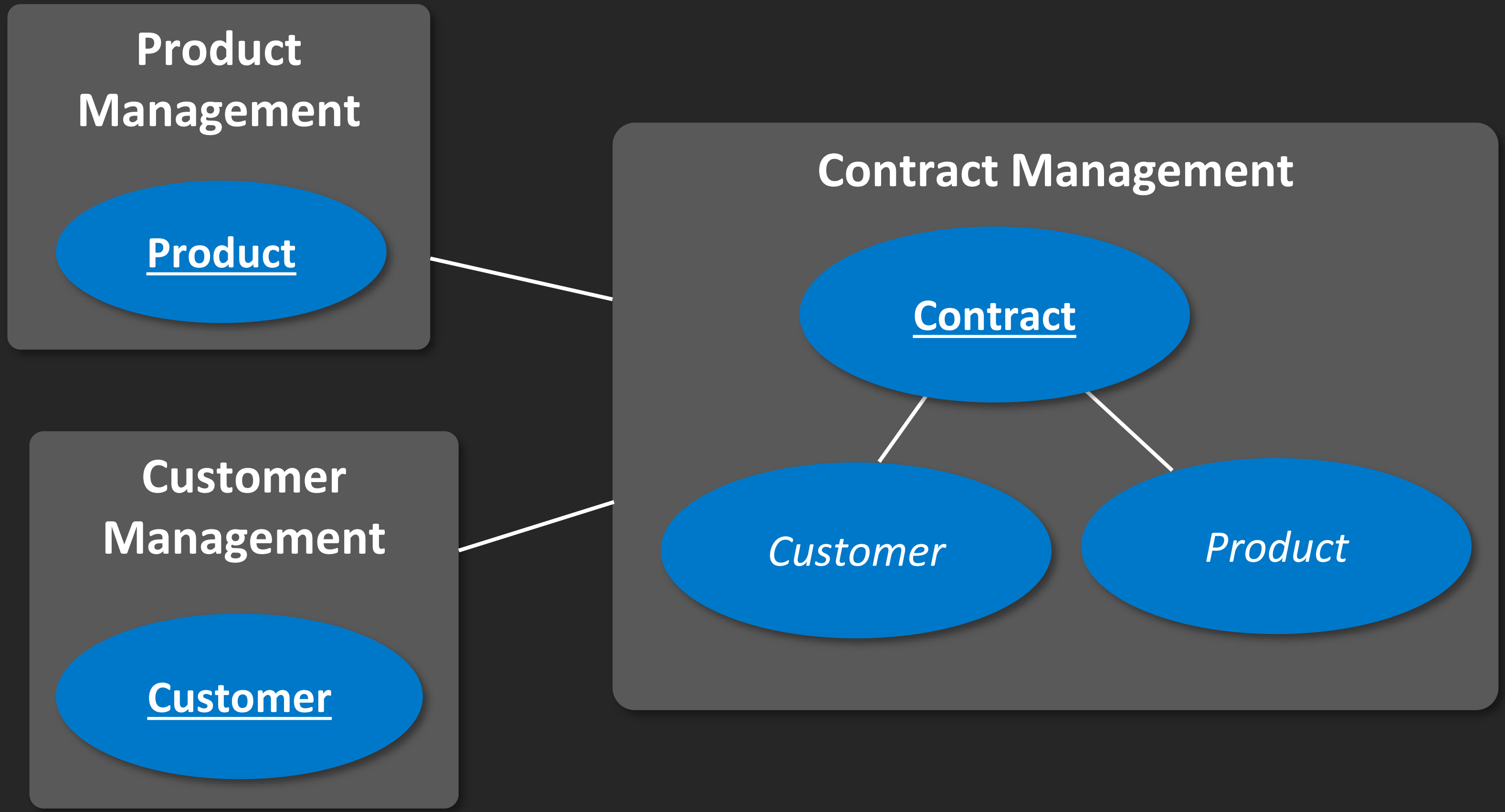
MVP
Microsoft
Most Valuable Professional

Introduction

- This session will feature the **lessons I've learned** building several systems using a **cloud-native architecture in .NET**
  - Focus is on CQRS, Domain Driven Design and Event Sourcing

- Because I'm not able to share any **customer code**, I've created **sample code** to support this presentation
  - Contains example implementations in .NET
  - I'll share the repo so you can dive deeper if you're interested

Domain overview (simplified)



