Modeling Microservices with DDD

Paulo Merson – pmerson@acm.org

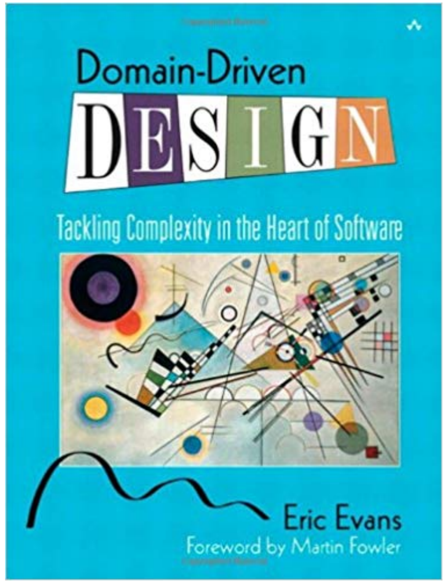
Joe Yoder – joe@refactory.com

Domain-Driven Design (DDD)

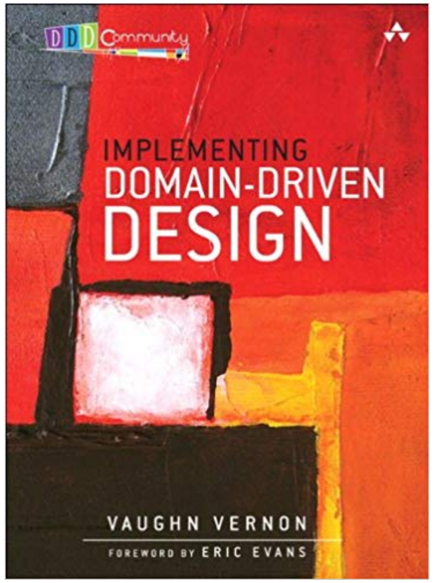
DDD is an approach to domain modeling created by Eric Evans

DDD is not an approach to microservice design

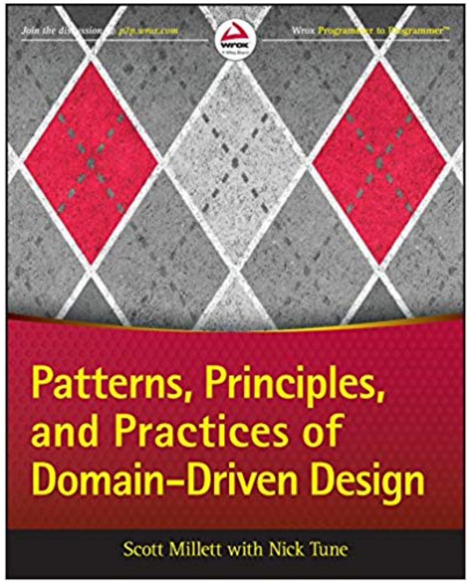
But DDD can help with some aspects of microservice design


