Fundamental IDEALS and Domain Driven Design (DDD) for designing modern service-based systems

Joe Yoder – joe@refactory.com
Twitter: @metayoda
https://refactory.com

SOLID is for OO design

Single responsibility principle
Open/closed principle
Liskov substitution principle
Interface segregation principle
Dependency inversion principle

What if I'm designing services and microservices?

Microservice style

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. [1]

The microservice style dictates that the deployment unit should contain only one service or just a few cohesive services

This deployment constraint is the distinguishing factor[2]

[1] Lewis, J. & Fowler, M. "Microservices." 2014 martinfowler.com/articles/microservices.html

[2] Merson, P. "Defining Microservices." SATURN blog, 2015. insights.sei.cmu.edu/saturn/2015/11/defining-microservices.html

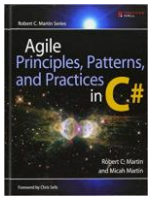
Guiding IDEALS for microservices

Interface segregation
Deployability
Event-driven
Availability over consistency
Loose Coupling
Single responsibility

Interface Segregation

Interface Segregation Principle

- *This principle deals with the disadvantages of "fat" interfaces […]*
- *The interfaces of the class can be broken up into groups of methods […]*
- *Each group serves a different set of clients*



Robert C. Martin & Micah Martin. *Agile Principles, Patterns, and Practices in C#.* Prentice Hall, 2006.

```
public Person createPerson(
    final String lastName,
    final String firstName,
    final String middleName,
    final String salutation,
    final String suffix,
    final String streetAddress,
    final String city,
    final String state,
    final char   gender,
    final boolean isEmployed,
    final boolean isHomeOwner)
  { // implementation goes here }
```

Interface Segregation for microservices

Services are often called by different types of clients, such as

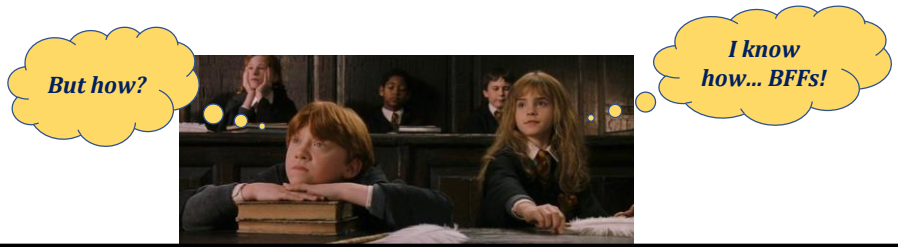
- Web applications, mobile apps, other backend services

Traditional SOA prescribed canonical schema:

- All clients should comply to the service contract

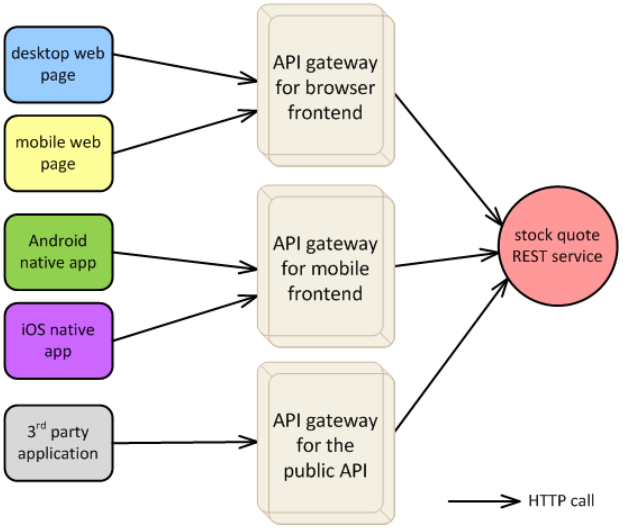
Today:

- Each client should see a service contract that best suits its needs



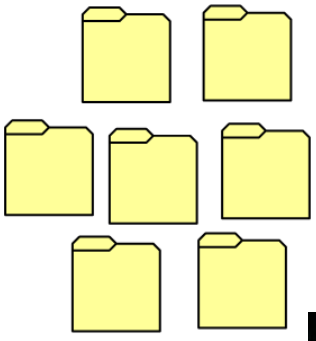
Backend for frontends (BFF)

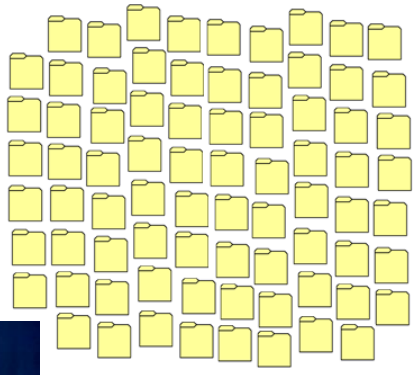
- Variation of API gateway
- One API gateway for each *type* of client (frontend)
- Each API gateway does routing, transformations, etc. as needed by each client
- Each frontend team can be responsible for their API gateway



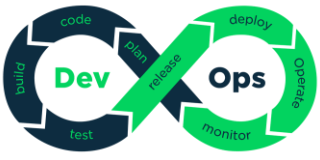
Deployability

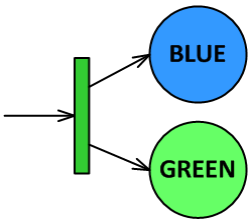
- Microservices hugely increased the number of deployment units