Microservices with CQRS

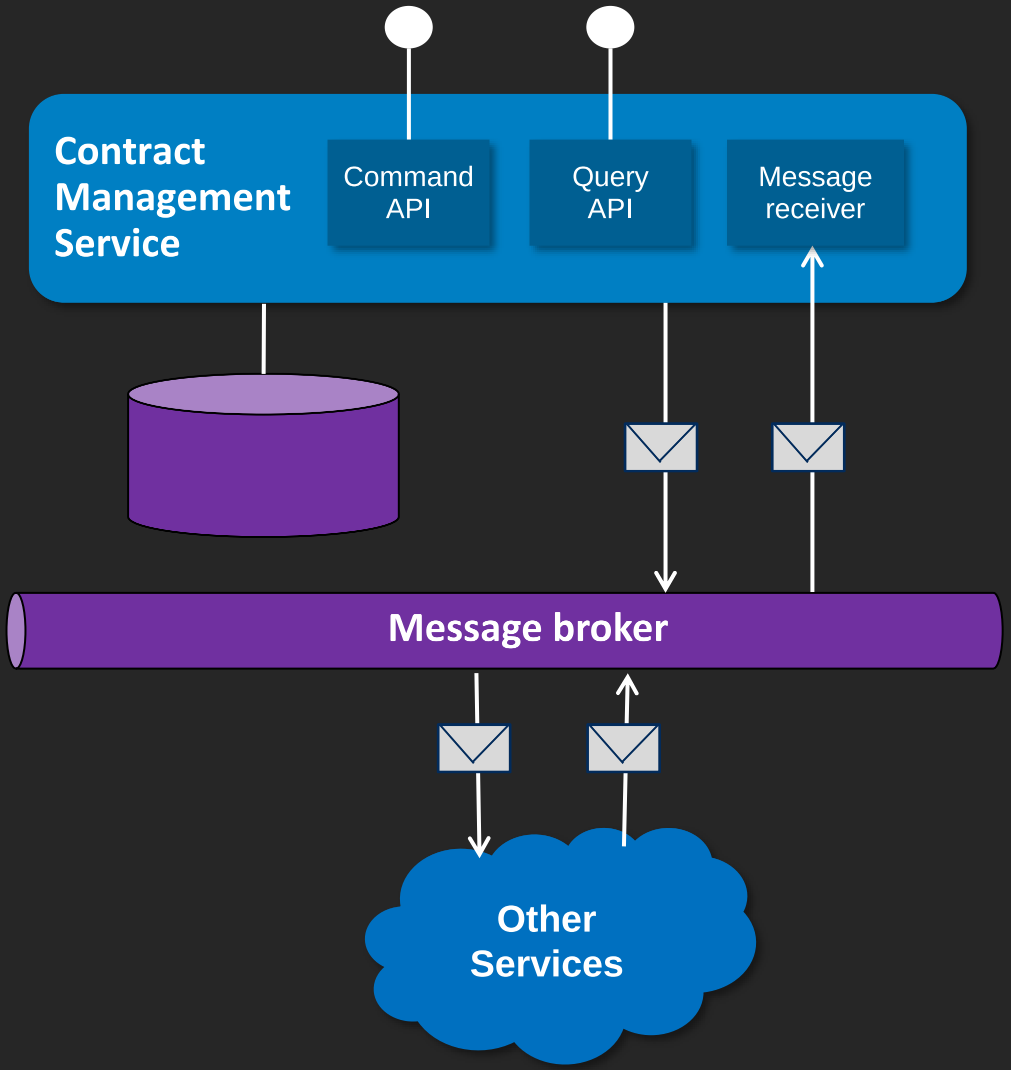
Microservices

- **Microservices** was selected as **top-level architecture** style
- Primary focus on **autonomy**
  - Service can be maintained by an autonomous team
  - Service can execute its (primary) tasks as autonomously as possible
- We've used the **CQRS pattern** for **our more complex services**
  - Great fit with DDD

