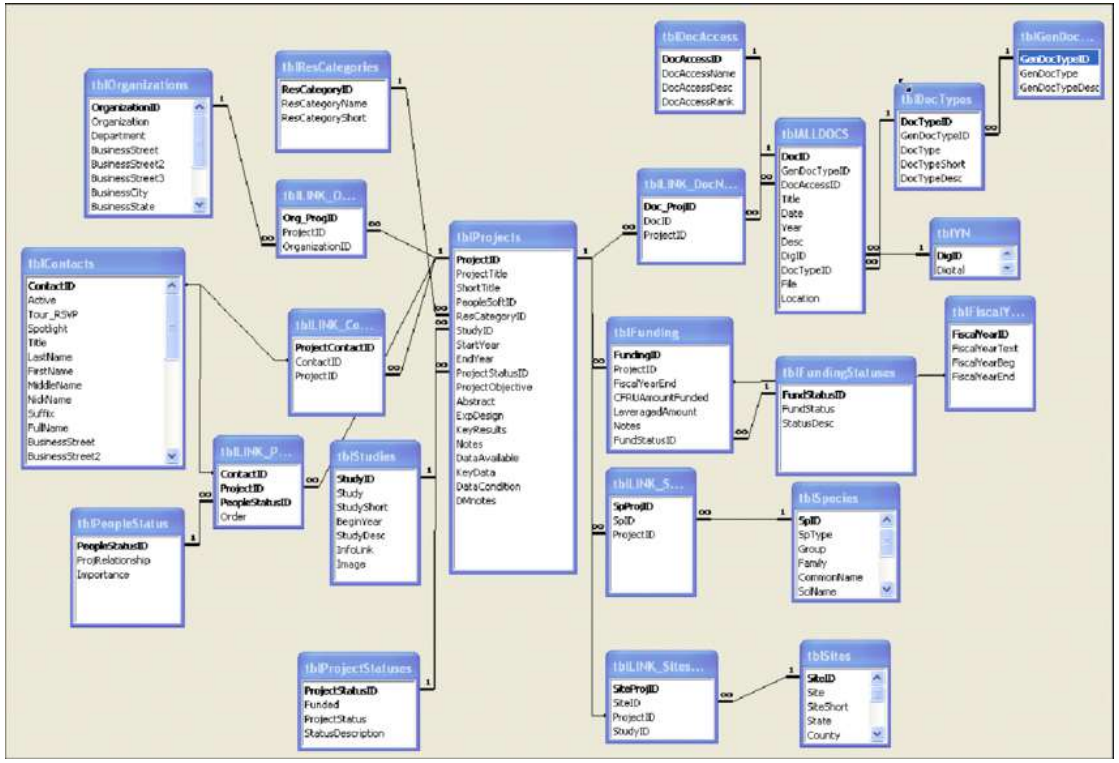
Event Sourcing and CQRS for dummies

Kim Van Renterghem
Kim.vanrenterghem@gmail.com

Traditional systems

Use a central db

With a common state



Whenever we change something we lose the previous state

Results | Messages

	LocationID	Name	CostRate	Availability	ModifiedDate
1	1	Tool Crib	0.00	0.00	2002-06-01 00:00:00.000
2	2	Sheet Metal Racks	0.00	0.00	2002-06-01 00:00:00.000
3	3	Paint Shop	0.00	0.00	2002-06-01 00:00:00.000
4	4	Paint Storage	0.00	0.00	2002-06-01 00:00:00.000
5	5	Metal Storage	0.00	0.00	2002-06-01 00:00:00.000
6	6	Miscellaneous Storage	0.00	0.00	2002-06-01 00:00:00.000
7	7	Finished Goods Storage	0.00	0.00	2002-06-01 00:00:00.000
8	10	Frame Forming	22.50	96.00	2002-06-01 00:00:00.000
9	20	Frame Welding	25.00	108.00	2002-06-01 00:00:00.000
10	30	Debur and Polish	14.50	120.00	2002-06-01 00:00:00.000
11	40	Paint	15.75	120.00	2002-06-01 00:00:00.000
12	45	Specialized Paint	18.00	80.00	2002-06-01 00:00:00.000
13	50	Subassembly	12.25	120.00	2002-06-01 00:00:00.000
14	60	Final Assembly	12.25	120.00	2002-06-01 00:00:00.000

When you change tracking/detection becomes difficult

Bugs are causing an inconsistent state

=> by correcting you often create a vicious circle

When there are Audits, it is impossible to prove that the state is correct.