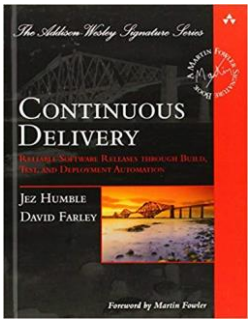




Good design and implementation alone don't warrant success

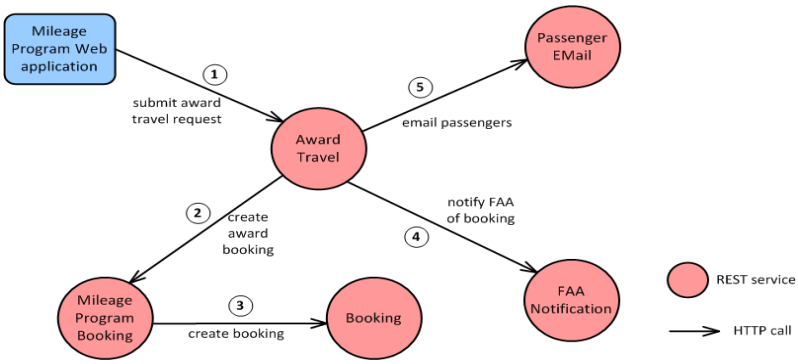






Event-Driven

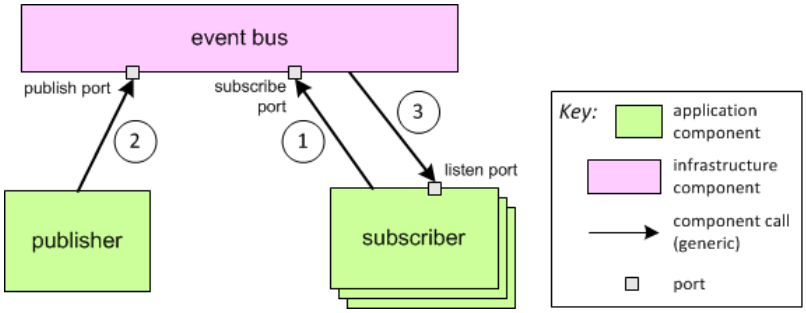
- Synchronous request-response calls are still everywhere



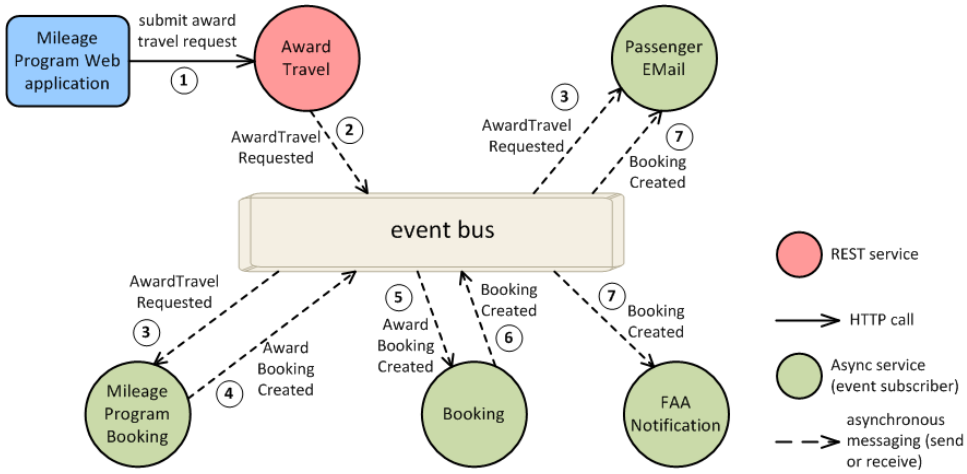
- But today's scalability and performance requirements pose a challenge that calls for events processed asynchronously

Event-Driven Architecture (EDA)

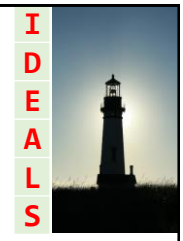
EDA is an architecture style in which components communicate primarily through asynchronous messages or events



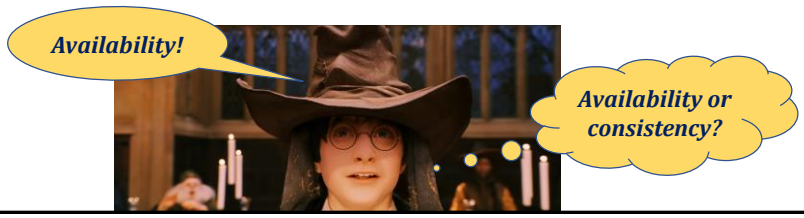
Event-driven example



Availability over consistency

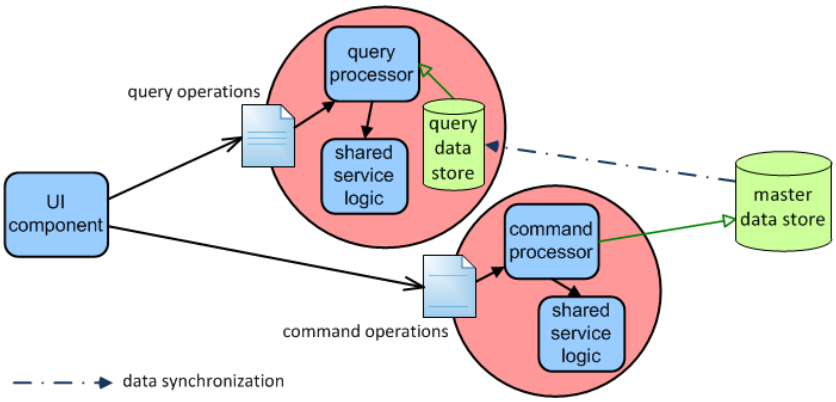


- The CAP theorem gives you two options: availability xor consistency
- We see enormous effort in industry to provide mechanisms to enable you to choose availability, ergo embrace *eventual* consistency
- Why? Users won't put up with lack of availability!