CQRS - Command Query Responsibility Segregation

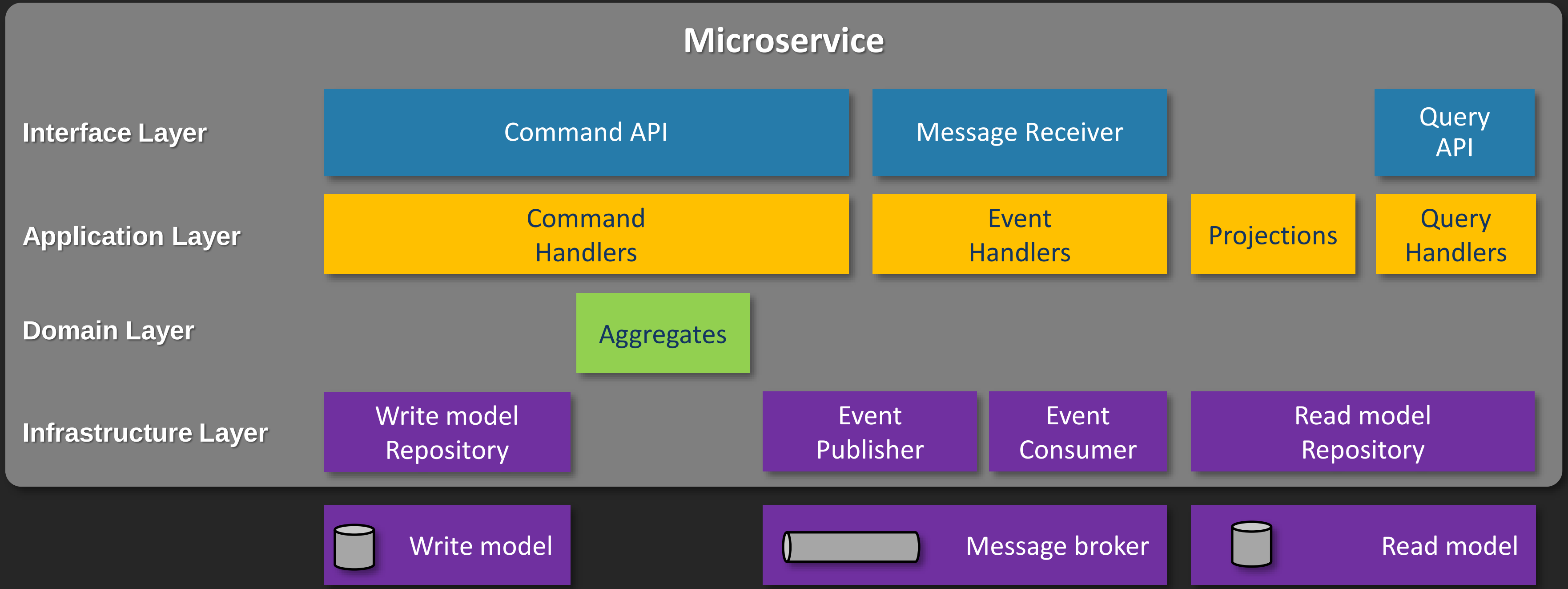
- Separate the **Command** part ("write") from the **Query** part ("read")
  - Only used in more complex services
- Write model contains **the state** of an aggregate
  - Optimized for writing data
  - Only for rehydrating the aggregate state
  - **Never** used for elaborate queries
- Read model(s) contain data for **querying**
  - Optimized for reads (might be denormalized)
  - Multiple read models can support multiple different data consumers
  - Read models can be used to cache data from other domains through events (autonomy)