There are no guarantees that every change is logged because the updates and the audit log do not happen in the same transaction.

Logs

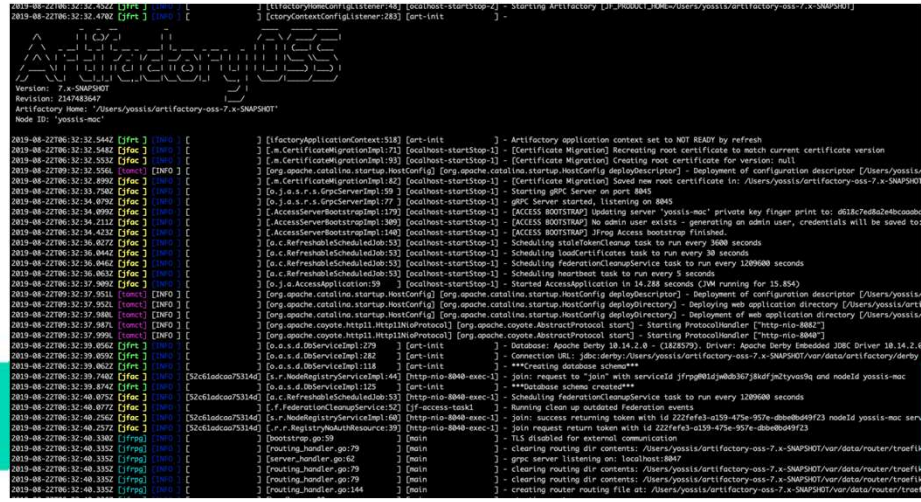
To the rescue!

Logs

- Shows when and what changes

Logs

- Logs are difficult to handle because they do not look at the changes but at the consequences of actions.



Logs

- They are also not about an entity but about all transactions in a domain. Often even about steps we take within a process.

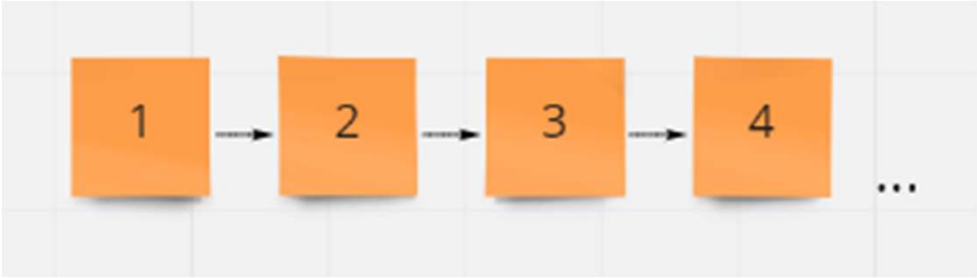
Logs

- They show when something changes.
  => But not how or why.

Logs

- They do not have audit trails. These are often added. But in turn are also not reliable.

What if we now used our logs as "the only source of truth“



What if we now used our logs as "the only source of truth“

Let the logs be about the transitions

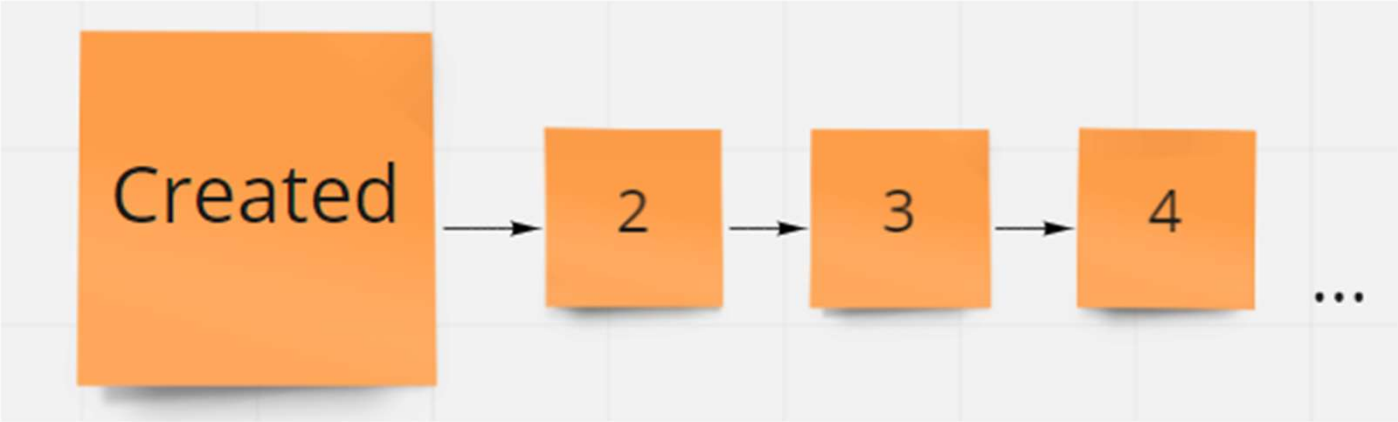
for example:

- Table reserved
- Two martinis ordered
- Martini's delivered
- A steak with fries ordered
- ...
- Bill paid
- Table left

What if we now used our logs as "the only source of truth“

You always start with a created event

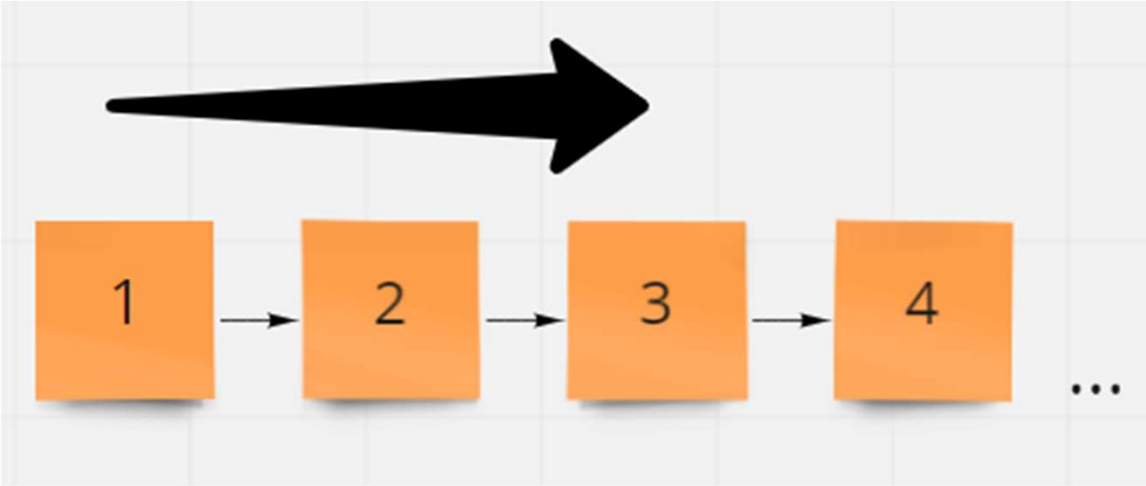
- Table reserved



What if we now used our logs as "the only source of truth“

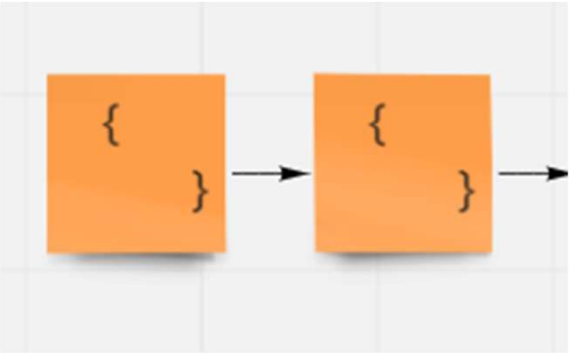
- Should be add only
  - 10l red wine ordered
  - 10l red wine order cancelled
  - 1l red wine ordered

no removing or adjusting!



