IWA

EVENT SOURCING

AND SOME OTHER STUFFS.

EVENT SOURCING

+ DOMAIN-DRIVEN DESIGN

WHAT IS EVENT SOURCING

SOURCE OF TRUTH IS EVENTS.

Events are all records that matters.

EVENT SOURCING IS ANCIENT TECH.

ACCOUNTING AND BANK

Action	Amount
AccountCreated	0
AmountDeposited	150
AmountDeposited	70
AmountWithdrawn	200
....	....

Your account balance is the result of all your transaction.

VERSION CONTROL SYSTEM

```
$ git log
....
....
```

Your current source code is the result of addition/deletions in all commits.

DATABASE

Write-ahead Log is just record of changes.

EXAMPLE: ORDER SYSTEM.

TRADITIONAL DATABASE.

```
CREATE TABLE OrderHeader (..);
CREATE TABLE OrderLineItem (..);
```

id	CustomerName	Address	Requested
1234	John Smith	blah ...	false

id	OrderId	Product	Quantity	Amount
111	1234	Burger	1	15
112	1234	Water	3	30

EVENT STORE.

STREAMNAME: "ORDER-1234"

EventNumber	EventType	PayLoad
1	OrderCreated	{ id: 1234 }
2	OrderShippingUpdated	{ CustomerName: 'John Smith', Address: '...'}
3	OrderItemAdded	{ Product: 'Burger', Quantity: 1, Amount: 15 }
4	OrderItemAdded	{ Product: 'Burger', Quantity: 3, Amount: 30 }

LET'S REMOVE AN ITEM AND SEND THE ORDER REQUEST.

TRADITIONAL DATABASE.

```
UPDATE OrderLineItem SET Amount = Amount-1 WHERE ...;
UPDATE OrderHeader SET Requested = true WHERE ...;
```

id	CustomerName	Address	Requested
1234	John Smith	blah ...	true

id	OrderId	Product	Quantity	Amount
111	1234	Burger	1	15
112	1234	Water	2	20

EVENT STORE.

STREAMNAME: "ORDER-1234"

EventNumber	EventType	PayLoad
1	OrderCreated	{ id: 1234 }
2	OrderShippingUpdated	{ CustomerName: 'John Smith', Address: '...'}
3	OrderItemAdded	{ Product: 'Burger', Quantity: 1, Amount: 15 }
4	OrderItemAdded	{ Product: 'Burger', Quantity: 3, Amount: 30 }
5	OrderItemRemoved	{ Product: 'Burger', Quantity: 1, Amount: 10 }
6	OrderRequested	{ }

1. ONLY EVENTS ARE PERSISTED, NOT STATE.

2. PAST EVENTS ARE IMMUTABLE.

HOW TO WRITE THE CONTROLLER

```
DoRemoveItem(id: 1234,Product: 'Water')
```

1. Load all events from beginning of time [0].
2. Sequentially apply each event.
3. Create resulting Events (`OrderItemRemoved`)
4. Performed the action by Persisting Events.
5. (Optionally) Apply the created events. Return latest state.

IN CODE

```
def remove_item()
  order = Order.load("Order-#{params[:id]}")
  saveToStream("Order-#{params[:id]}",
    'OrderItemRemoved',
    {
      Product: product,
      Quantity: quantity,
      Amount: amount,
    }
  )
end
```

SOME CODE

```
class Order < Aggregate
  def load(streamName)
    events = loadAllFromStream(streamName)
    @state = {}
    events.each do |ev|
      @state = apply(@state, ev)
    end
  end
end
```

```
class Order < Aggregate
  def apply(state, ev)
    switch ev.type
      case  'OrderCreated':
        return { requested: false, items: [] }
      case 'OrderItemAdded':
        item = state[:items][ev.payload[:Product]] ||= {
          Quantity: 0,
          Amount: 0
        }
        item[:Quantity] += ev.payload[:Quantity]
        item[:Amount] += ev.payload[:Amount]
      case 'OrderItemRemoved':
        ...
    end
  end
end
```

AGGREGATE

- A Transactional Consitency Boundary of a business logic.
- Not an Entity, Graph of Entities.
- An Aggregate is contained in one Event Stream.

REBUILDING STATE.

```
[1] OrderCreated(id: 1234)

Order = {
  id: 1234,
  CustomerName: null,
  Address: null,
  Items: [],
  Total: 0,
}
```

REBUILDING STATE.

```
[2] OrderShippingUpdated(
  CustomerName: 'John Smith',
  Address: 'blah...'
)


Order = {
  id: 1234,
  CustomerName: 'John Smith',
  Address: 'blah...',
  Items: [],
  Total: 0,
}
```

REBUILDING STATE.

```
[3] OrderItemAdded(Product: 'Burger', Quantity: 1, Amount: 15)

Order = {
  id: 1234,
  CustomerName: 'John Smith',
  Address: 'blah...',
  Items: [ {...} ],
  Total: 15,
}
```

REBUILDING STATE.

```
[4] OrderItemAdded(Product: 'Water', Quantity: 3, Amount: 30)

Order = {
  id: 1234,
  CustomerName: 'John Smith',
  Address: 'blah...',
  Items: [ {...}, {...} ],
  Total: 45,
}
```

LOAD PERFORMANCE?

- Each aggregate is "usually" small.
- You can always create snapshot (Closing the book).

WRITE PERFORMANCE?

- Append is ***FAST***.
- Transactional boundary per each stream.
- Optimistic Locking (Save with ExpectedVersion)

READ PERFORMANCE?

- Transform data to your flat denormalized form.
- Feed data to your search server
- CQRS

PRO: 1. AUDIT AND LOGGING.

- Audit First System.
- Log is now "free".
- State cannot be there without Log.

PRO: 2. GREAT FOR ANALYTICS.

- You can not "go back in time" to collect data.
- You can "replay" the events.
- Temporal query (What was the state of the system at 2018/10/02?)

CURRENT STATE IS DERIVATIVE OF EVENTS.

YOU LOSE DATA AS SOON AS YOU STORE STATE.

CAN YOU DECIDE WHAT TO LOSE?

PRO 3: FIT FOR DISTRIBUTED SYSTEM.

- DDD and Aggregate is good design for microservice.
- Event Stream is simple to use/integrate.

PRO 4: FORCE BUSINESS ANALYSIS.

- What the business does is more important than how to design data structure.

CON: FORCE BUSINESS ANALYSIS.

- You can't design the system until you know what happens.
- "Sprint 1: I want to cook something. Let's boil the eggs.".

HANDLING CONSITENCY.

- Aggregate: Boundary of transaction consistency.
- Read-only/Query system can live with eventual consistency.

HANDLING GDPR.

- Create Separated Stream for Sensitive Data
- Delete those stream

... OR you can replay and filter out Identification data.

IMPLEMENTATIONS.

- Event Store: https://eventstore.org/
- Rails Event Store: https://railseventstore.org/
- Roll your own.

THAT'S IT.

Q&A AND THANK YOU.