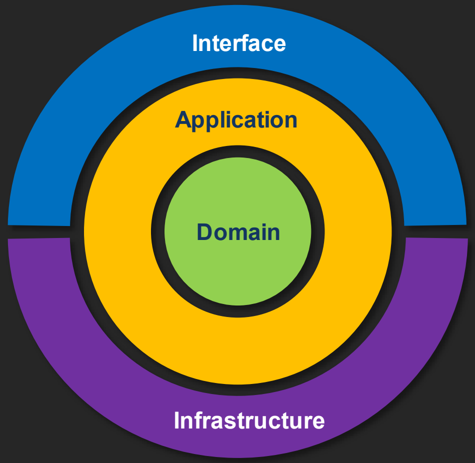
Contract Management Service



Generic Microservice pattern

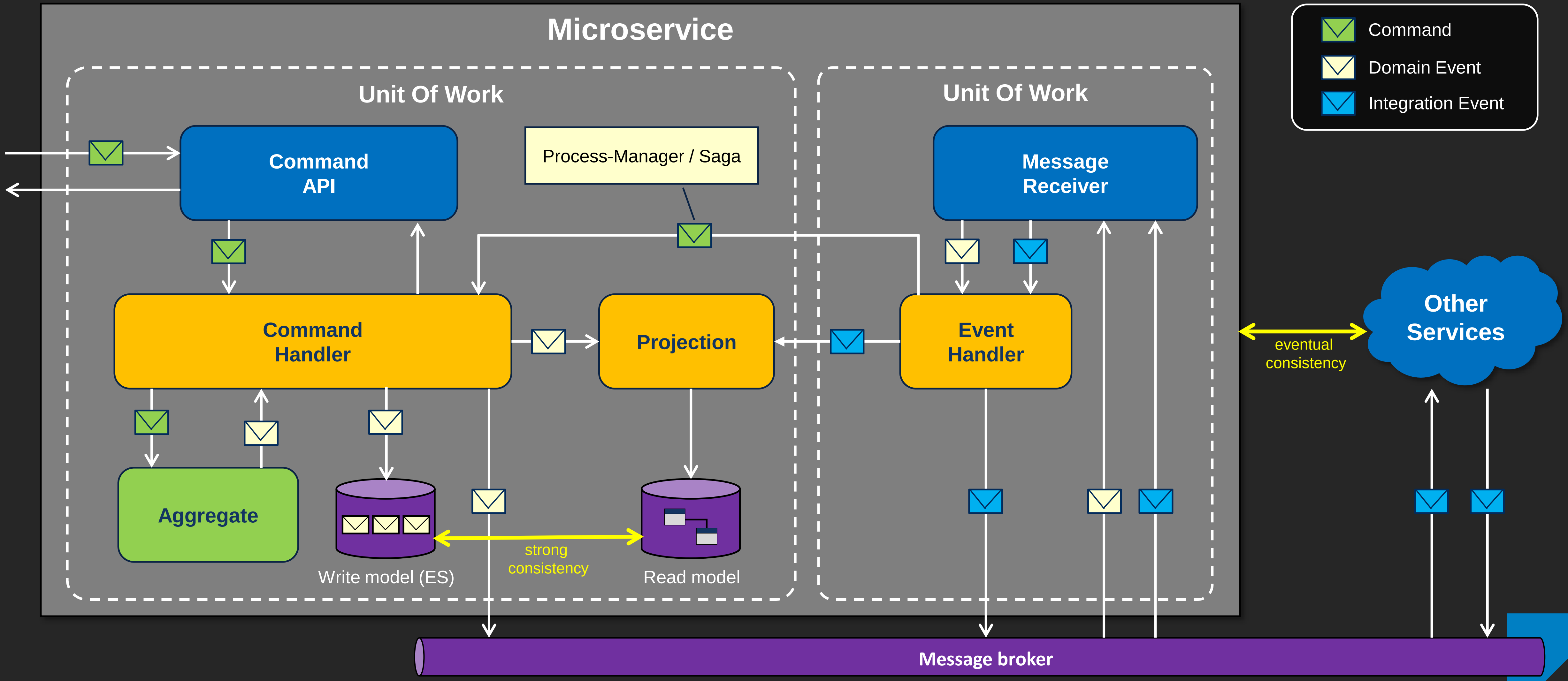
- Generic pattern for implementing CQRS that offers consistency over services
  - The steps to handle a command are always the same, only the business logic differs
  - We created a small set of convenience base-classes with boiler plate code (~200 loc)





Uses onion architecture layering

Handling a command code walkthrough



Why the Unit Of Work?

- Enforce the principle: **one command == one business transaction**

- Transaction is always scoped **within 1 aggregate**

- Results in:
  - Narrower locks
  - Smaller risk of concurrency issues
  - No reentrancy into the domain

- Updating the write- and read-model in 1 transaction eliminates the need for dealing with **eventual consistency**
  - Less complex

Why domain events vs. integration events?

- We want to have a **clear distinction** between the **"inside"** and the **"outside"** of a domain or service (DDD **bounded context** with strict boundaries)
- **Domain events** are for communicating changes **within a bounded context**
  - Event-sourced services use them to store the state as an event-stream
  - Projections use them for updating the read-model(s)
  - Other aggregates in the same bounded context could be triggered by them
- **Integration events** are for communicating changes **outside a bounded context**
  - Other services could be triggered by them to:
    - Execute a command / start a process
    - Update a local cache read-model with the information from the event
  - An integration event can be different in naming and structure from the corresponding domain event

Domain Driven Design

Domain Driven Design

- Only for the **more complex** domains / services
  - ☑ ContractManagement service

- **Focus** should always be on the domain
  - This is basically the only thing the business is interested in
  - Teams create or change the domain first (supported with domain level unit-tests)
  - Much emphasis on domain boundaries, terminology (*ubiquitous language*) and business intent
  - Only after the team is happy with the domain, the application and integration stuff is added