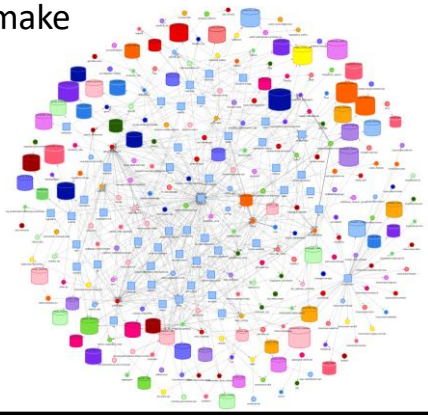
Availability over consistency in practice

- CQRS and Service Data Replication



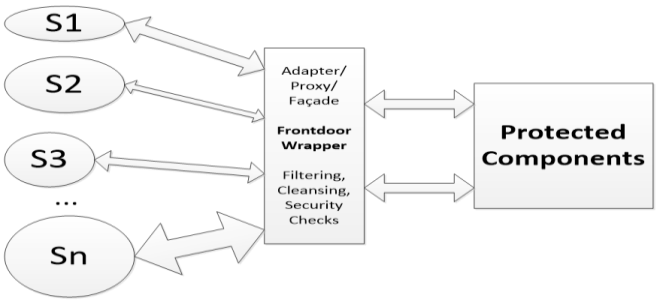
Loose coupling

- High coupling has dependency between components or services
- These interdependencies and connections can make the system harder to evolve and maintain



Loose coupling for microservices

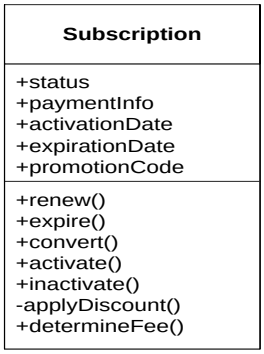
- Model around the business domain (DDD)
- Carefully design the contract
- Use wrapper patterns (adapter, façade, decorator, proxy)
- Use EDA, API Gateway, Asynchronous Messaging, Hypermedia, …



Single responsibility

Single Responsibility Principle:

- *If a class has more than one responsibility, [they] become coupled […]*
- *This kind of coupling leads to fragile designs that break in unexpected ways when changed […]*
- *[SRP] is one of the simplest of the principles, but one of the most difficult to get right*



Subscriptions: subscribe, status, promotions, payments

Single responsibility for microservices

If a microservice is packed with responsibilities,
it might bear the pains of the monolith

If its responsibility is too slim

- several microservices might need to interact to fulfill a request
- data changes might be spread across different microservices

Distrubuted transaction cadaver!

DDD to the rescue

DDD can help you define the size of your microservice

- Not the LOC size
- The size in terms of functional scope

*Let's look at how to **Model Microservices with DDD***

A well designed microservice shall have a single responsibility

Guiding IDEALS for microservices

Interface segregation
Deployability
Event-driven
Availability over consistency
Loose Coupling
Single responsibility

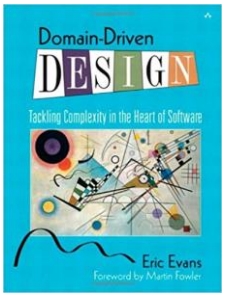
Modeling Microservices with DDD

Motivation

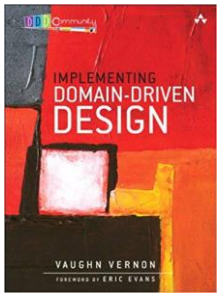
How do I model my microservices
What is a good size of a microservices
How do I avoid coupling problems
How do I deal with distributed data and transactions

Domain-Driven Design (DDD)

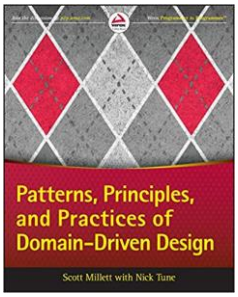
DDD is an approach to domain modeling created by Eric Evans
DDD is not an approach to microservice design
But DDD can help with some aspects of microservice design



