Fun.CQRS

Functional and Fun CQRS in Scala

Agenda

- DDD / CQRS / Event Sourcing
- Aggregates / Commands / Events in Scala
- Akka and asynchrounous programming
- Akka Persistence and Event Sourcing

Not About

- Akka Typed
- "Function and Reactive Domain Modeling" - Debasish Ghosh

DDD / CQRS / ES

- Aggregate is a **DDD** concept. It has a *root* and zero or more entities and value objects underneath
- Commands and Events are used in CQRS
- Events can be persisted and replayed

Event Driven / Sourcing

- CQRS is Event Driven, but not necessarily implements Event Sourcing
- in **synchronous** CQRS:
  - tx(Cmd ⇒ Aggregate ⇒ Event ⇒ View)
- in **asynchronous** CQRS:
  - tx(Cmd ⇒ Aggregate ⇒ Event)
  - tx(Event ⇒ View)

Scala And CQRS

On creation

```
Cmd => Event
Event => Aggregate
// therefore we have
Cmd => (Aggregate, Event)
```

Post-creation

```
(Aggregate, Cmd) => Seq[Event]
(Aggregate, Event) => Aggregate
// therefore we have
(Aggregate, Cmd) => (Aggregate, Seq[Event])
```

Async API

On creation

```
Cmd => Future[Event]
Event => Aggregate
// therefore we have
Cmd => Future[(Aggregate, Event)]
```

Post-creation

```
(Aggregate, Cmd) => Future[Seq[Event]]
(Aggregate, Event) => Aggregate
// therefore we have
(Aggregate, Cmd) => Future[(Aggregate, Seq[Event])]
```

Async API - Inconvenience

On creation

```
(cmd:CreateFoo) => Future.successful(FooCreated("foo"))
```

Post-creation

```
(foo:Foo, cmd:ChangeName) => Future.successful(Seq(FooNameChanged("bar")))
```

Sync/Async API - Lift

On creation

```
Cmd => Event
Cmd => Future[Event]
// Yeah!!
Cmd => Throwable
```

Post-creation

```
(Aggregate, Cmd) => Event
(Aggregate, Cmd) => Seq[Event]
(Aggregate, Cmd) => Future[Event]
(Aggregate, Cmd) => Future[Seq[Event]]
// Yeah!!
(Aggregate, Cmd) => Throwable
```

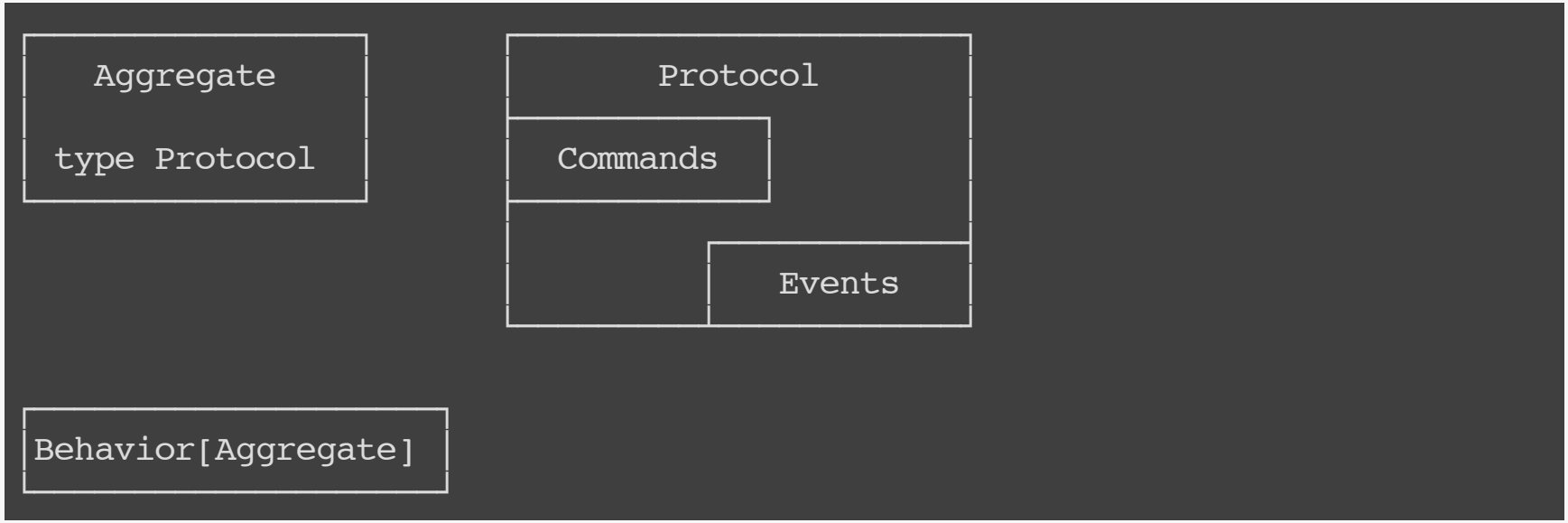
Akka And DDD/CQRS

- Is the Aggregate an Actor?
- Or does it live inside an Actor?
- If it lives inside an Actor, the Actor must know it's hosting an Aggregate
- Akka Persistence for Event Sourcing
- Akka Persistence Query for generating Views (experimental)