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

2003



2013



2015

DDD key concepts for Microservices

Domain

- Core domain

Aggregate

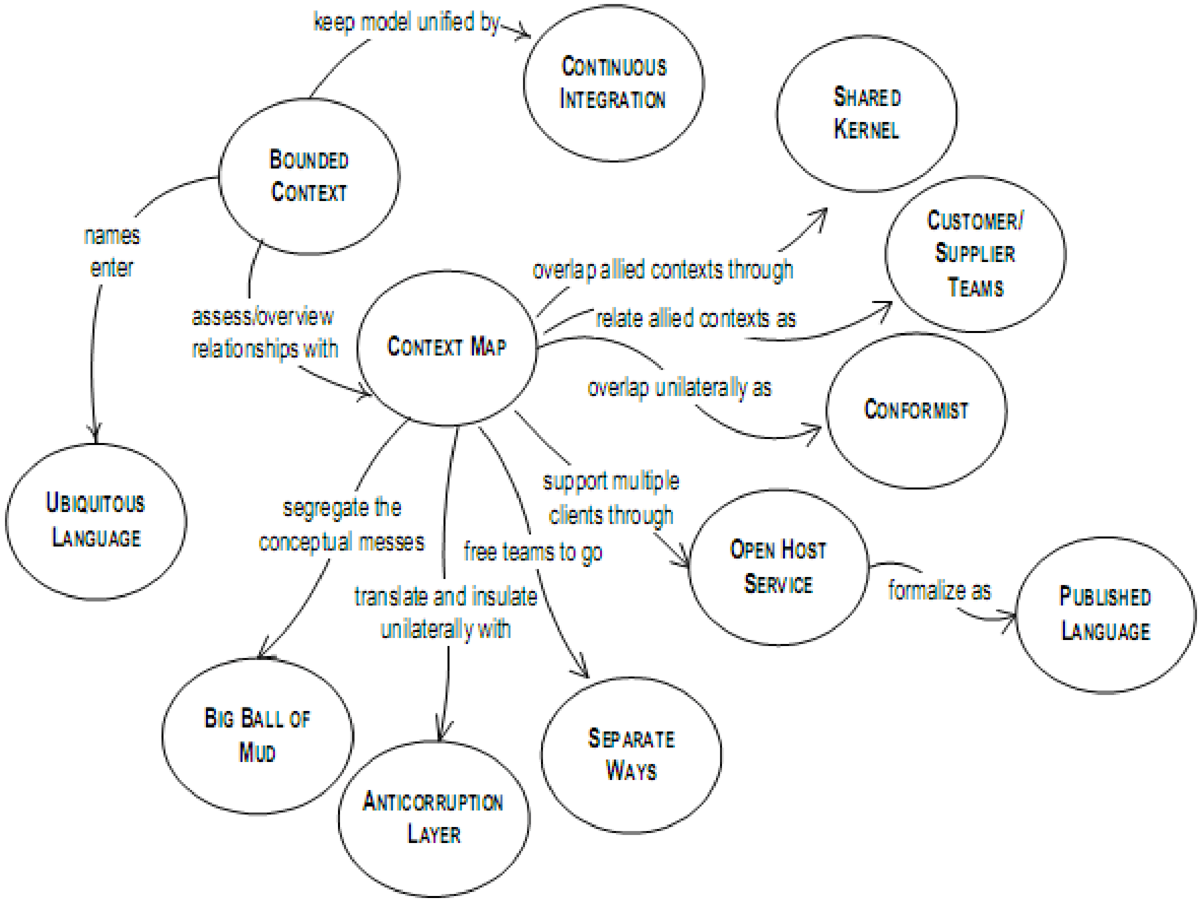
- Entity, value object, aggregate root

Bounded context

Ubiquitous Language

Domain event

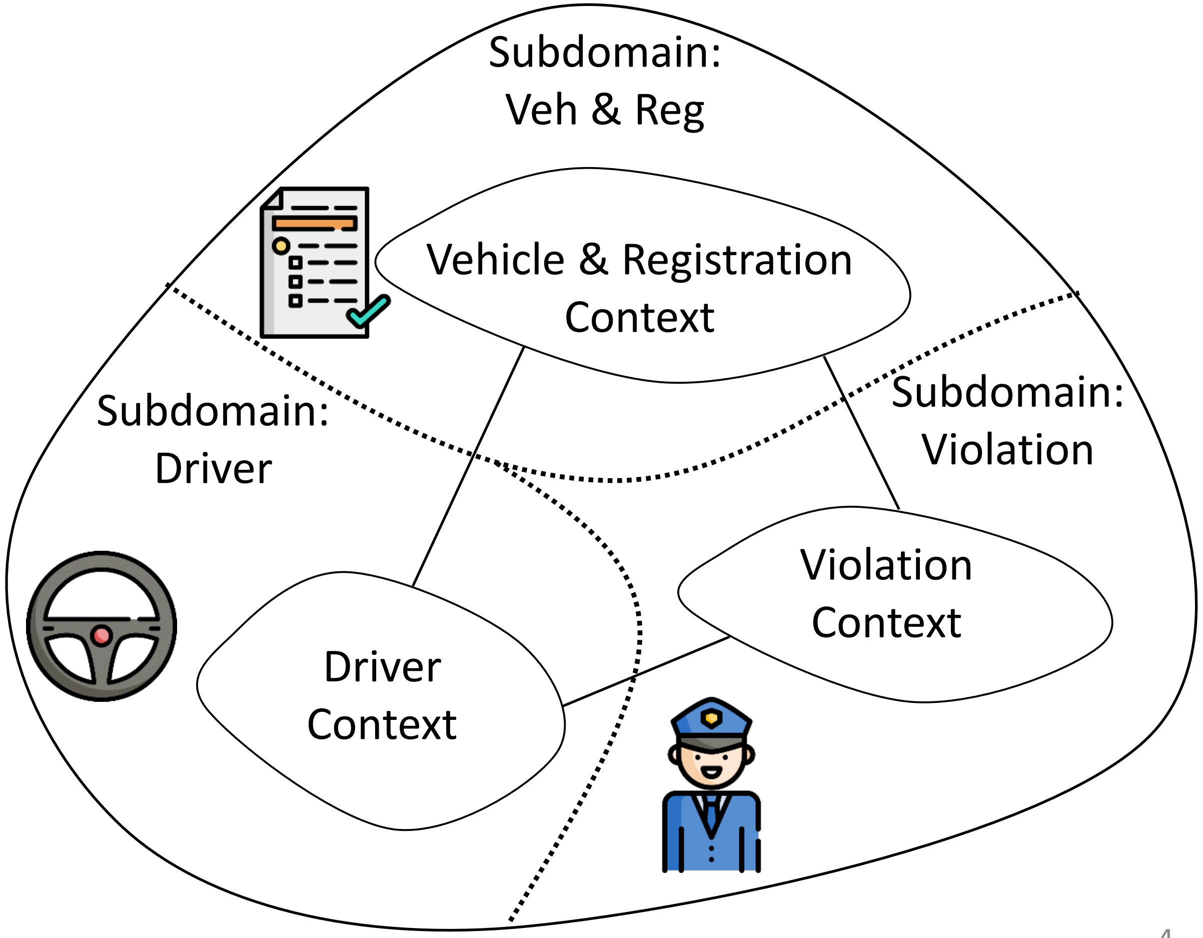
Maintaining Model Integrity



Domain model

Each domain and subdomain
has its domain model

Traffic Ticket Domain



Entity

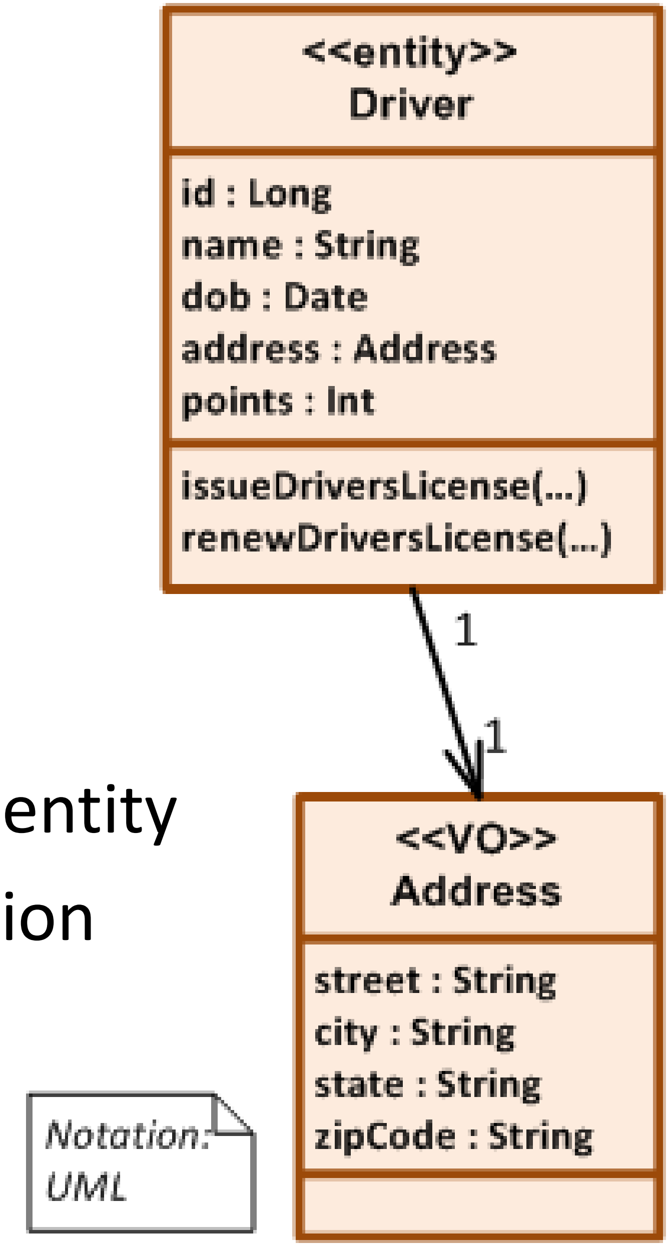
Entities have an ID and a life cycle, focus is on behavior, not data (rich object model)

Examples: Driver, Customer, Order, Payment

Value Object

Value objects represent characteristics or values in an entity

Examples: Address, Amount, Distance, Price, Geolocation



<<entity>>
Driver

id : Long
name : String
dob : Date
address : Address
points : Int

issueDriversLicense(...)
renewDriversLicense(...)

<<VO>>
Address

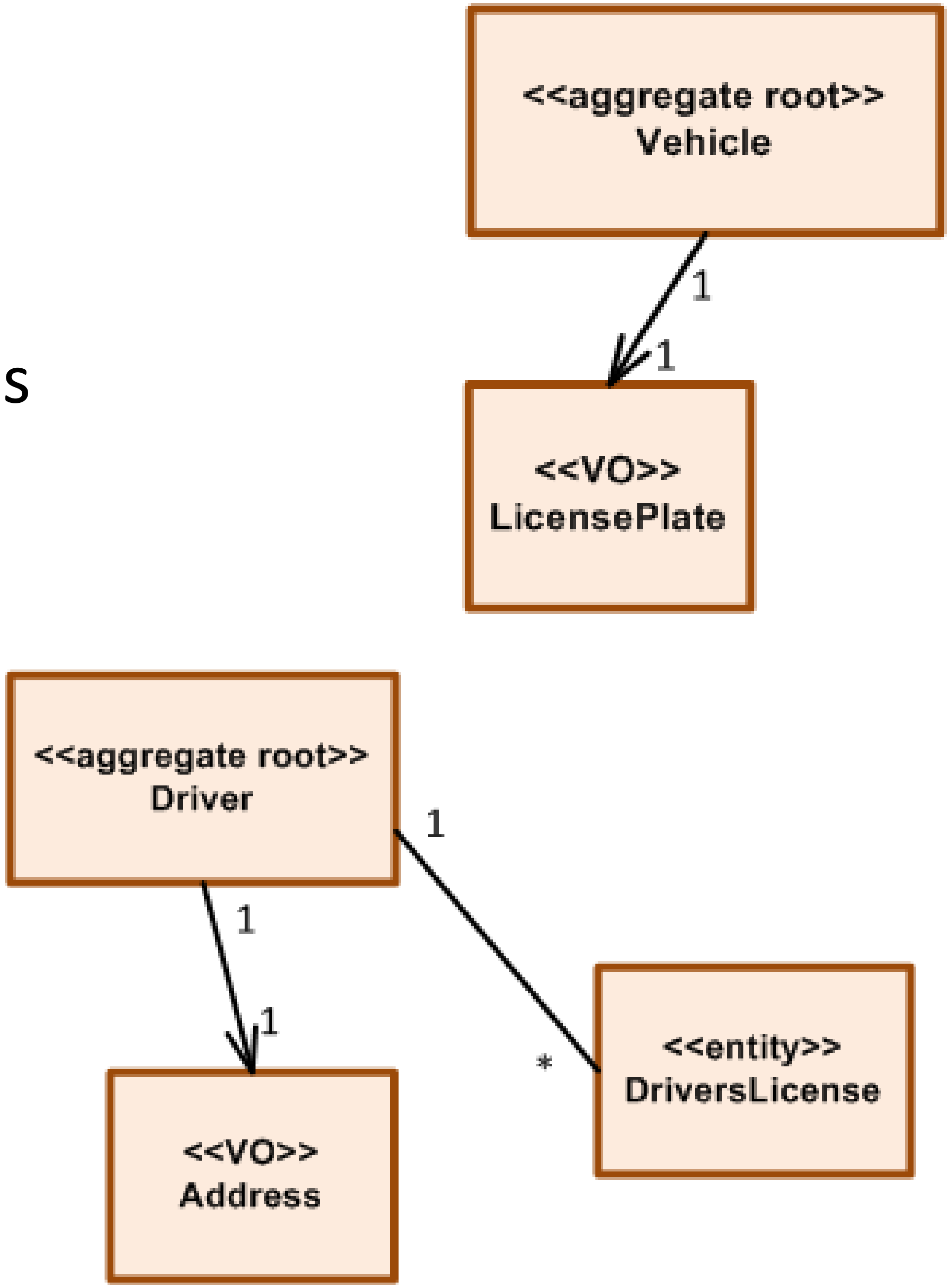
street : String
city : String
state : String
zipCode : String

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*

External objects 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

What if my operation requires updating multiple aggregates?