2003

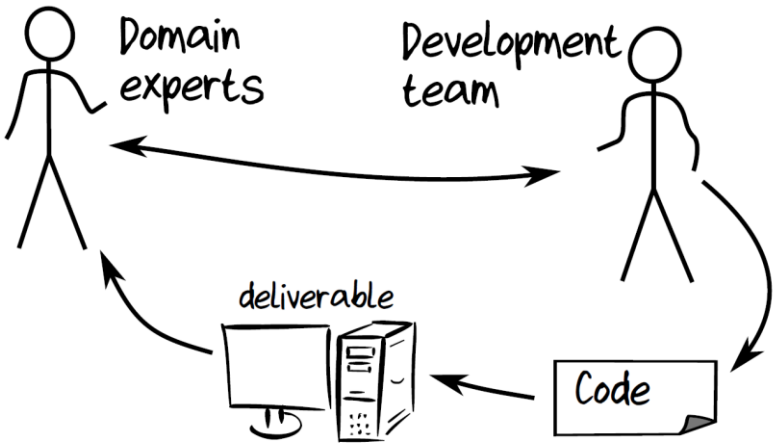


2013

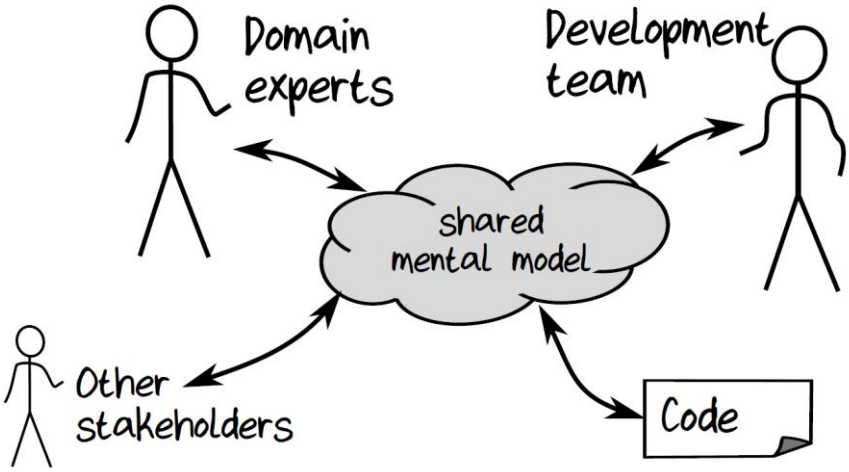


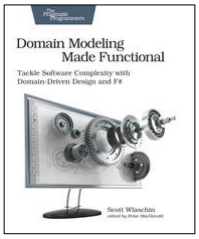
2015

Agile Approaches encourage Domain Experts



Variation with a Shared Mental Model





https://pragprog.com/book/swdddf/domain-modeling-made-functional

2018

DDD main concepts

Domain
- Core domain

Aggregate
- Entity, value object, aggregate root

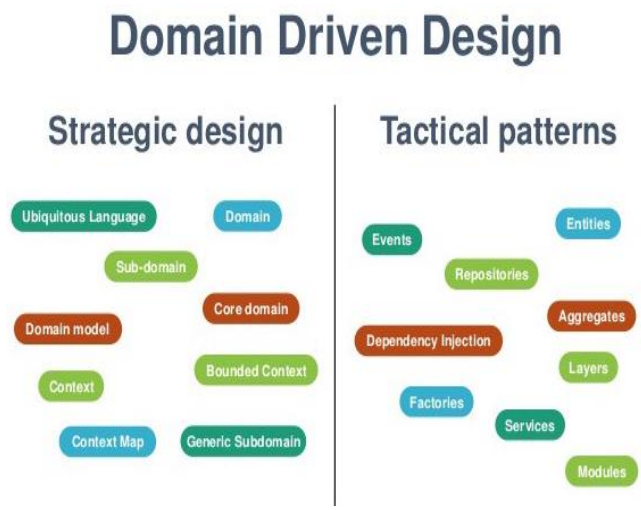
Bounded context
- Context map, Anticorruption Layer

Ubiquitous language

Application service, domain service

Repository

Domain event



Domain

Domain is the problem to be solved with software in an organization

It includes the concepts and business rules needed to achieve the business goals of the organization

Examples of organizations and their domains:
- DHL: shipping parcels
- Supreme Court: judicial cases involving the Constitution or federal law
- Angelo's Pizza: produce and sell pizza

Core domain

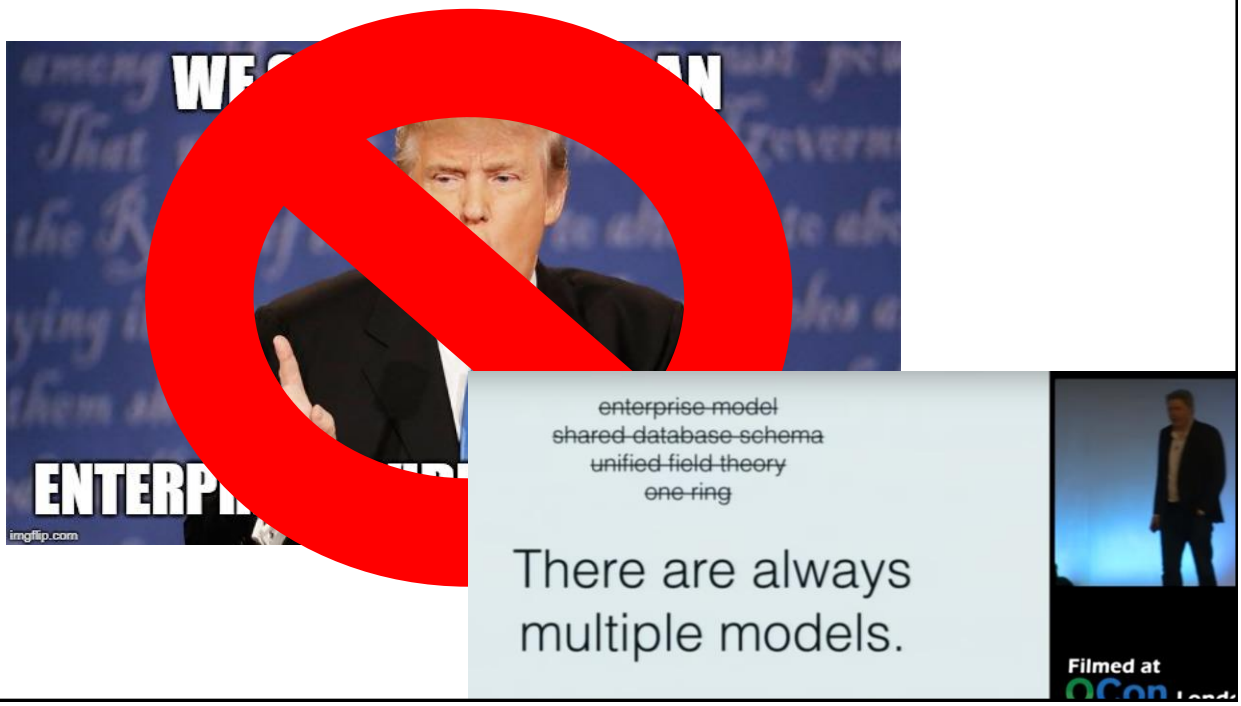
Domain is the generic term

A domain is typically composed of subdomains

A domain can be a

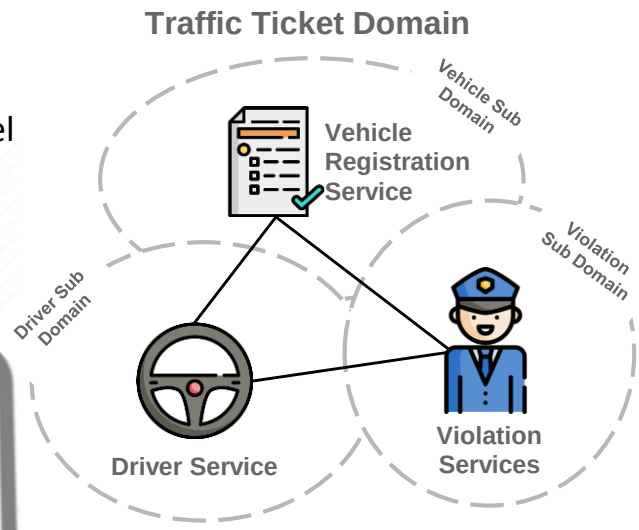
- *core domain*—is crucial for the success of the organization
- *supporting subdomain*—models important aspects of the business that are not core to the business
- *generic subdomain*—required by the business in an auxiliary fashion