Protocol And Behavior

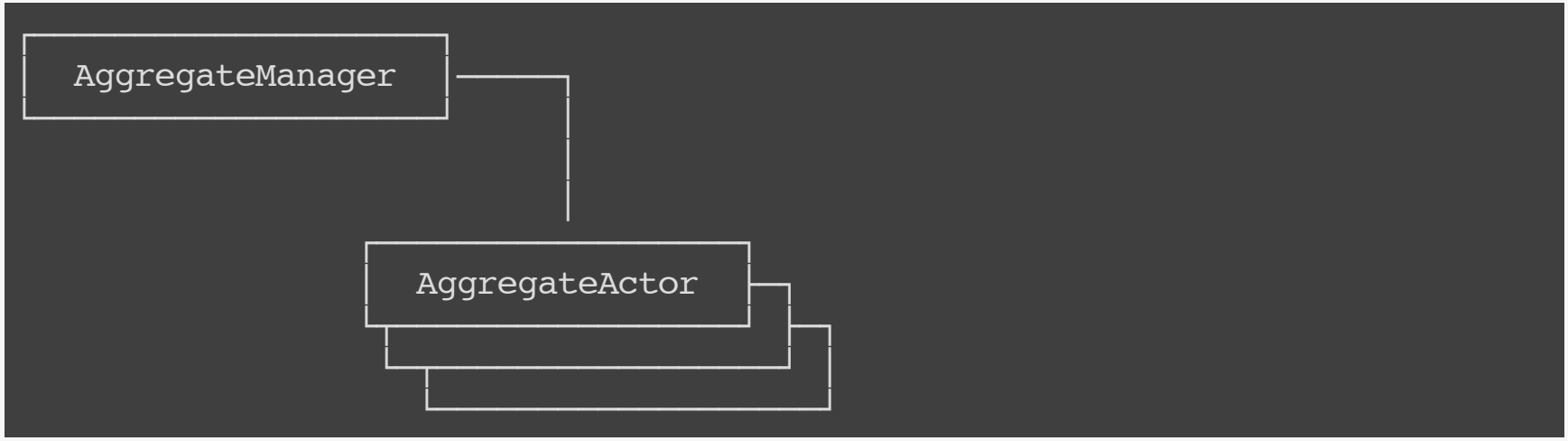
- Protocol is the set of commands and events for a given Aggregate
- Behavior is the implementation
  - Conditions to accept commands
  - Possible failures
  - Modify Aggregate state

Protocol And Behavior



AggregateManager

- Create AggregateActors by id
- Forward messages to right AggregateActors



AggregateActor

- AggregateActor is initialized with Behavior of Foo
- Responsible for Foo lifecycle and events storing

Protocol messages are sent to Actor and applied to Aggregate through its Behavior

Projections

- We read from the Event Store to produce Views
- Akka Persistence Query new experimental module
- Produces a Reactive Stream source with the selected events

Shop Aggregates

- Customer
  - just info from a customer
- Product
  - create, change name and change price
- Order (references Customer and Products by identifier)
  - created for a customer
  - add / remove products
  - execute / cancel

Shop Projections

- Customer Aggregate event → *CustomerView*
- Product Aggregate event → *ProductView*
- Customer, Product and Order events → *OrderView*

Problem With OrderViewProjection

What to do with Events from *Customer* and *Product*?

They will probably arrive before the first order is created.
Should we query the *CustomerViewRepo* and *ProductViewReop* whenever we need more info?

Will they reflect the expected state?

Solutions

1. Have one single event stream and one consumer
    - can be a serious bottleneck
2. Implement specific logic for each single event for each view
    - can lead to increasing complexity
3. Copy data by reusing existing projections, but saving in another Repository
    - need for more storage, but simpler and reusable code
    - Demo project uses that approach (check it)
4. Synchronous views, at least the main views
    - Response time penalty, but better user feedback

Thank You!

http://twitter.com/@renatocaval

https://github.com/strongtyped/fun-cqrs