Functional Event-Sourcing

Uberto Barbini @ramtop
Asad Manji

```
git clone https://github.com/uberto/okotta-es.git
```

Workshop Goals

In this workshop you will learn:

- Event Sourcing pattern: store changes instead of state
- The Functional Approach: define behavior combining functions operating on immutable data
- CQRS: how having two models give us more flexibility
- Architectural Pattern
- Kotlin language (just a bit)

Timeline

00:00 - 00:30	Presentation
00:30 - 01:00	First exercise: implement a functional Finite State Machine
01:00 - 01:15	Discussion and intro of next exercise
01:15 - 01:45	Second exercise: fix broken tests in command handler
	Third exercise (extended): fix a projection
01:45 - 02:00	Discussion of solutions
02:00 - 02:30	Microservice architecture with CQRS - lessons learned

Exercise 0

Let's write an "hello world" in Kotlin

```kotlin
fun main(){
    println("Hello World")
}
```

Kotlin Playground

val/var

data classes

extension functions

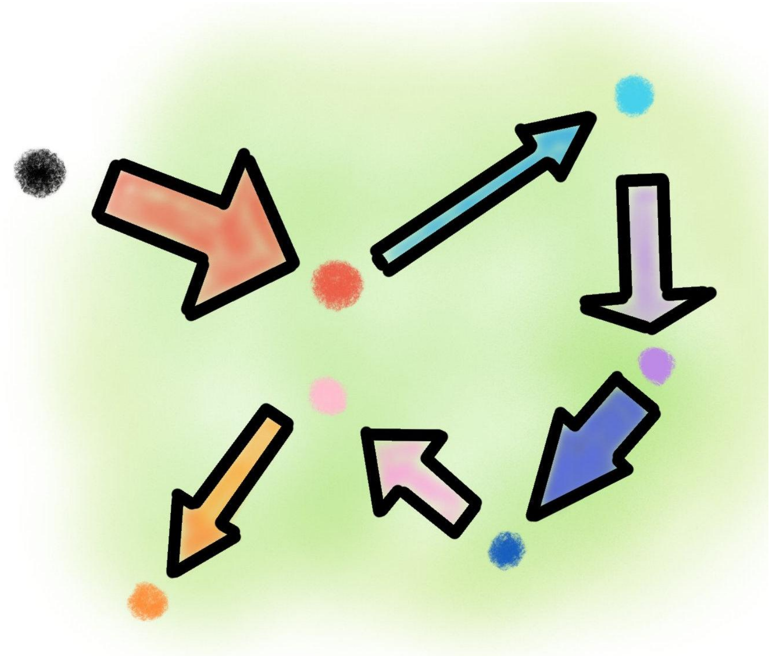
nullable types

sealed classes and when

functions as parameter and results

Event Sourcing and Functional Programming

What functional programming is really about...



Immutability: Referential Transparency

Precise Types: Low cardinality

Purity: Only inputs determine outputs

Totality: No exceptions

Higher Order Functions: functions as values

Event Sourcing and Functional Programming

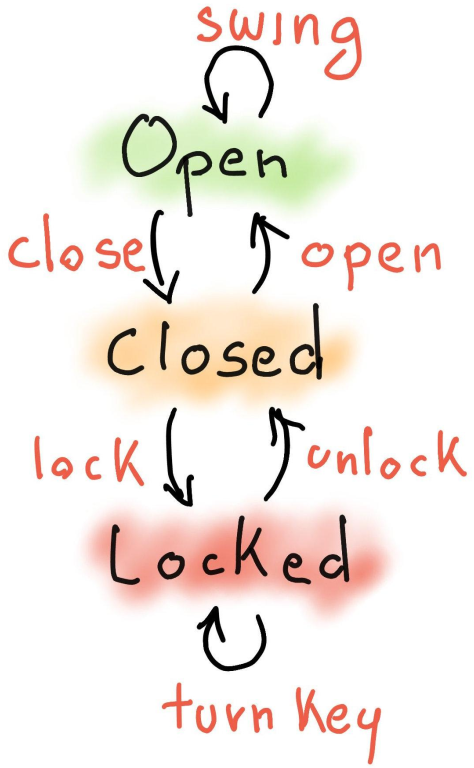
Event Sourcing:

Storing the changes instead of the new state

Events are the “atoms” of System state change

Nothing can change without an event, every event can change only one transactional aggregate

```
State + Event => State
```