The classification terms are not important; identifying core domains is important



Domain model

Each domain and subdomain
has its domain model



Entity

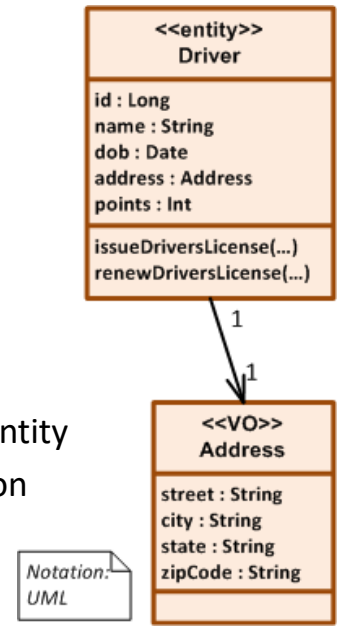
Entities have an ID and a life cycle, focus is on behavior, not data (rich object model)

Examples: Driver, Customer, Order, Payment

Value Object

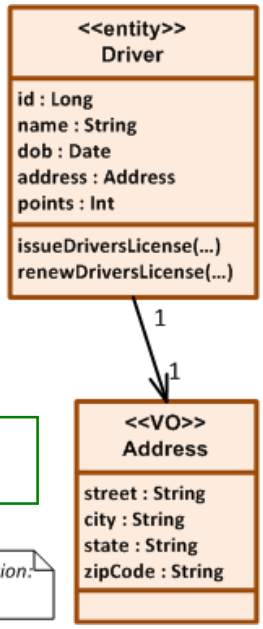
Value objects represent characteristics or values in an entity

Examples: Address, Amount, Distance, Price, Geolocation



Example of Entity using @Entity JPA annotation (ORM mapping details omitted)

```
@Entity
class TicketType(
        var id: Long,
        var description: String,
        var severity: Severity,
        var points: Int
) {
    enum class Severity { CITYVIOLATION, CIVILINFRACTION,
                                          TRAFFICMISDEMEANOR }
}
```



Example of VO using @Embeddable JPA annotation

```
@Embeddable
class LocationVO(
        var location: String,
        var municipalityId: Long
}
```

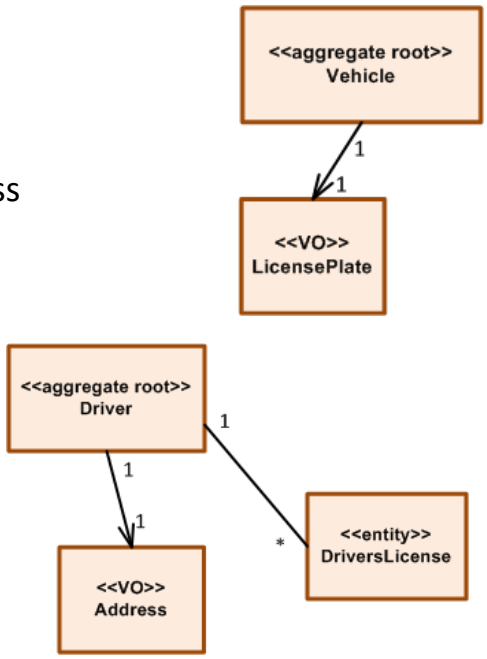
Note that a VO may hold a reference to an entity

Notation: UML

Aggregate

- An *aggregate* represents a cohesive business concept, such as Vehicle, Driver, Ticket, …
- An aggregate has one or more entities with possible value objects
- One entity is the *aggregate root*
- The typical aggregate has one entity and a few VOs, but aggregates with 2-3 entities are common

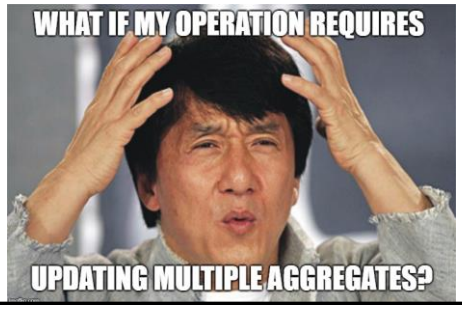
External objects/functions only see the aggregate through the aggregate root



Aggregate transactional consistency

- An aggregate defines a (transactional) consistency boundary
- It remains transactionally consistent throughout its lifetime
- It is often loaded in its entirety from the database
- If an aggregate is deleted, all of its objects are deleted

A database transaction should touch only one aggregate



Inter-aggregate references

- Aggregate A may reference aggregate B
- The reference must use the ID of aggregate B