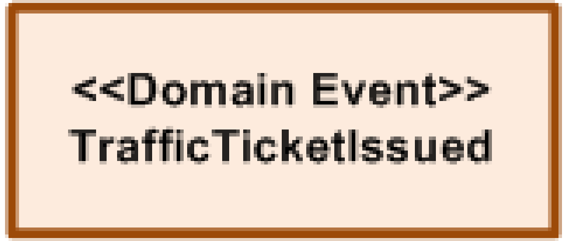
Domain Events

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



<<Domain Event>>
TrafficTicketIssued

Bounded Context

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

The scope of a BC can be

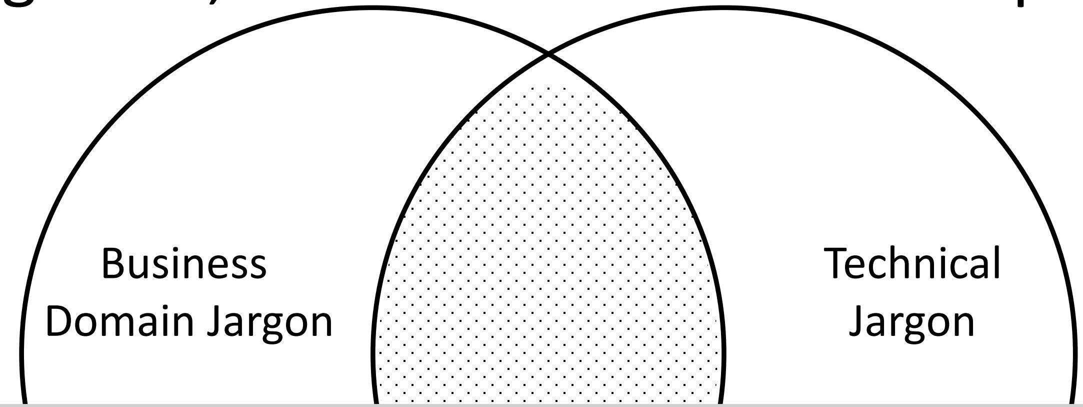
- The entire domain model of a subdomain (recommended)
- Domain models of 2+ subdomains (often happens with legacy systems)
- Part of the domain model of a subdomain (when we won't implement the other part)

In practice…

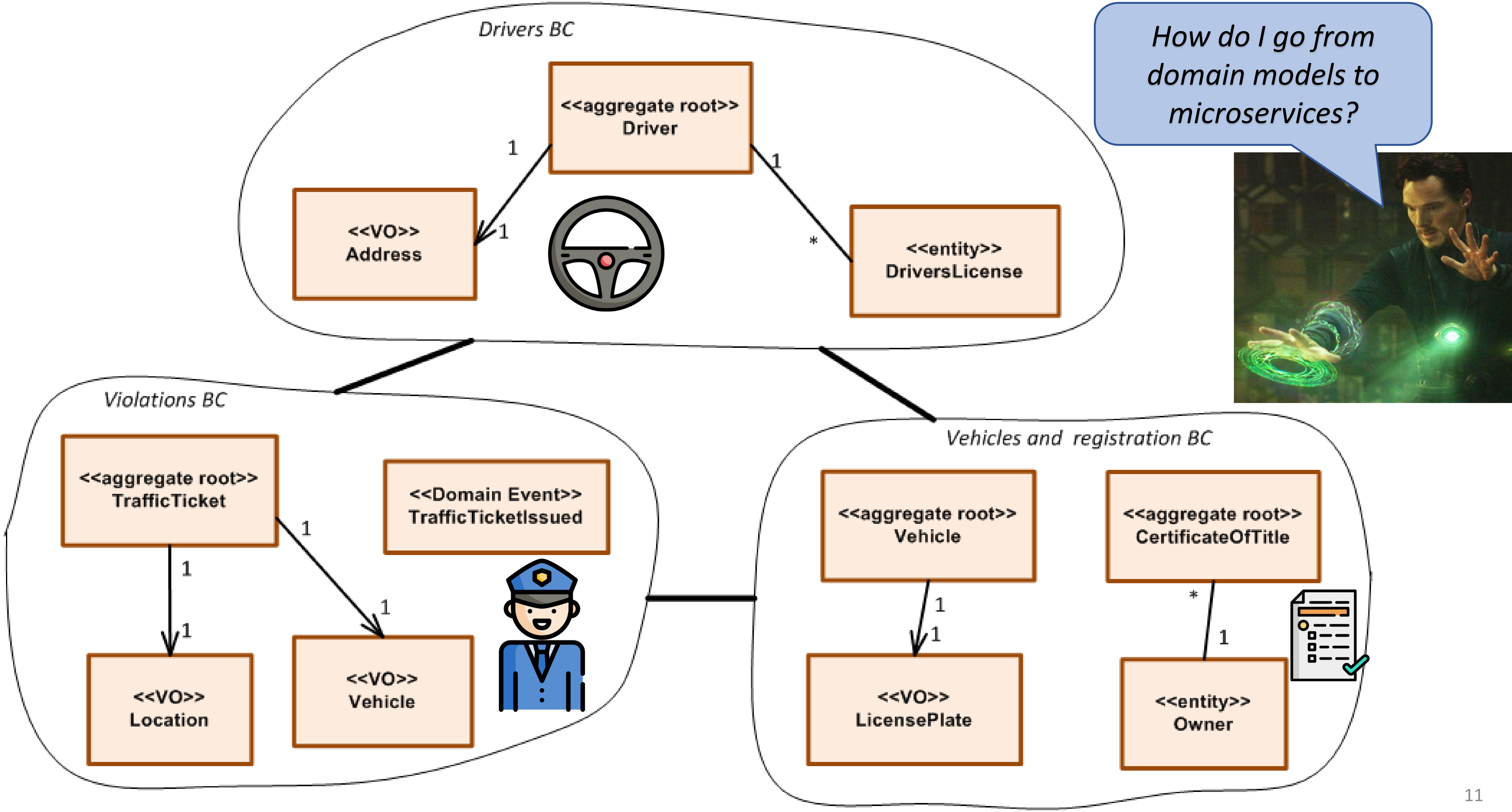
- The scope of a BC is often the scope of a traditional application system
- BCs are autonomous and a developer should be able to tell whether a concept is in or out of a BC

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*



How do I go from domain models to microservices?

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

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

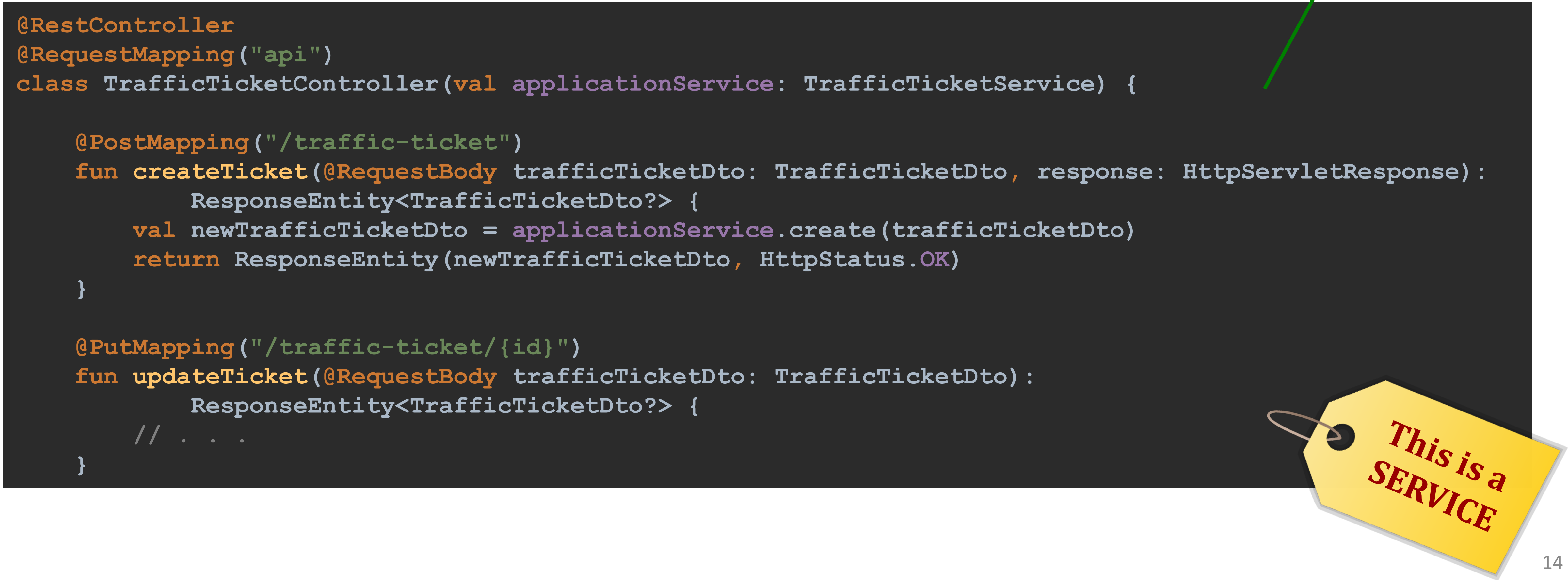
DDD can help you define the size of your microservice—not the LOC size, the size in terms of functional scope