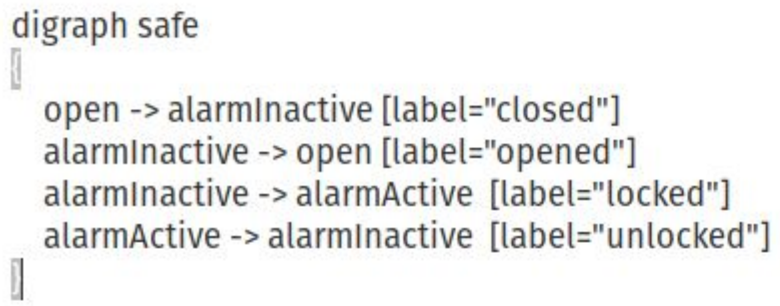
Event Sourcing and Functional Programming

> *State transitions are an important part of our problem space and should be modelled within our domain* -- Greg Young

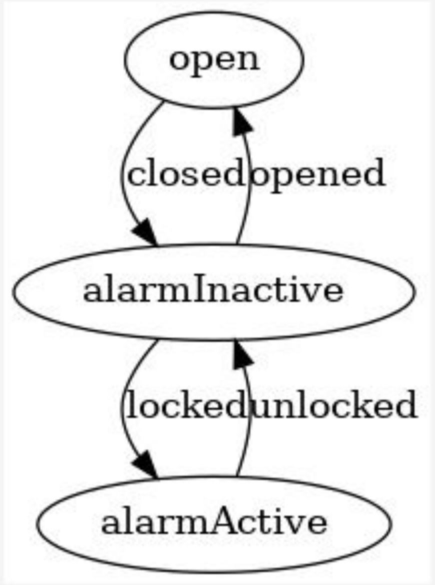
Event Sourcing makes the object-relational impedance mismatch very easy to solve.

We can map what's happening in the domain in a more precise way, and we can decide how to react to that later.

Exercise 1 - Implement a simple state machine



```
digraph safe
{
   open -> alarmInactive [label="closed"]
   alarmInactive -> open [label="opened"]
   alarmInactive -> alarmActive  [label="locked"]
   alarmActive -> alarmInactive  [label="unlocked"]
}
```



Exercise 1 - Implement a simple state machine

Pull the code from

`https://github.com/uberto/okotta-es.git`

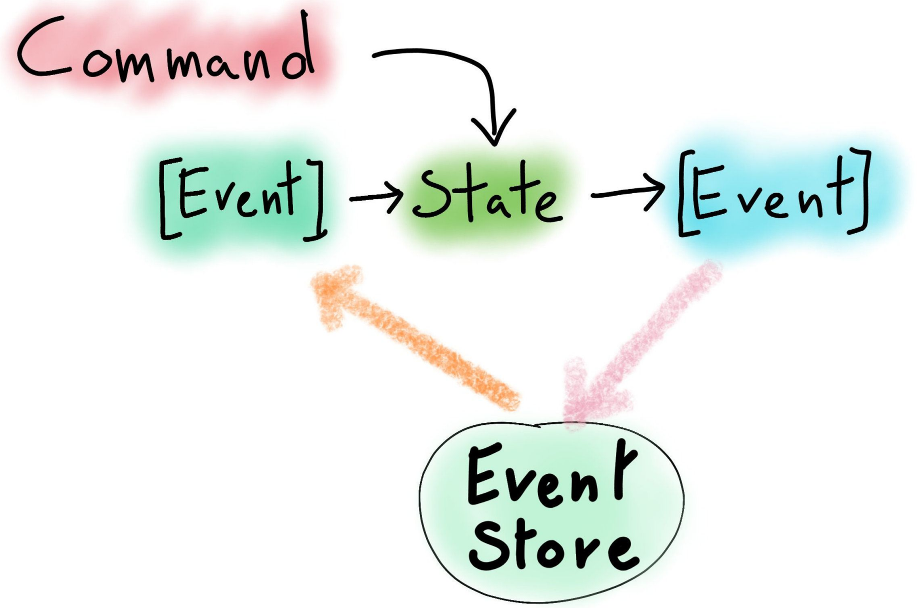
List of exercises

`example-ticketing/Exercises.txt`

```
Exercises 1: fix SafeEventStateTest
```

Commands and Queries

CQRS is not the same as Event Sourcing, they can be implemented separately but they work very well together



Commands and Events

A Command is a request for changing the internal State of the system

A Command can “fail” if the current state of the System is not what the command needs

A successful Command emits a list of Events

Each Command is executed in an atomic context

Queries and Projections

Each query needs a snapshot of the state

Queries typically need different data and denormalization from the domain model.

So we separate the write model (commands) from the read model (queries) in CQRS.