DDD way ☺

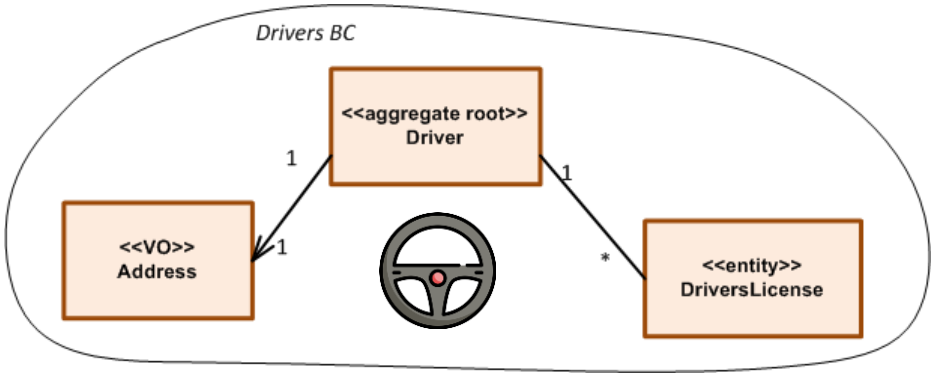
```
@Entity
class TrafficTicket(
    var id: Long,
    var dateTime: Date,
    var location: LocationVO,
    var vehicleId: Long,
    var ticketTypeId: Long
) {
    var driverId: Long? = null
    var notes: String? = null
}
```

Traditional OO way

```
@Entity
class TrafficTicket(
    var id: Long,
    var dateTime: Date,
    var location: LocationVO,
    var vehicle: Vehicle,
    var ticketType: TicketType
) {
    var driver: Driver? = null
    var notes: String? = null
}
```

Bounded Context

A *bounded context* (BC) delimits the scope of a domain model



Bounded Context

A *bounded context* (BC) delimits the scope of a domain model

The scope of a BC can be

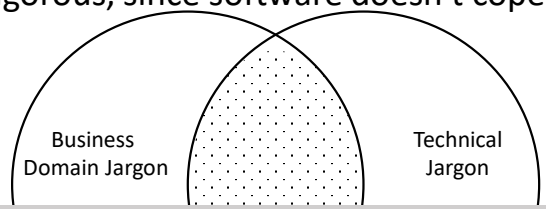
- The entire domain model of a subdomain (recommended)
- Domain models of 2+ subdomains (often happens with legacy systems)
- Part of the domain model of a subdomain (when we won't implement the other part)

In practice…

- The scope of a BC is often the scope of a traditional application system
- BCs are autonomous and a developer should be able to tell whether a concept is in or out of a BC

Ubiquitous language in a nutshell

- **Ubiquitous Language** is the term Eric Evans uses in **Domain Driven Design** for the practice of building up a **common**, **rigorous language** between **developers** and **domain experts**. This language should be **based on** the **Domain Model** used in the software - hence the need for it to be rigorous, since software doesn't cope well with ambiguity.



Domain experts should object to terms or structures that are awkward or inadequate to convey domain understanding; developers should watch for ambiguity or inconsistency that will trip up design.
-- Eric Evans

Domain Events

A domain event
- is something of interest that has happened to an aggregate
- should be expressed in past tense
- typically represents state change
- should be represented by a class in the domain model
- may be organized in an event class hierarchy

Examples:
- Traffic Ticket Issued
- Traffic Ticket Paid
- Driver Created
- Driver’s License Suspended

What's the right size of a microservice?

If it's too large, it might bear the challenges of a monolith

If it's too small:

- Several microservices might need to interact to fulfill a request
- Data changes might be spread across different microservices
- Distributed transactions might be needed

DDD can help you define the size of your microservice

- Not the LOC size
- The size in terms of functional scope

Before we discuss *how* we need to understand *what* is a microservice

What is a microservice *in practice*?

- Let's build an example with a REST (http) backend service

This service exposes 2 endpoints

```
@RestController
@RequestMapping("api")
class TrafficTicketController(val applicationService: TrafficTicketService) {

    @PostMapping("/traffic-ticket")
    fun createTicket(@RequestBody trafficTicketDto: TrafficTicketDto, response: HttpServletResponse):
            ResponseEntity<TrafficTicketDto?> {
        val newTrafficTicketDto = applicationService.create(trafficTicketDto)
        return ResponseEntity(newTrafficTicketDto, HttpStatus.OK)
    }

    @PutMapping("/traffic-ticket/{id}")
    fun updateTicket(@RequestBody trafficTicketDto: TrafficTicketDto):
            ResponseEntity<TrafficTicketDto?> {
        // . . .
    }
```

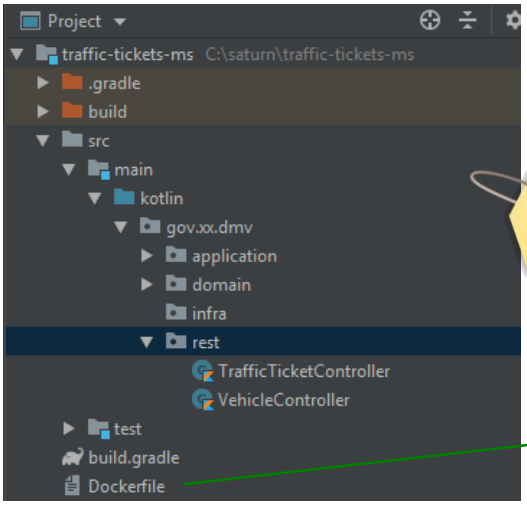
The @RestController typically calls the DDD application service

```kotlin
@RestController
@RequestMapping("api")
class VehicleController(val applicationService: VehicleService) {

    @PostMapping("/vehicle")
    fun createVehicle(@RequestBody vehicleDto: VehicleDto, response: HttpServletResponse):
            ResponseEntity<VehicleDto?> {
        val newVehicleDto = applicationService.create(vehicleDto)
        return ResponseEntity(newVehicleDto, HttpStatus.OK)
    }

    @GetMapping("/vehicle/plate/{plate}")
    fun getVehicleByLicensePlate(. . .)
```

This is a SERVICE

- TrafficTicketController and VehicleController are both REST services
- But are they microservices?