Too many small microservices can kill your design

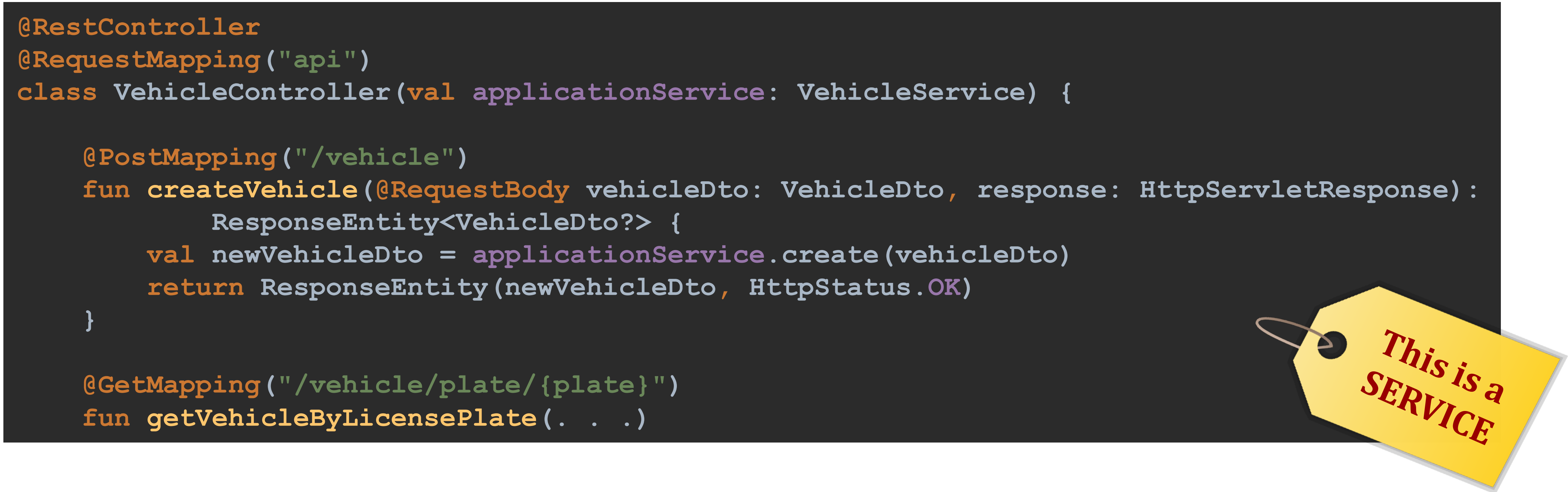
What is a microservice *in practice*?

- 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?> {
        // . . .
    }
```

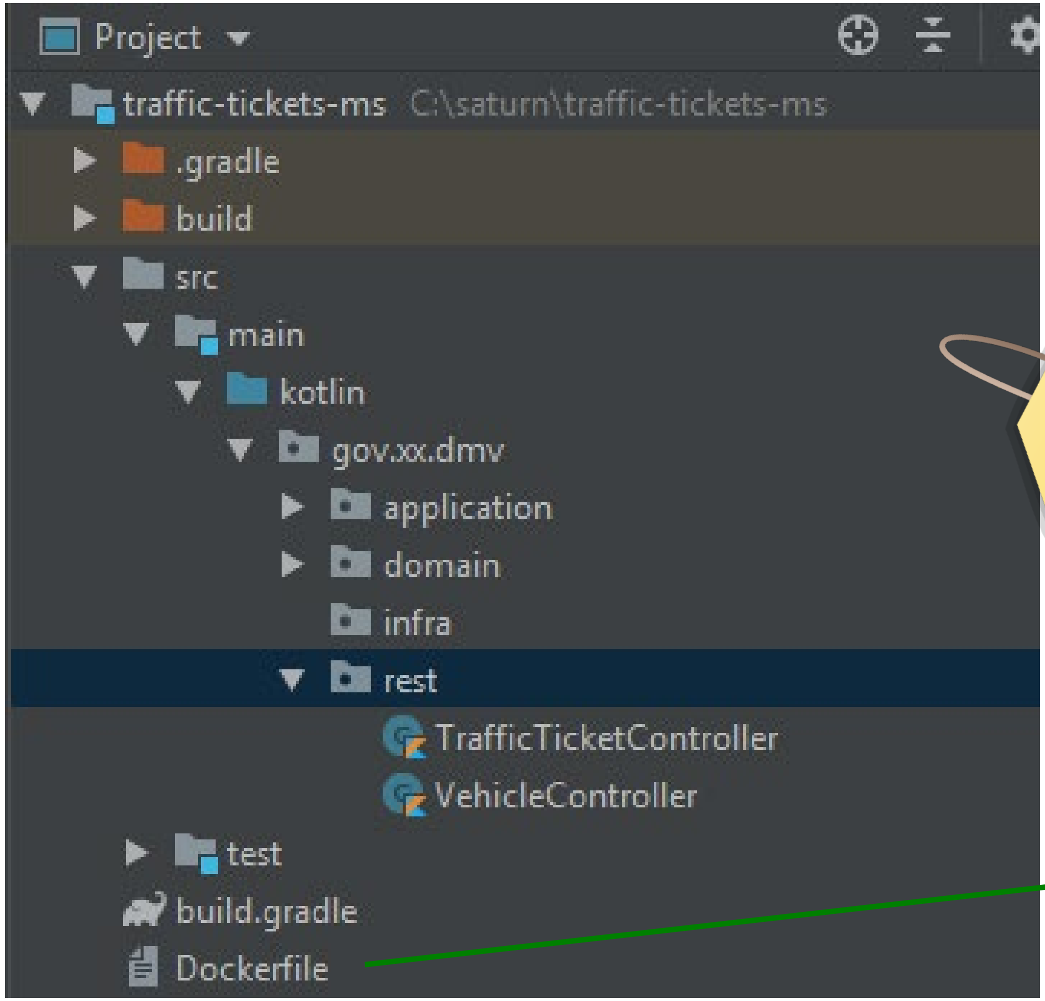


```
@RestController
@RequestMapping("api")
class VehicleController(val applicationService: VehicleService) {

    @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(. . .)
```

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

I don't know yet. How are the services deployed?

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