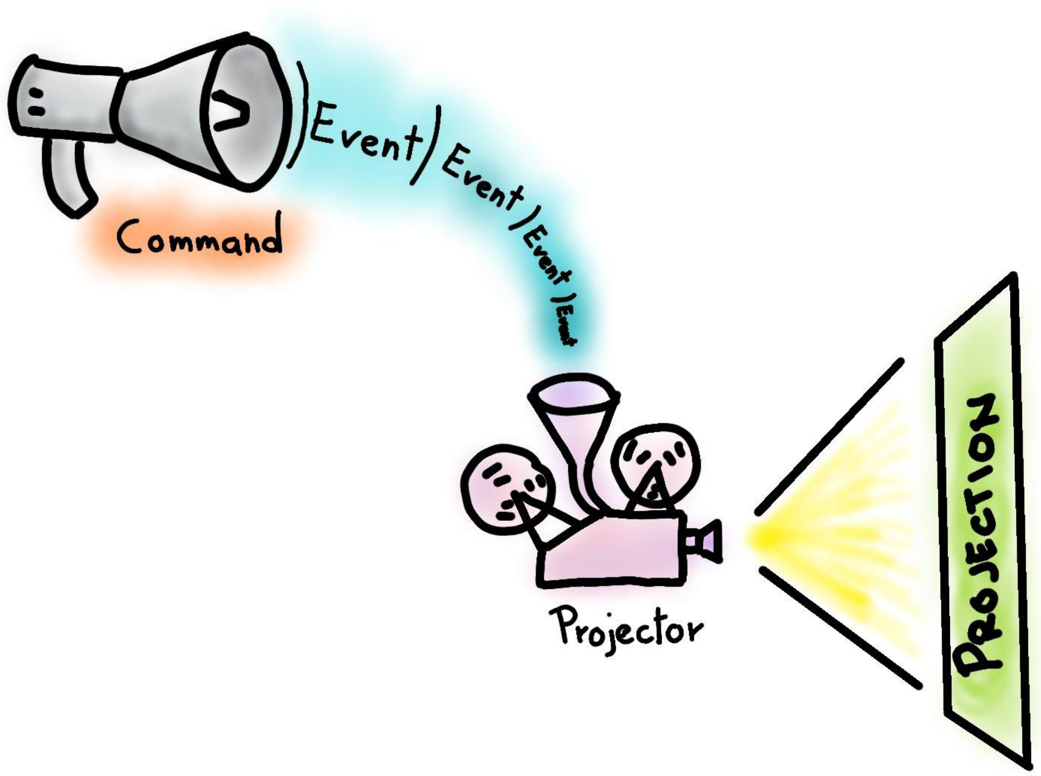
In Event Sourcing, we generate the read model using projections and queries work on projections.

Queries and Projections

Commands emit events.

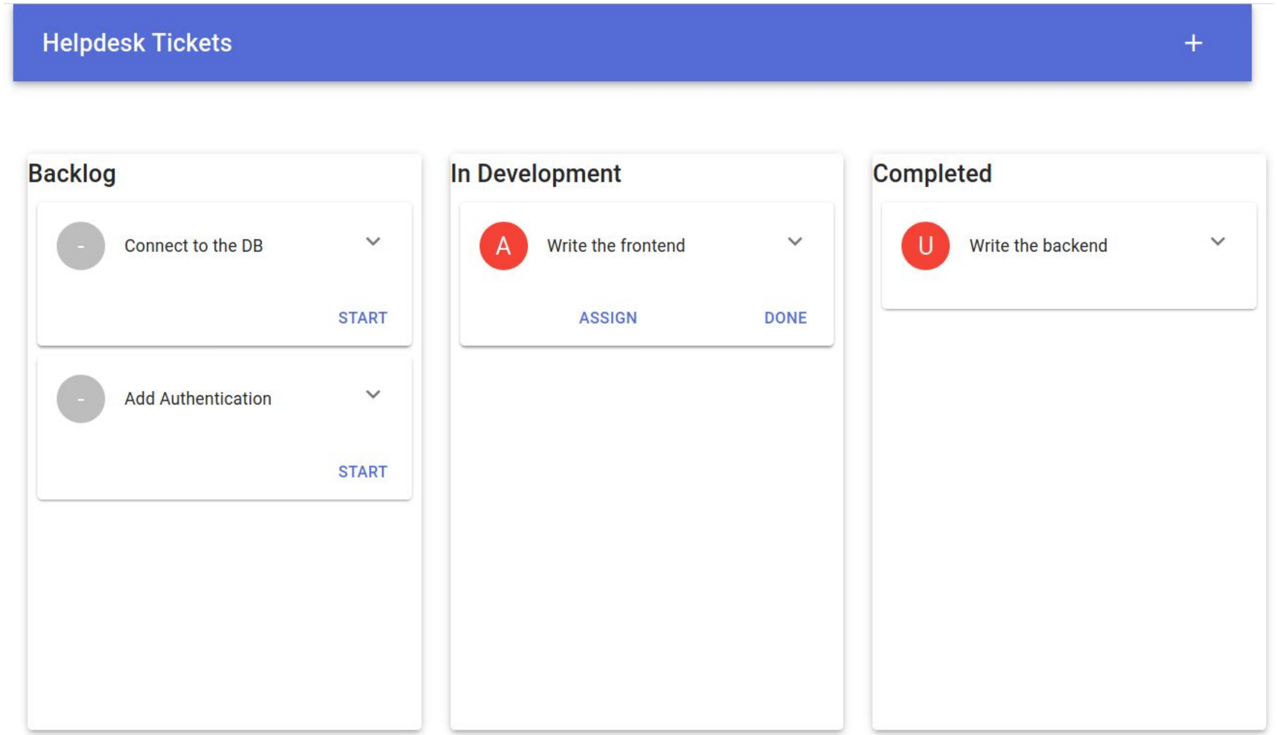
Then we project events to create custom read-only Projections.