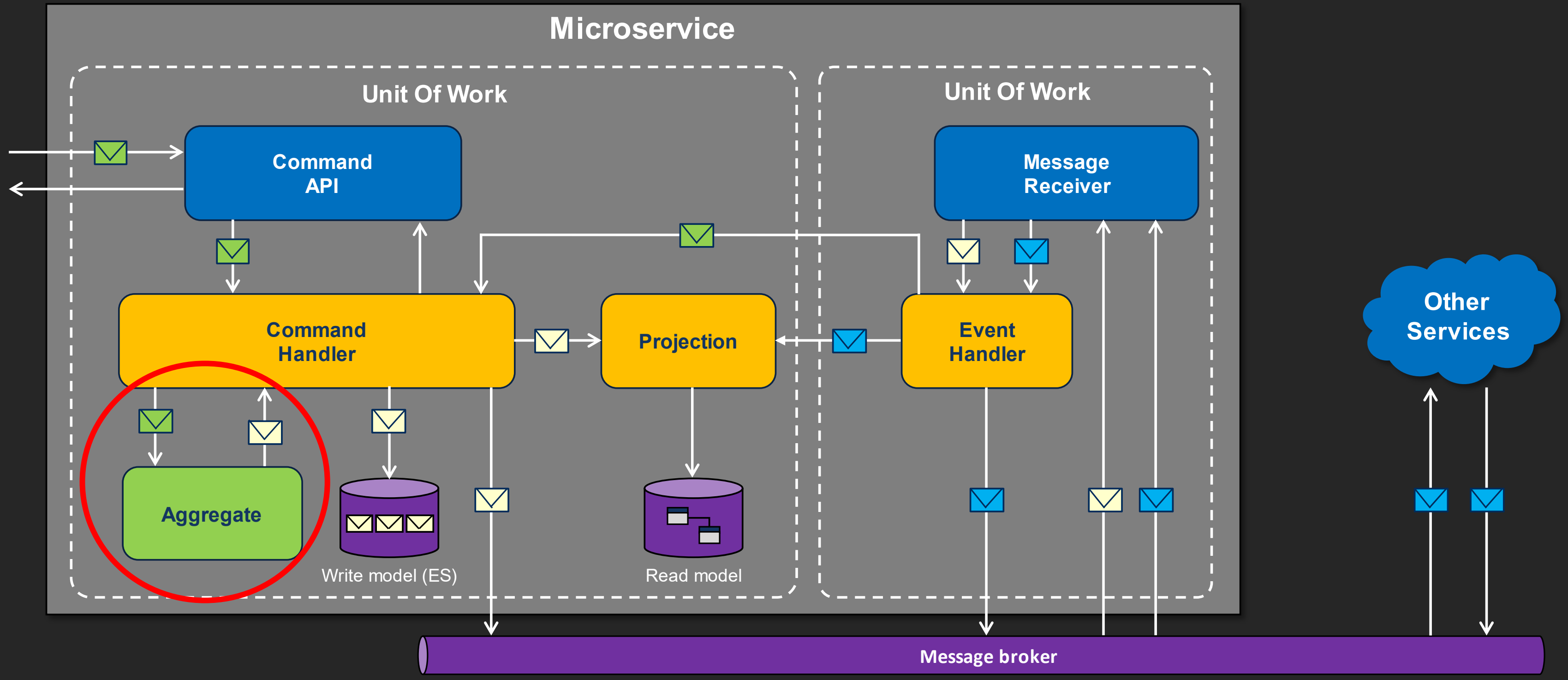
- We implemented the **strategic DDD patterns**
  - Aggregates, Entities , Value-objects, Repositories, Domain-services

Domain Driven Design - Aggregates

- An **aggregate** is a set of entities that belong together
- The **aggregate root** is a special entity that forms the only entry-point into an aggregate
- It offers operations that will **handle commands**
- It makes sure the entire aggregate is in a **consistent state** after making changes
  - Pre-validation: before changing the state of the aggregate
  - Post-validation: after changing the state of the aggregate

Domain Driven Design - Aggregates

- An aggregate is **always event-driven**
  - Always command in, domain event(s) out
  - Not necessarily event-sourced!