If both services are part of the same deployment unit, then it's one microservice



This is a MICROSERVICE with two services

The deployment unit in this case is a docker image

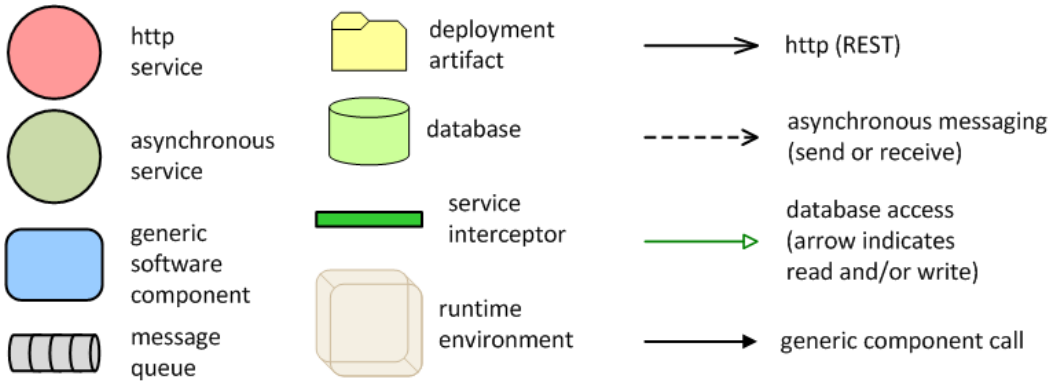
Scenarios for microservice scope and interaction

1. One-aggregate BC, one service, one microservice
2. A few aggregates in the BC, a few services, one microservice
3. Two BCs, two microservices, they interact via events
4. Two BCs, two microservices, they interact via API calls with ACL
5. Two BCs, two microservices, they interact via data replication

For operations handled entirely within the BC

For operations that require inter BC interaction

Notation used in diagrams showing microservices

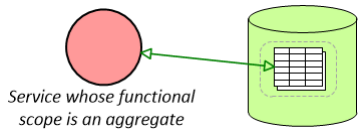


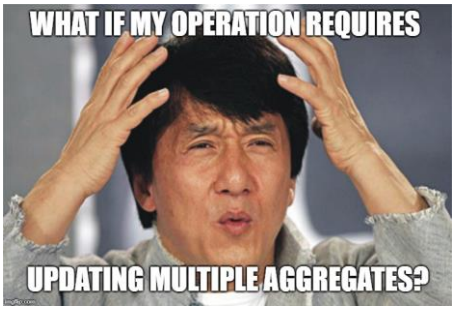
DDD and microservice scope (1)

DDD can help define the microservice size

Scenario 1: a data changing operation affects a single aggregate

- One aggregate → one service
- One service → one microservice



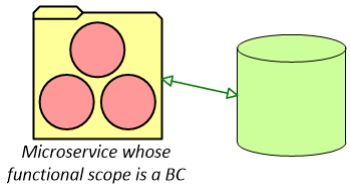


DDD and microservice scope (2)

Scenario 2: operation affects a few aggregates within the same BC

- Each aggregate → one service
- A few aggregates → one BC
- One BC → one microservice

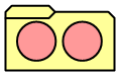
No distributed transaction because services run in the same VM

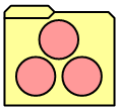


Putting it simply

A single service… 

can be packaged as a microservice 

But a microservice may contain 2 services… 

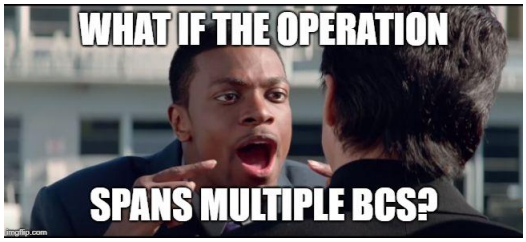
or 3… 

or even more, as long as they're **cohesive**

DDD and microservice scope (3)

- Domain-level business logic spanning multiple aggregates can be placed in a *domain service*
- The domain service interacts with different entities in the same BC

Suggestion: create a @DomainService annotation that is @Transactional



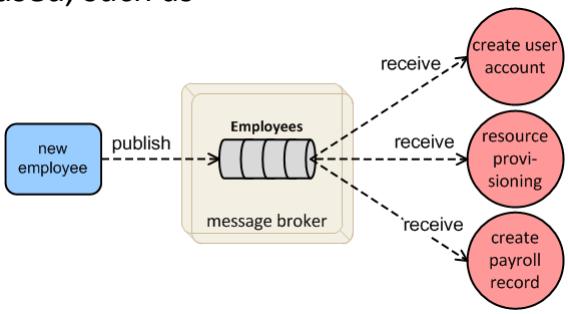
Transactions over multiple BCs

Scenario 3: operation affects data in different BCs

- Each BC → one microservice
- Use domain events for inter microservice communication

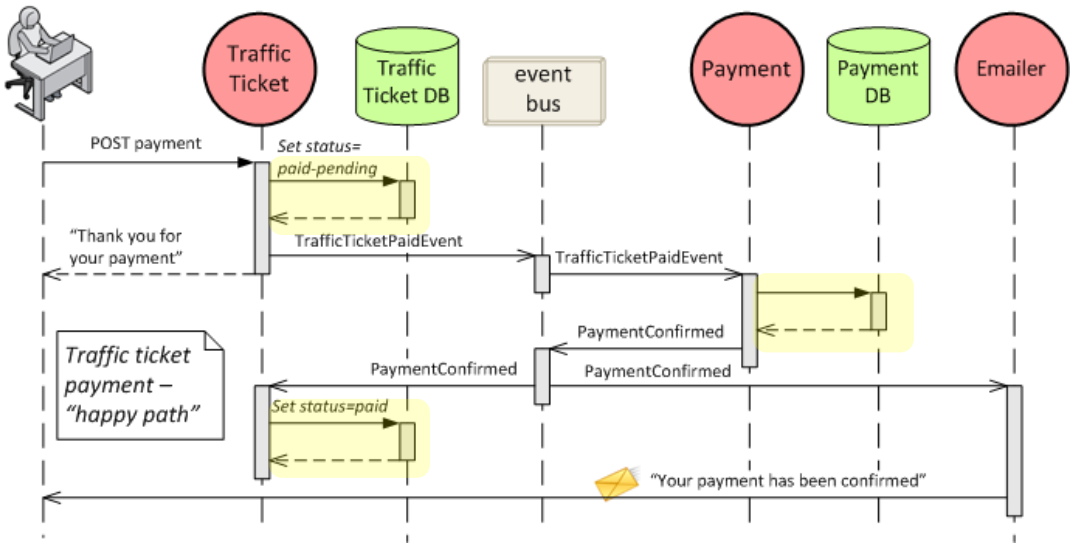
Publish-subscribe technologies can be used, such as