```
Project
traffic-tickets-ms  C:\saturn\traffic-tickets-ms
  .gradle
  build
  src
    main
      kotlin
        gov.xx.dmv
          application
          domain
          infra
          rest
            TrafficTicketController
            VehicleController
    test
  build.gradle
  Dockerfile
```

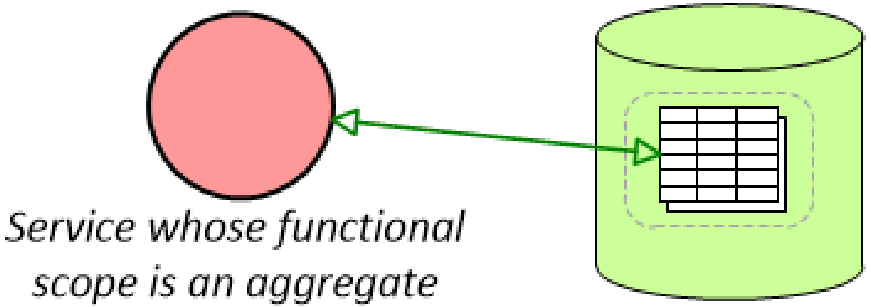
This is a MICROSERVICE with two services

The deployment unit in this case is a docker image

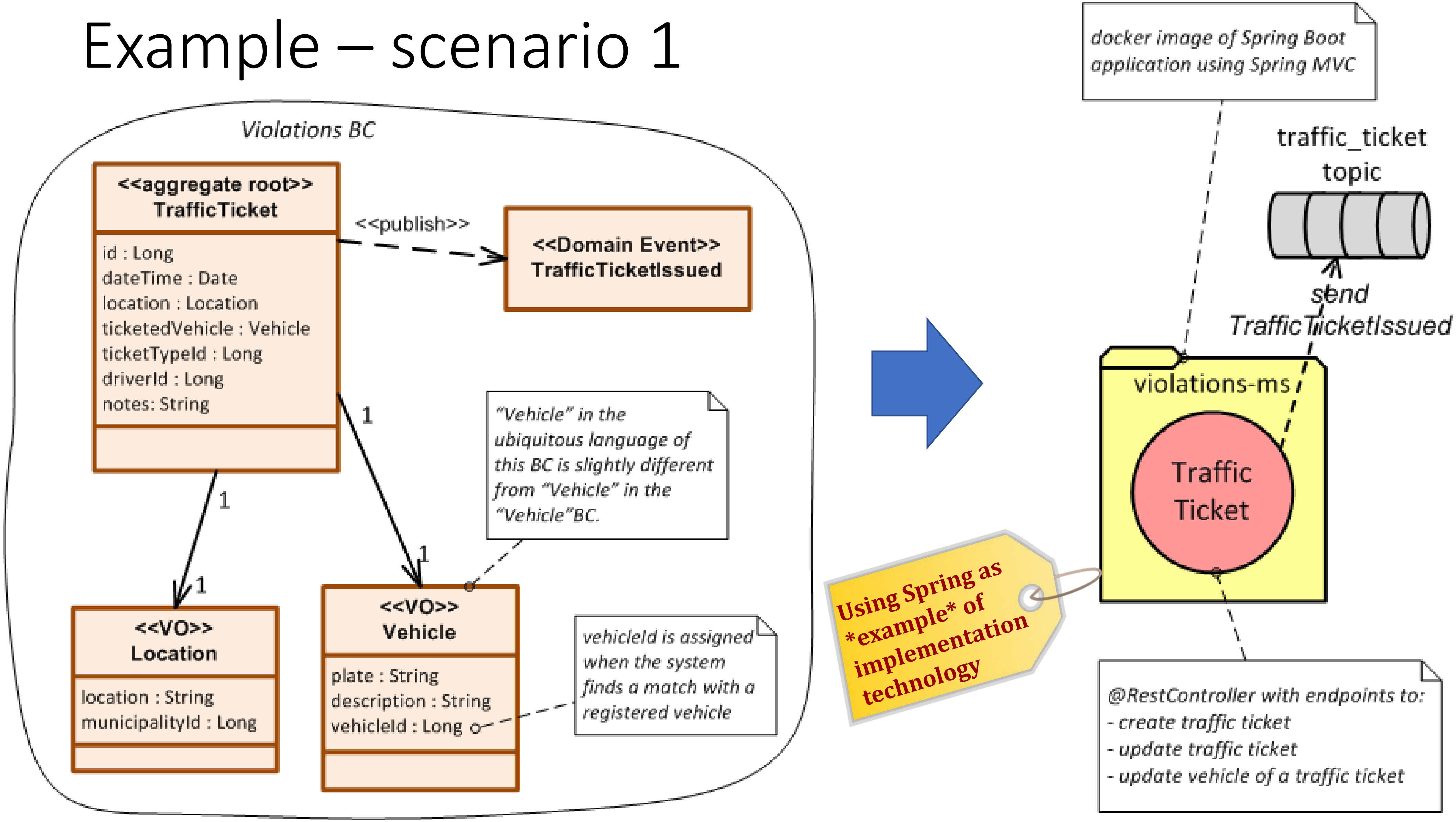
DDD and microservice scope – scenario 1

Scenario 1: data changing operations affect a single aggregate

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



Example – scenario 1



"Vehicle" in the ubiquitous language of this BC is slightly different from "Vehicle" in the "Vehicle"BC.

vehicleId is assigned when the system finds a match with a registered vehicle

docker image of Spring Boot application using Spring MVC

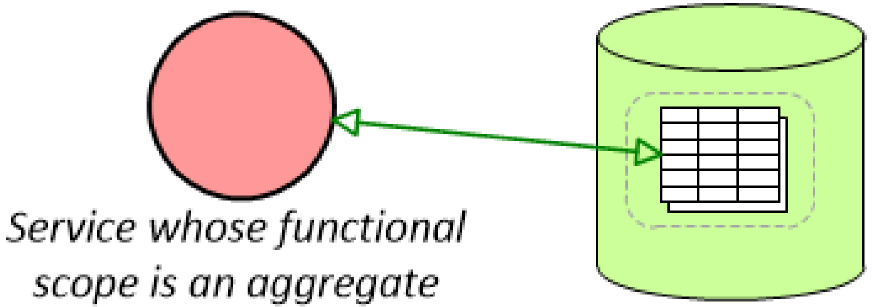
Using Spring as *example* of implementation technology

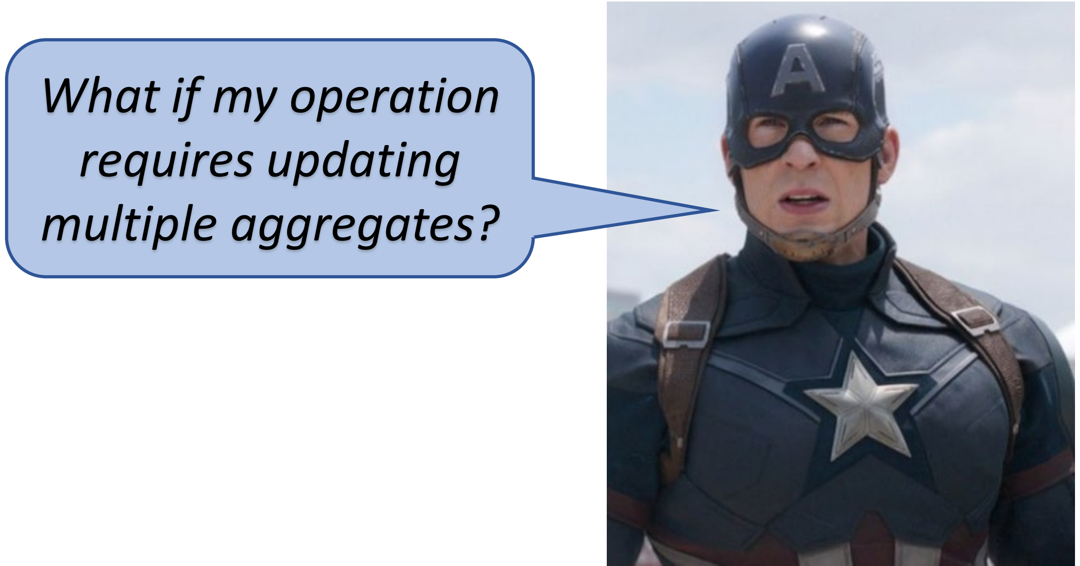
@RestController with endpoints to:
- create traffic ticket
- update traffic ticket
- update vehicle of a traffic ticket

DDD and microservice scope – scenario 1

Scenario 1: data changing operations affect a single aggregate