Domain Driven Design - Aggregates

- The **command handling** process within an aggregate always consists of the following steps:
  1. Check business rules in the command-handler method (pre-validation)
  2. Create a domain-event and "apply" it to the aggregate
  3. Corresponding event-handling method changes the state of the aggregate (no other side-effects or external calls allowed!)
  4. Check the overall consistency of the aggregate (post-validation)
  5. "Publish" the domain-event to communicate the changes made to the aggregate

Domain Driven Design - ValueObjects

- **ValueObjects** are items in the domain that are not entities (no clear Identifier) and are equal based on their Value

- **.NET Records** are a great fit for implementing them
  - Immutable and automatically equitable based on property values

- We implemented 3 ways of **instantiating** a ValueObject
  - **TryParse** - try to create a ValueObject instance from a scalar value with validation
  - **Parse** - calls TryParse and throws an exception when invalid
  - **Constructor** - use the value passed into the constructor without validation.
    This is used by repository when rehydrating an event-sourced aggregate from storage (enables changing of validation rules over time)

Domain Driven Design - Unit-testing

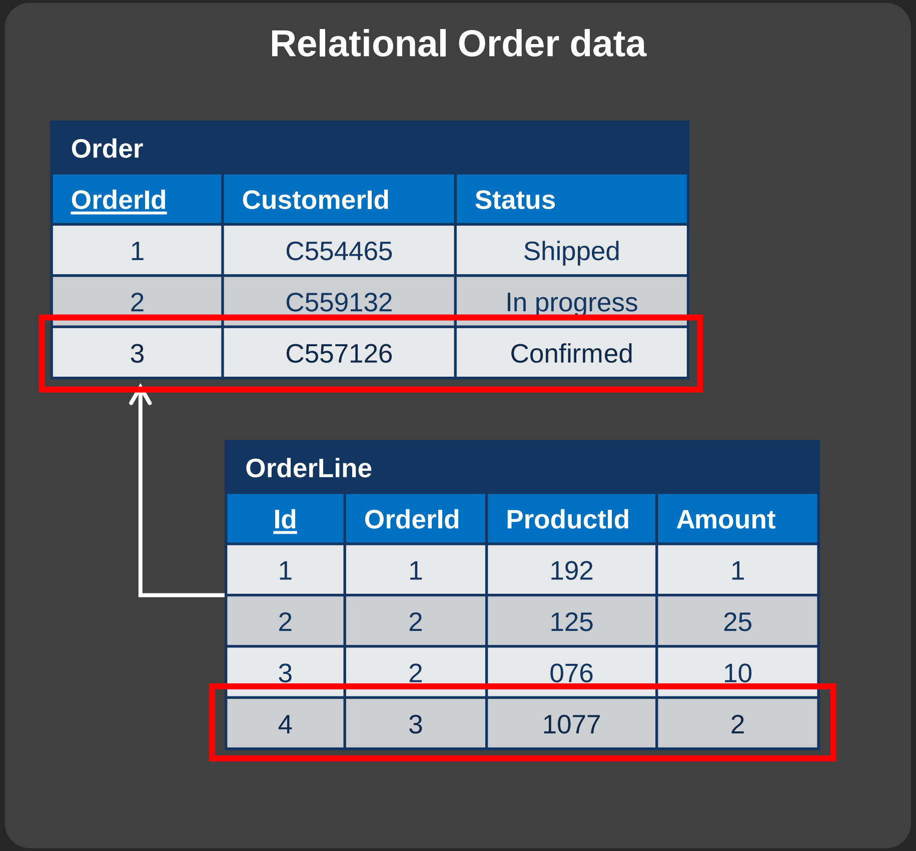
- The team **tests** functionality (and regression) with **unit-tests** on **domain level**

- Unit-test steps:
  - Create necessary state by creating a list of events
  - Rehydrate an aggregate instance by passing the events into the constructor
  - Test functionality by firing a command at the aggregate
  - Assert valid operation by checking:
    - Changed properties (state) of the aggregate
    - "Published" domain event(s)
    - The *IsConsistent* property and the business-rule violations

Event sourcing

Why event sourcing?

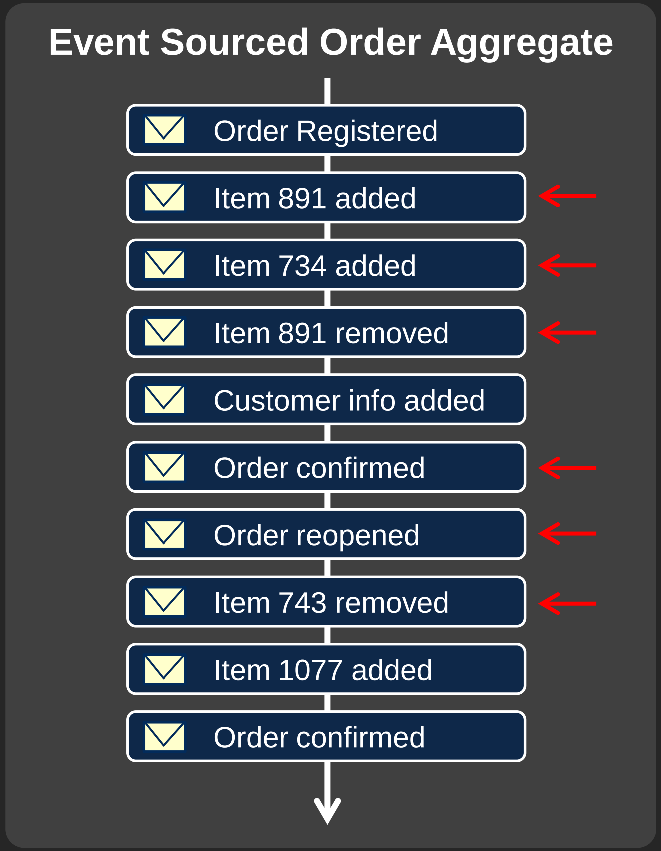
- **Aggregate state** (write model) is stored as a **collection of events**
  – Every domain event that made a state change in chronological order