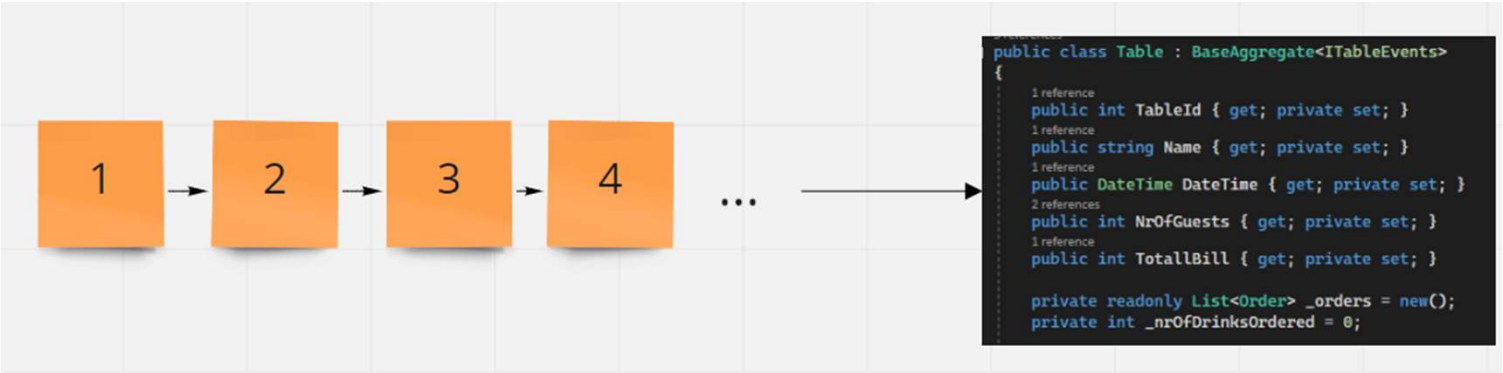
What if we now used our logs as "the only source of truth“

- use structured data such as a document store or json document



What if we now used our logs as "the only source of truth“

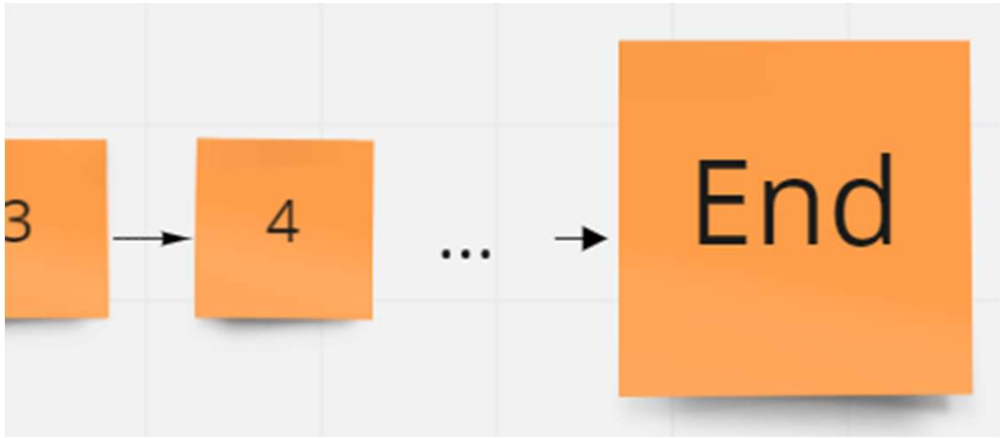
- Now you have time in your model and the sum of all transitions is the current state



What if we now used our logs as "the only source of truth“

When stream ends, mark it with an event

- Bill paid
- Table left

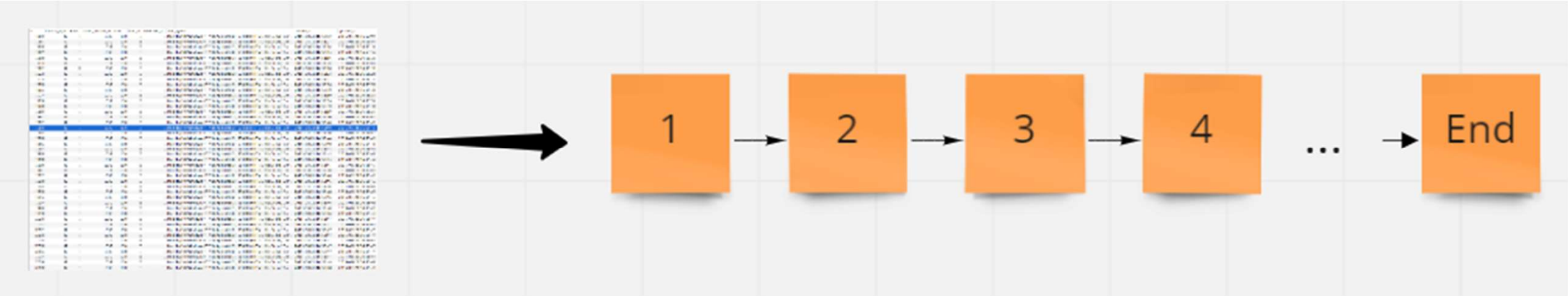


What if we now used our logs as "the only source of truth“

- Domain events are always in the past

This is what we call “Event sourcing”

Instead of a table with rows for entities.
Use a stream for each aggregate.