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



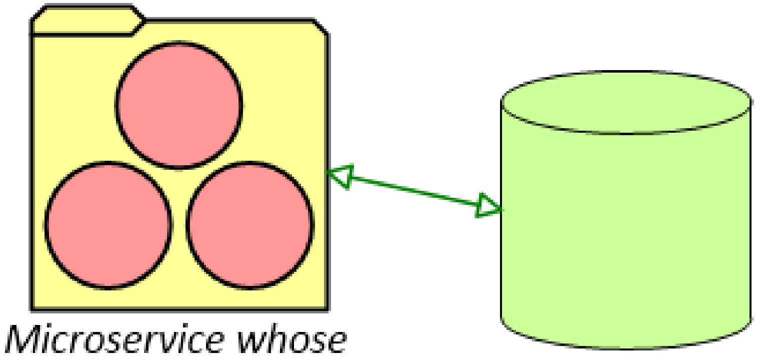


DDD and microservice scope – scenario 2

Scenario 2: operations affect a few aggregates within the same BC

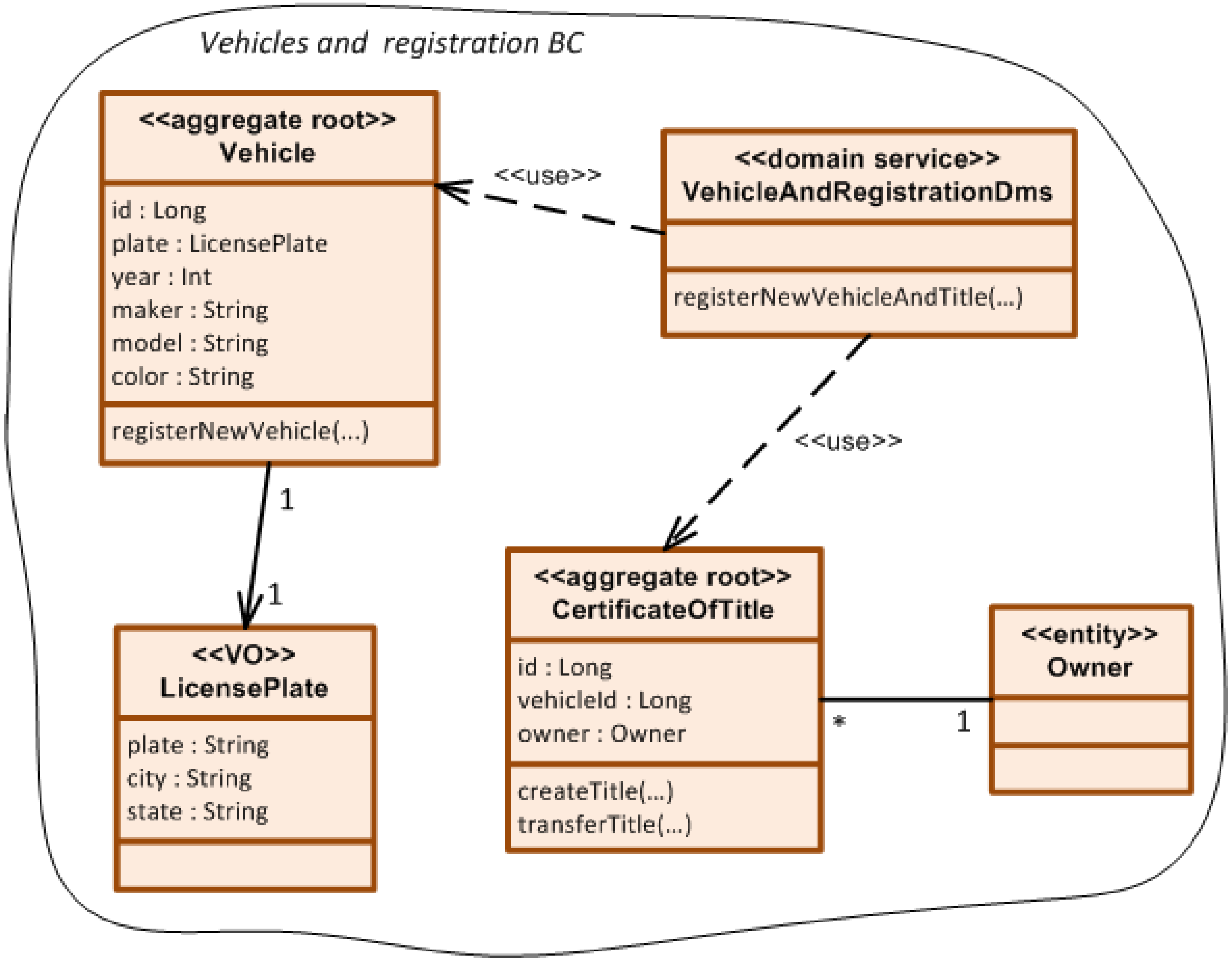
- Each aggregate → one service
- A few aggregates → one BC
- One BC → one microservice

No distributed transaction because services run in the same VM



Microservice whose functional scope is a BC

Example – scenario 2



Vehicles and registration BC

<<aggregate root>>
Vehicle

id : Long
plate : LicensePlate
year : Int
maker : String
model : String
color : String

registerNewVehicle(...)

<<domain service>>
VehicleAndRegistrationDms

registerNewVehicleAndTitle(...)

<<VO>>
LicensePlate

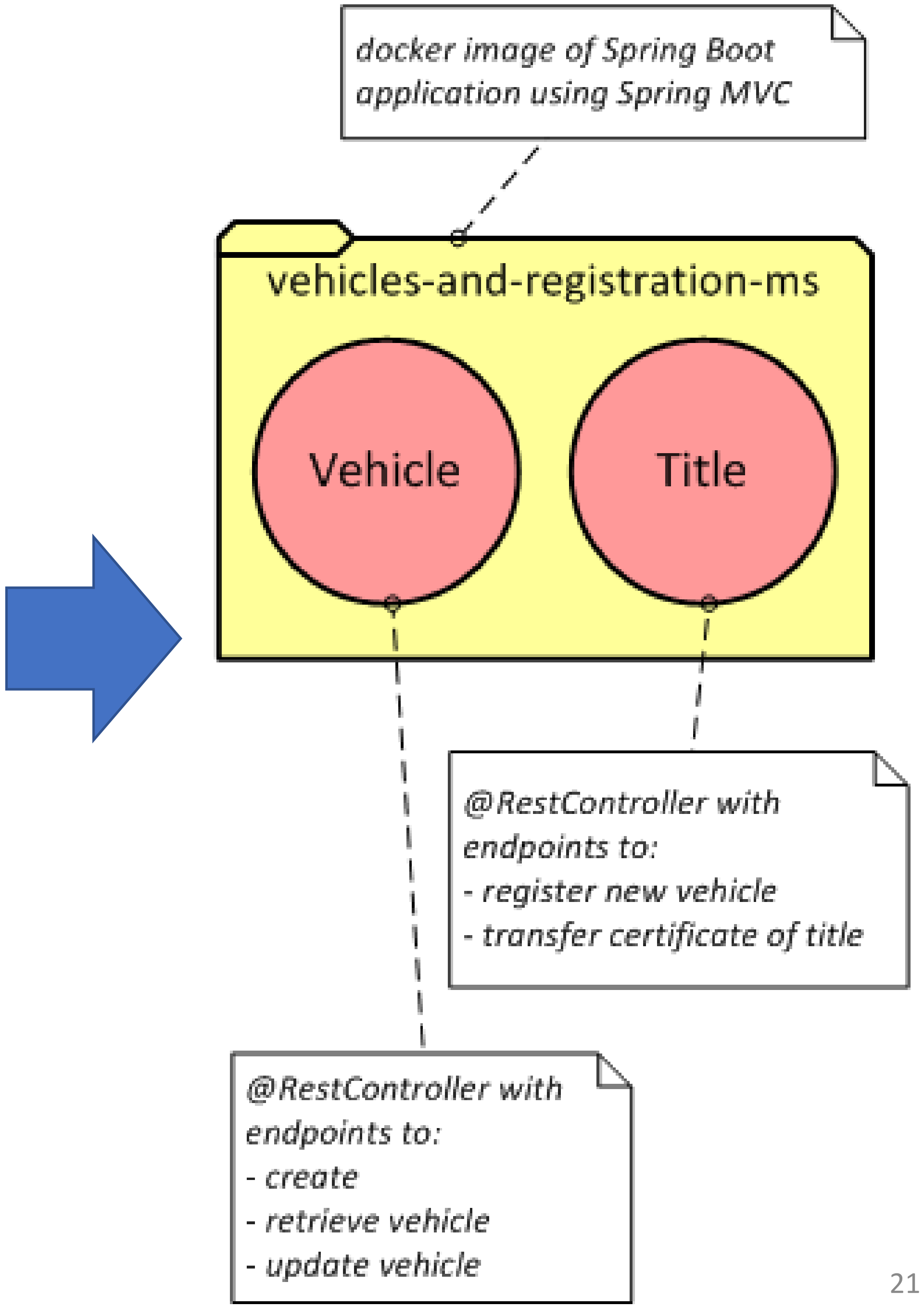
plate : String
city : String
state : String

<<aggregate root>>
CertificateOfTitle

id : Long
vehicleId : Long
owner : Owner

createTitle(...)
transferTitle(...)

<<entity>>
Owner



docker image of Spring Boot application using Spring MVC

vehicles-and-registration-ms

@RestController with endpoints to:
- register new vehicle
- transfer certificate of title

@RestController with endpoints to:
- create
- retrieve vehicle
- update vehicle

Cross-entity domain logic