- Kafka
- RabbitMQ
- Vert.x
- Akka
- Eventuate Tram

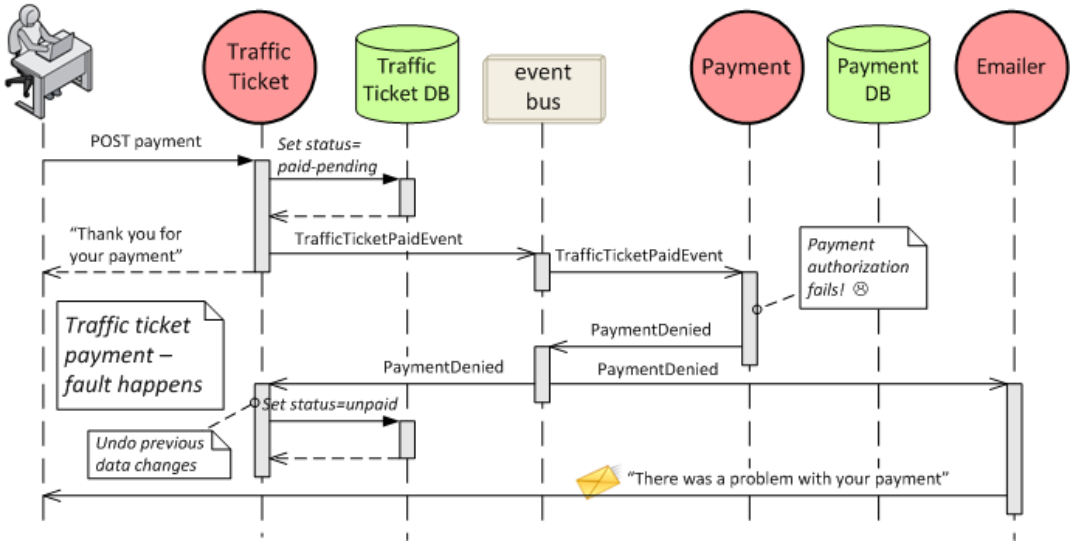


Event-based saga example (1)

Local DB transaction



Event-based saga example (2)



Event-based interaction – benefits

Maintainability

- Publishers and subscribers are independent and hence loosely coupled
- There's more flexibility to add functionality by simply adding subscribers or events

Scalability and throughput

- Publishers are not blocked, and events can be consumed by multiple subscribers in parallel

Availability and reliability:

- Temporary failures in one service are less likely to affect the others

Event-based interaction – challenges (1)

Maintainability

- The event-based programming model is more complex:
- Some of the processing happens in parallel and may require synchronization points
- Correction events, and mechanisms to prevent lost messages may be needed
- Correlation identifiers may be needed

Testability

- Testing and monitoring the overall solution is more difficult

Interoperability and portability

- The event bus may be platform specific and cause vendor lock-in

Event-based interaction – challenges (2)

- Good UX is harder if end user needs to keep track of events
- We traded transactional consistency for *eventual consistency*

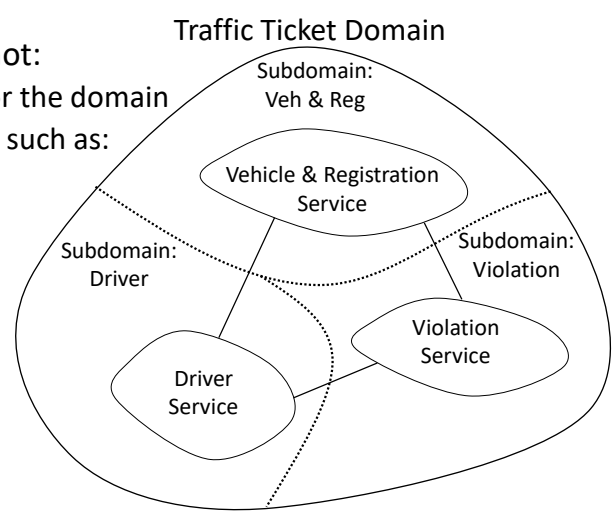
Takeaways

- IDEALS are good design principles for designing microservices
- Domain Driven Design (DDD) can help with defining microservices
- DDD key concepts (for microservice design) are domain, subdomain, bounded context, aggregate, and entity
- A service (e.g., REST) can have the scope of an aggregate
- Model a microservice around the bounded context
- We can use domain events for inter-microservice (i.e., inter-BC) interaction

Takeaways

Whether you use DDD or not,
or you are creating microservcies or not:

- Model around business capabilities or the domain
- Model the domain by using concepts such as:
  - entities,
  - aggregates,
  - bounded context,
  - ubiquitous language



Guiding IDEALS for microservices

Interface segregation
Deployability
Event-driven
Availability over consistency
Loose Coupling
Single responsibility

Joseph Yoder
https://refactory.com
joe@refactory.com
Twitter @metayoda

Thanks to Paulo Merson