Projections are only eventually consistent with the domain.



Helpdesk - a simple service

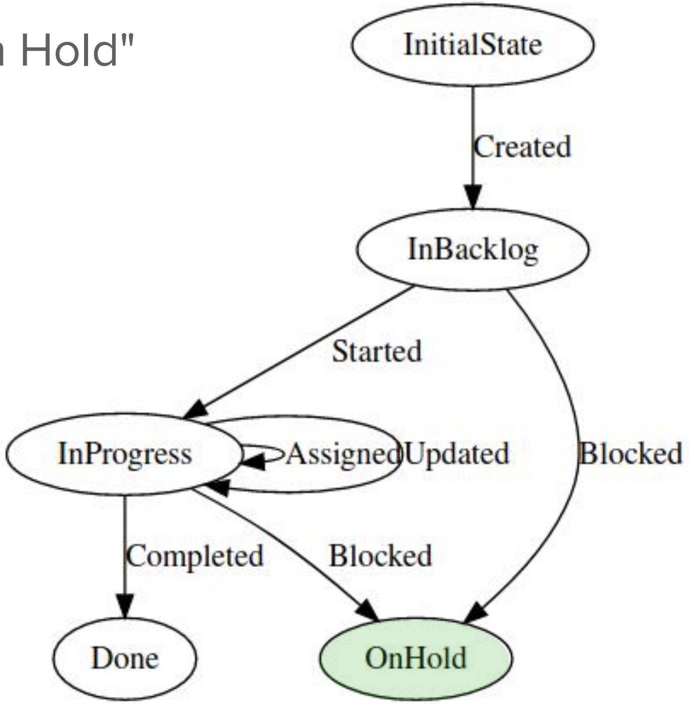
As an example let's see how a simple ticketing service can be build using Event Sourcing and CQRS using functional paradigm.



Live Coding

Let's add a new command to put a ticket "On Hold"

```
digraph ticket {
InitialState -> InBacklog[label="Created"];
InBacklog -> InProgress[label="Started"];
InProgress -> InProgress[label="Assigned"];
InProgress -> InProgress[label="Updated"];
InProgress -> OnHold[label="Blocked"];
InBacklog -> OnHold[label="Blocked"];
InProgress -> Done[label="Completed"];
}
```



Exercise 2: Fixing broken tests

Pull the code from

`https://github.com/uberto/okotta-es`

cd `example-ticketing`

```
../gradlew build
../gradlew run
```

The exercise consists in fixing the broken test on the double assignment

Exercise 3: Fix a projection (extended)

This exercise consists of creating a query for the HelpDesk Projection to count tickets for a given state and connect it to an http endpoint (and optionally connect to UI)

Example HTTP request:

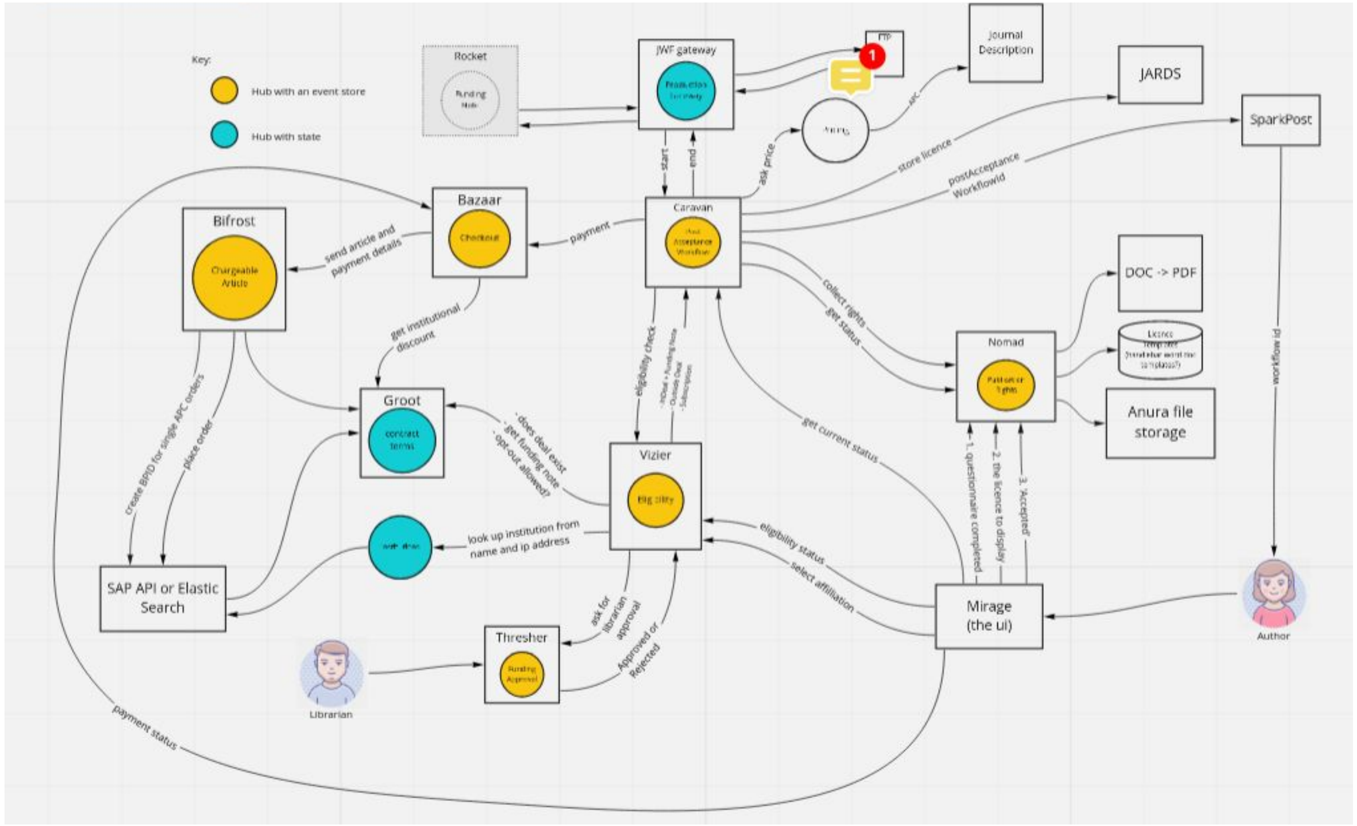
```
curl http://localhost:8080/tickets/count
```

Example HTTP response:

```
{
 “Backlog”: 3,
 “InDevelopment”: 1,
 “Done”: 4
}
```

Discussion of exercise and solutions

Distribuited CQRS Architecture



Kind of Services

- Stateless Services
- Simple State Services
- State Machine ES Services

Communications Between Services

- Avoid too many nested HTTP sync calls
- HTTP as Async Protocol, remember and retry later.
- Triggers for batch jobs

Lessons Learned

- Each Event should work on a single Entity (*aggregate* in DDD lingo). For this reason the Entity of an Event Sourcing model must coincide with a "transactional unit".
- You should start always drawing (and keeping updated) the state diagram, you will go back to that often while writing the code.
- Each event should have a single destination (State) but can have multiple origins.
- States are "situations" better expressed with "...ing" often "waiting for xxx" or "[staying]ready". States are determined by what should happen after, that is their behavior, not what happened before.

Commands and Projections

- Commands shouldn't have data that is already available on the current state.
- CommandHandler should be able to fetch data from outside only when depending on the state
- EntityEvents are not the same as the business events in the external world. CommandHandler is also working as an ACL from external events
- We don't migrate EntityEvents tables in db, either we update the db serialized format on the fly or we create new events.
- We can define the storage format for the events at the last possible moment. We can use in-memory projections also in production until data becomes too heavy and slows application startup.
- To "migrate" a persisted projection, we just rebuild it from scratch using a different database table name.

Takeouts

CQRS

- Pros
  - Simplifies the api design
  - The read model is easy to map to UI
- Cons
  - More work than a CRUD
  - Tricky to adapt it to RESTful api

Questions