- 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

What if the operation spans multiple BCs?

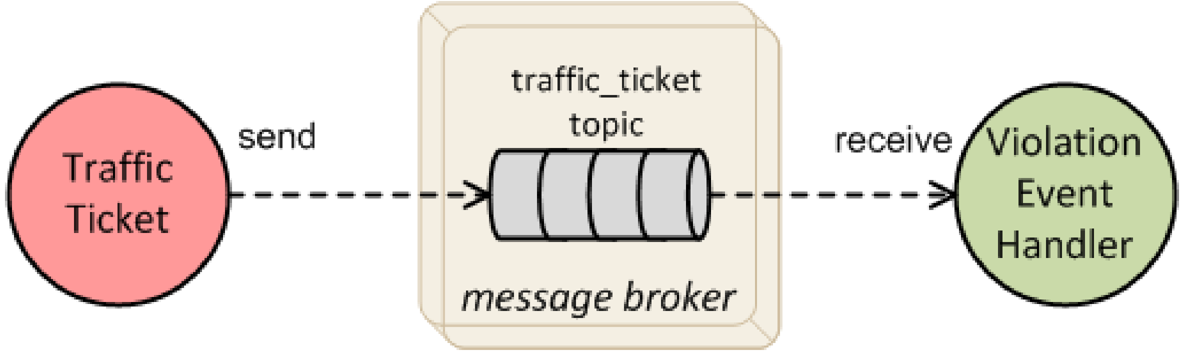
DDD and microservice scope – scenario 3

Scenario 3: operations affect data in different BCs

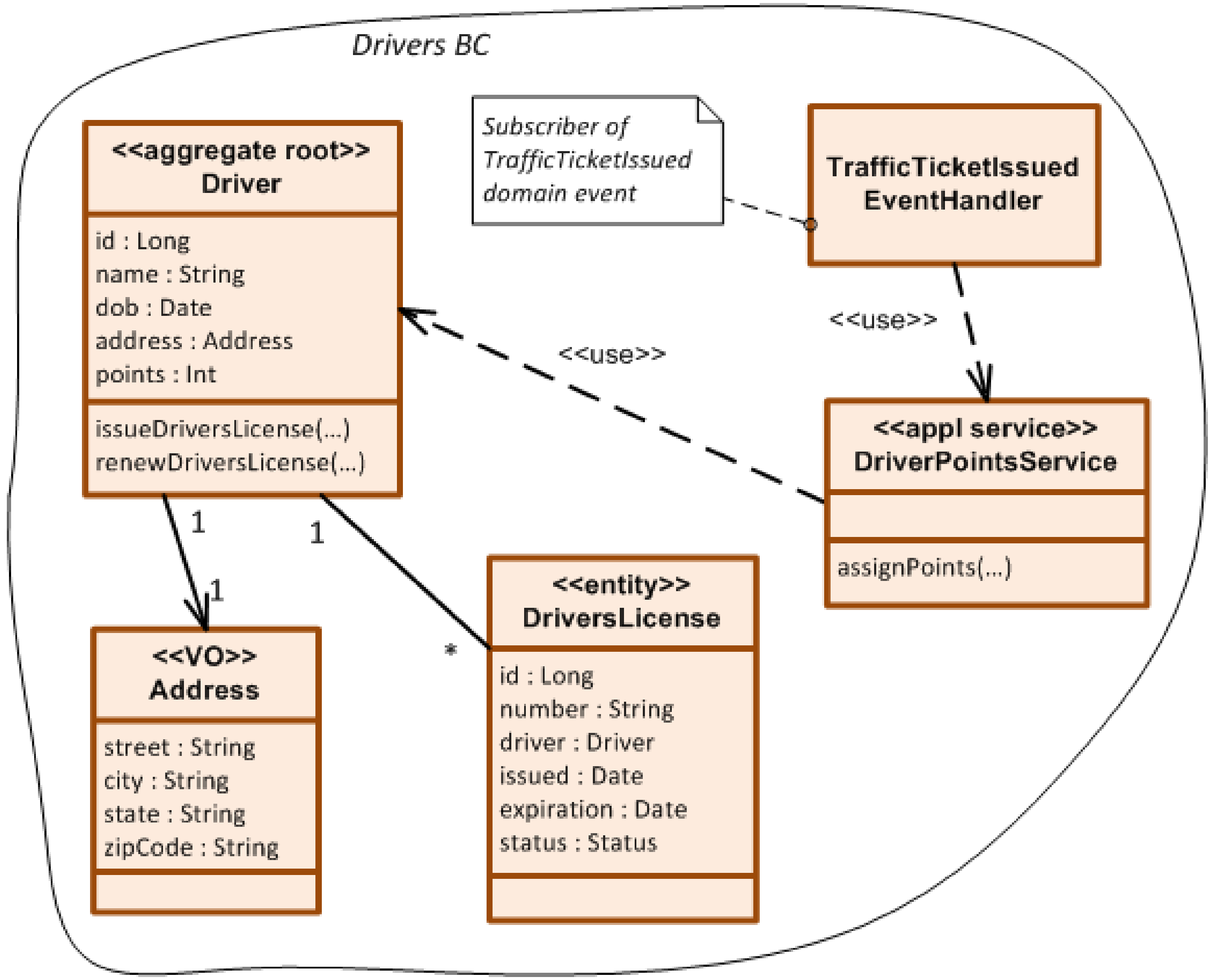
- Each BC → one microservice
- Use domain events for inter microservice communication

Message brokers that support publish-subscribe can be used,

- Kafka
- RabbitMQ
- AWS Kinesis/SNS
- Vert.x
- Akka
- …



Example – scenario 3



Drivers BC

<<aggregate root>>
Driver

id : Long
name : String
dob : Date
address : Address
points : Int

issueDriversLicense(...)
renewDriversLicense(...)

<<VO>>
Address

street : String
city : String
state : String
zipCode : String

<<entity>>
DriversLicense

id : Long
number : String
driver : Driver
issued : Date
expiration : Date
status : Status

Subscriber of TrafficTicketIssued domain event

TrafficTicketIssued EventHandler

<<use>>

<<appl service>>
DriverPointsService

assignPoints(...)

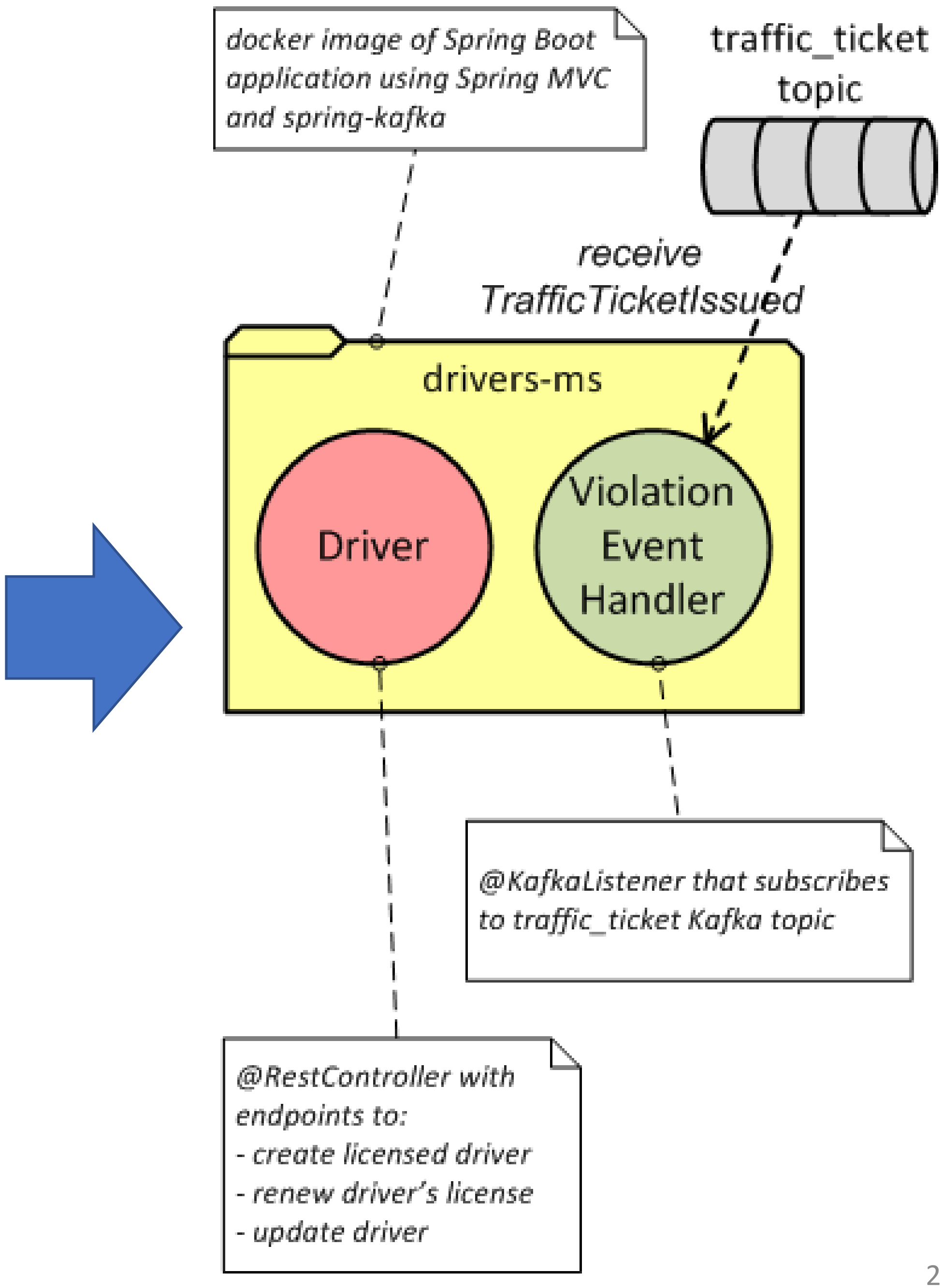
<<use>>

1

1

1

*



docker image of Spring Boot application using Spring MVC and spring-kafka

traffic_ticket topic

receive TrafficTicketIssued

drivers-ms

Driver

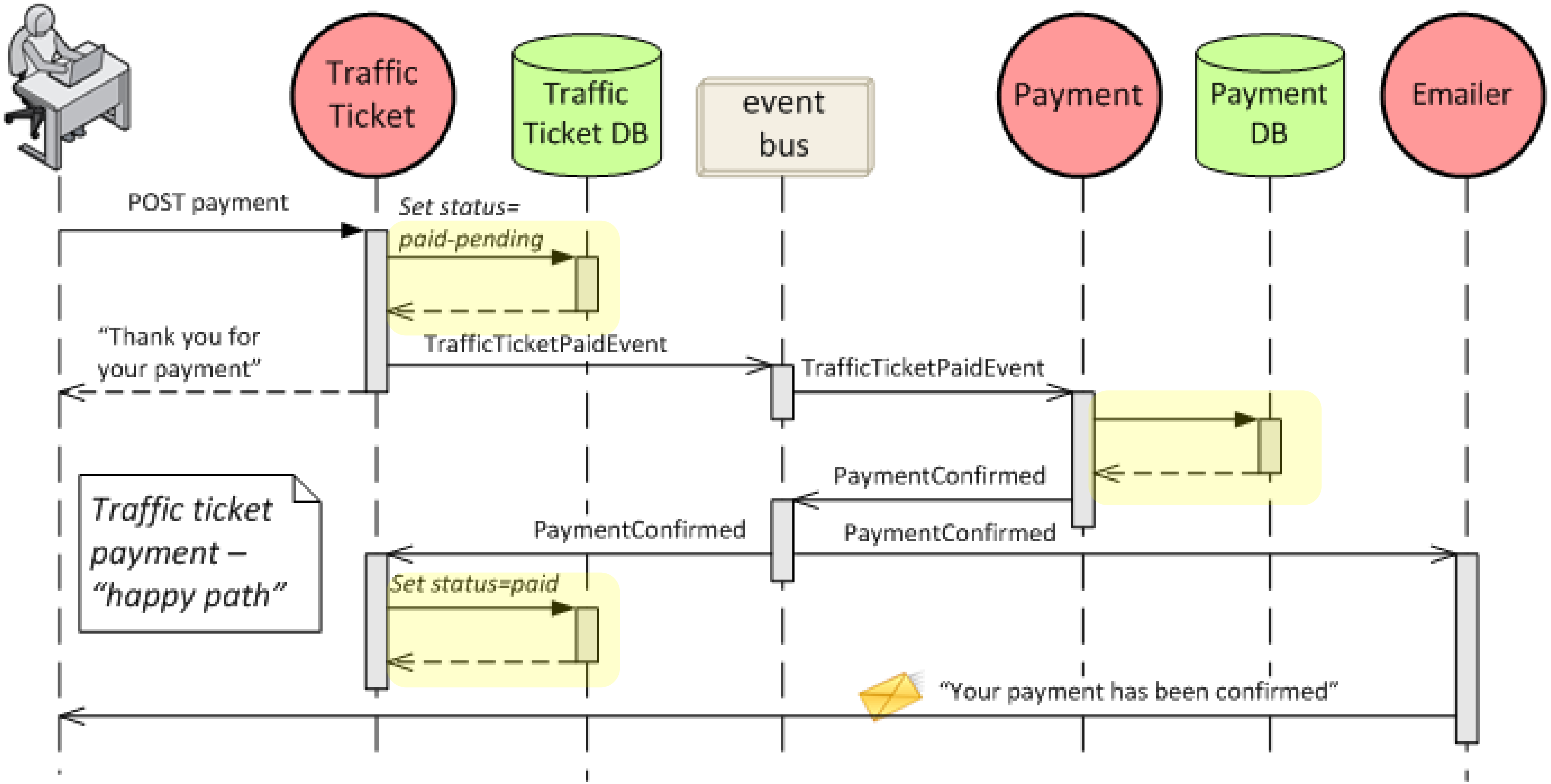