Relational Order data

Order		
OrderId	CustomerId	Status
1	C554465	Shipped
2	C559132	In progress
3	C557126	Confirmed

OrderLine			
Id	OrderId	ProductId	Amount
1	1	192	1
2	2	125	25
3	2	076	10
4	3	1077	2



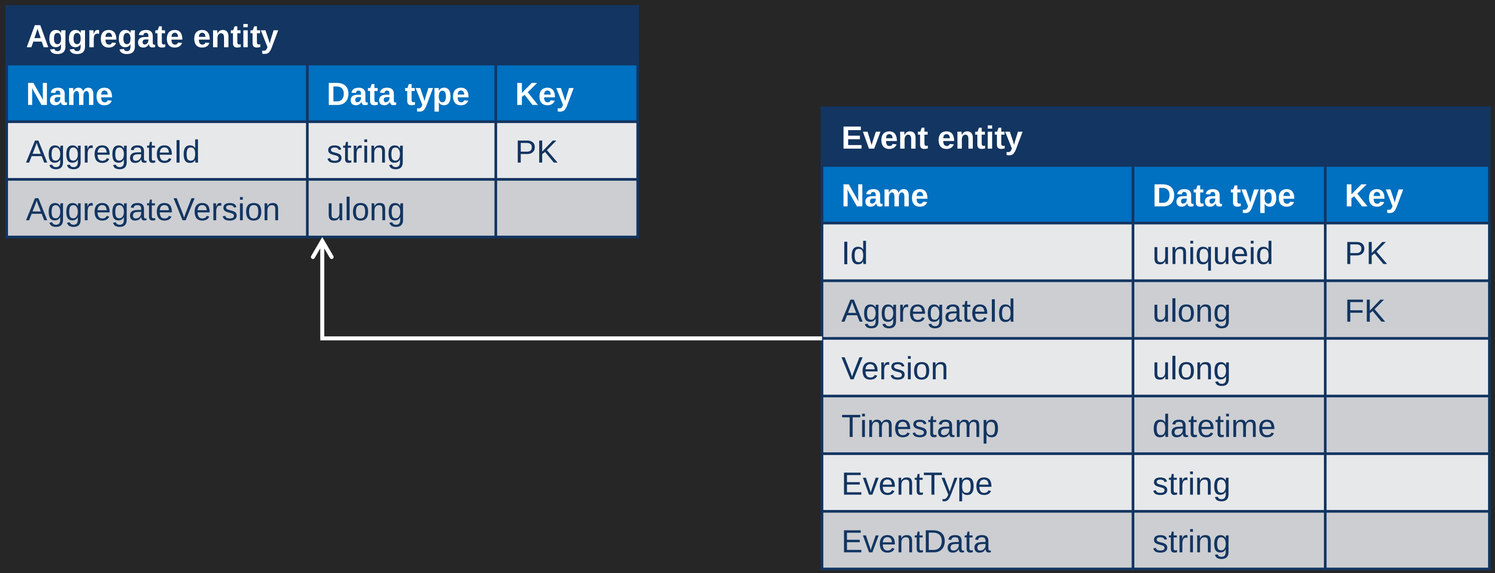
Event Sourced Order Aggregate

Event Sourcing - technical implementation

- We chose a **DIY solution** over an **ES product**
  - ES products solutions are often feature packed and **can** be overkill
  - First run with ES for some time to really know what you need (you could implement a product later)
- **SQL Server** as event store
  - Existing experience in the dev and ops team
  - High availability and disaster recovery already available
- **EF Core** as ORM
  - Existing experience in the dev team
  - Code-first migrations for creating the database

Event Sourcing - SQL Server

- We've created 2 entity classes for storing the events for an aggregate
  - [Aggregate].[AggregateVersion] column is used for optimistic concurrency control
  - [Event].[MessageType] column contains the type of the event
  - [Event].[MessageData] column contains JSON serialized message object
- We use the **same data-model for every aggregate**, so no data migrations!