Violation Event Handler

@KafkaListener that subscribes to traffic_ticket Kafka topic

@RestController with endpoints to:
- create licensed driver
- renew driver's license
- update driver

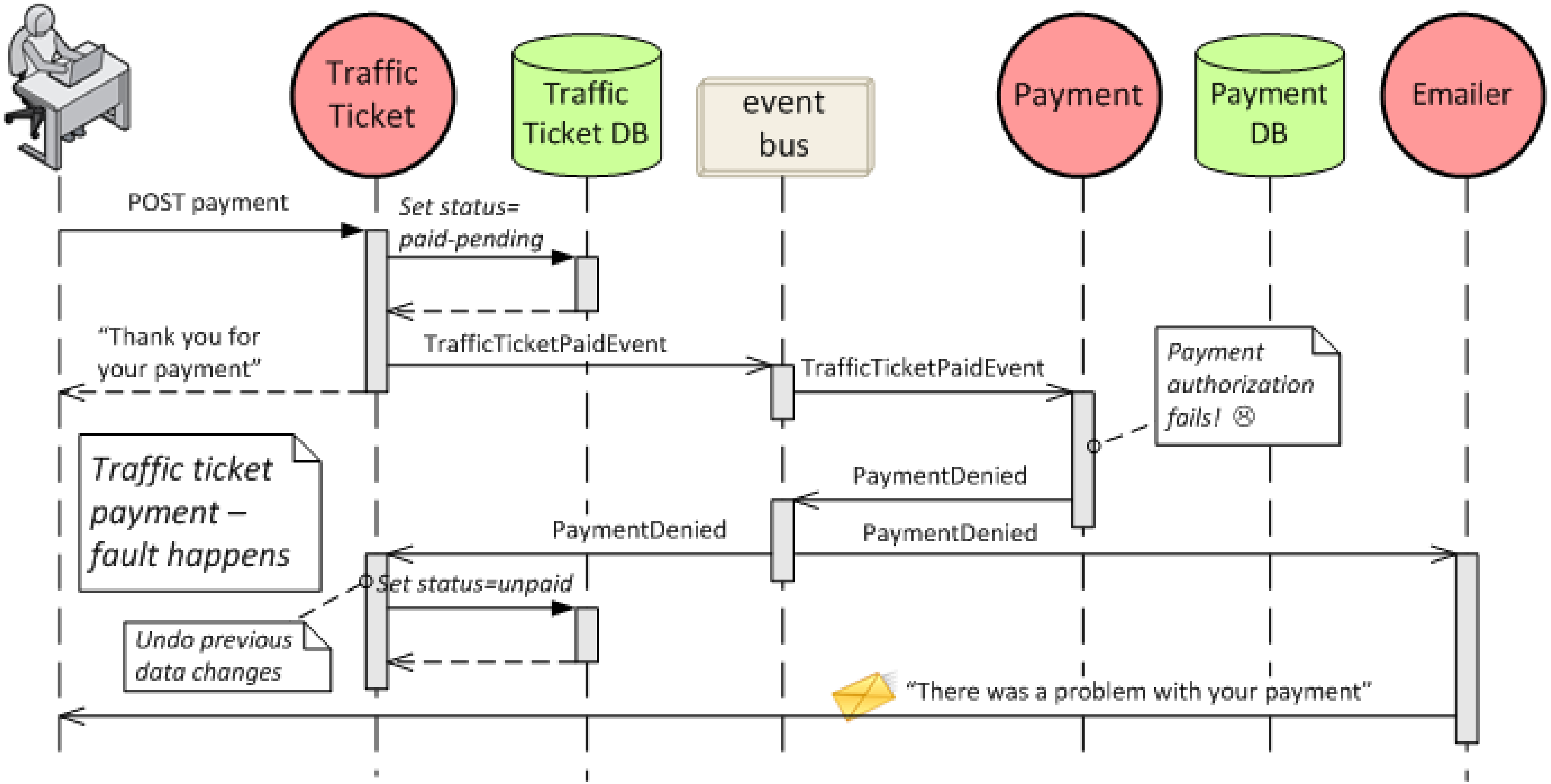
Event-based saga example (1)

Local DB transaction



Traffic ticket payment – "happy path"

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 adding subscribers or events

Scalability and throughput

- Publishers are not blocked
- 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:
  - Processing may happen in parallel and 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*

Availability over consistency is the logical choice in most cases

Takeaways (1)

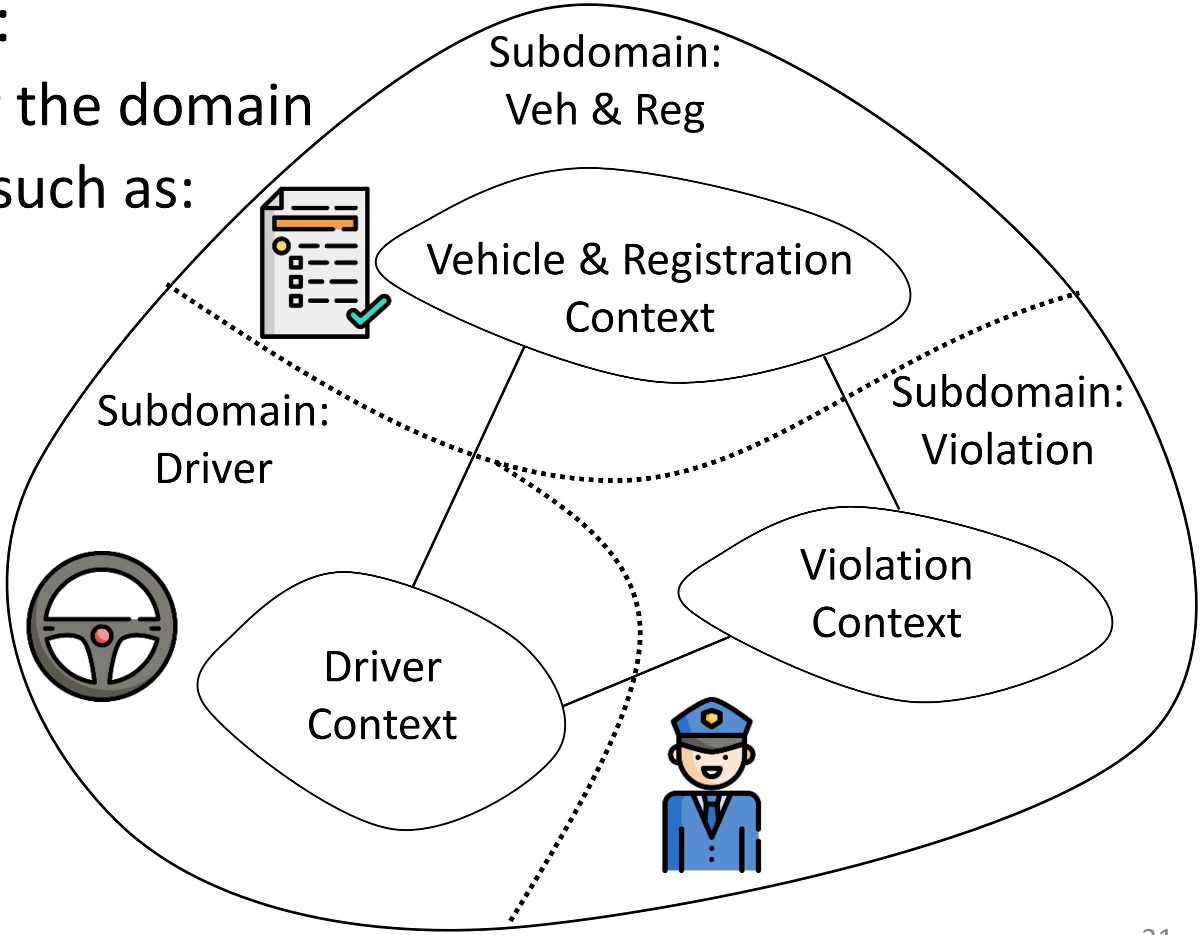
- Domain Driven Design (DDD) can help with defining microservices
- DDD key concepts (for microservice design) are bounded context, aggregate, and entity
- A service (e.g., REST) can have the scope of an aggregate
- A microservice can have the scope of a bounded context
- We can use domain events for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

Takeaways (2)

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

Traffic Ticket Domain



Questions?

Paulo Merson
pmerson@acm.org

Joseph Yoder
joe@refactory.com