Aggregate entity

Name	Data type	Key
AggregateId	string	PK
AggregateVersion	ulong	

Event entity

Name	Data type	Key
Id	uniqueid	PK
AggregateId	ulong	FK
Version	ulong	
Timestamp	datetime	
EventType	string	
EventData	string	

Event Sourcing

- I did not encounter the need for implementing **snapshots** (yet)
  - Small number of events per aggregate
  - An aspect we explicitly take into account in the design of our aggregates (prevent it)
  - Replaying events has always been more than fast enough for the perf requirements
- **NEVER** add columns to the write model for **query** purposes!
  - Queries are always executed on the read-model(s) of a service

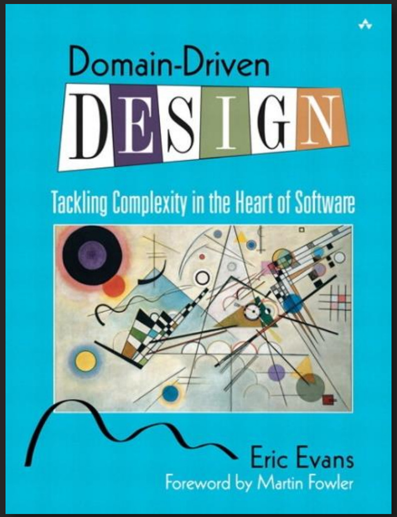
Event Sourcing - Event versioning

- A **new version** of an event is created for **breaking changes**
  - Adding an new mandatory property or removing an existing mandatory property
  - Renaming stuff (you should avoid that as much as possible)
- The version of an event is **part of the event type**
  - ContractCreated, ContractCreatedV2, ContractCreatedV3
  - We've tried several approaches and preferred this (more explicit)
  - Multiple versions can exist at the same time
  - We never update events in the event store for versioning (really, never? ... no NEVER!!)
- If you cannot decide on a **default value** for new properties, it's **not a new version** of the event but rather a **new event type**!

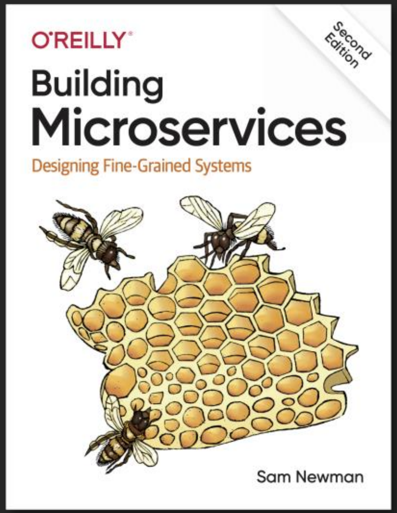
Event Sourcing - Event versioning

- We support 2 ways of handling multiple event versions
  - The aggregate supports multiple event versions
    - Explicit mapping in code
    - Part of unit-tests
  - The repository translates between event versions when deserializing
    - Using weak schema JSON deserialization
    - Adding JSON attributes on event properties
    - Using custom JsonConverters

Some useful resources


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

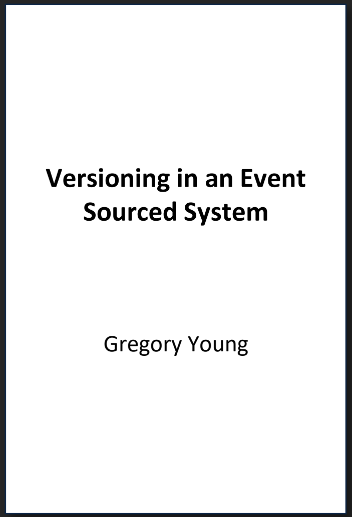
ISBN-13: 978-0321125217



ISBN-13: 978-1492034025

Microsoft
.NET Microservices: Architecture for Containerized .NET Applications
Cesar de la Torre
Microsoft Corporation

https://docs.microsoft.com/en-us/dotnet/architecture/microservices/


Versioning in an Event Sourced System

Gregory Young

https://leanpub.com/esversioning

https://github.com/edwinvw/cloud-native-net

Thank you!

Edwin van Wijk
Principal Architect

InfoSupport
Solid Innovator

MVP
Microsoft
Most Valuable Professional