This is what we call “Event sourcing”

Domain events must always be playable.

It should always be possible to replay your domain event.
You obtain this by having it played immediately after you have completed your command. Then save the event to the db(stream).

Example in db(stream)

Where to begin?

Event Storming

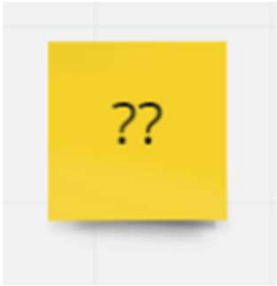
To explore your domain.

Event Storming

Because your events have yet to be determined and discovered.

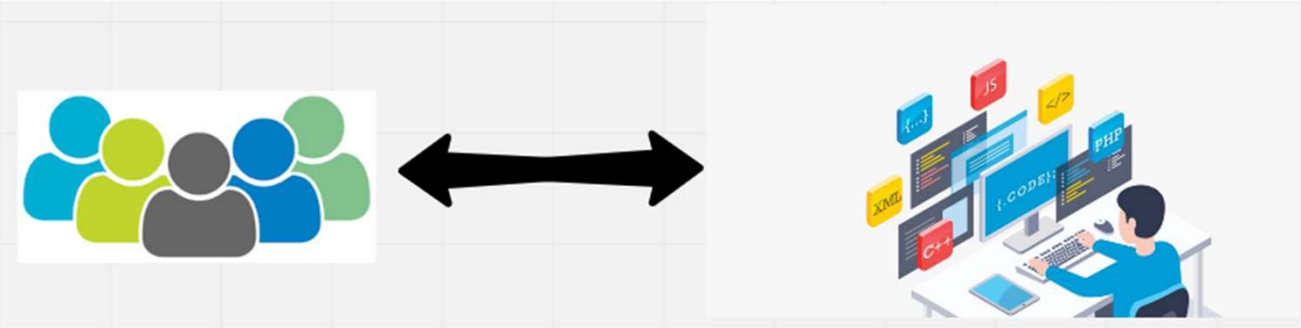
Event Storming

Determine your aggregate.
=> your root object within your bounded context



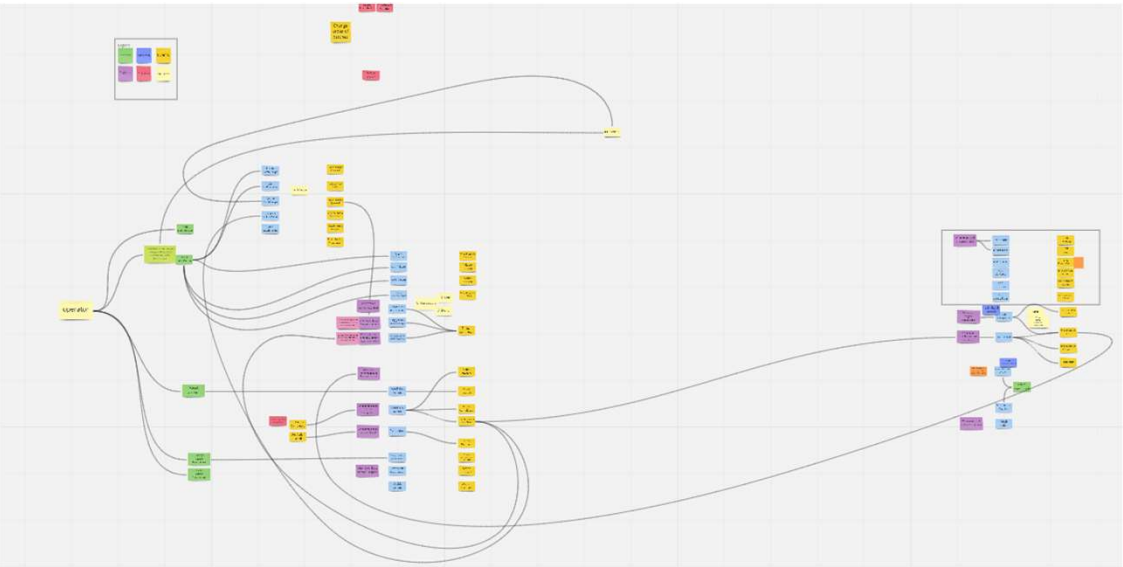
Event Storming

- Bridge between your developers and stakeholders.
- Share the same knowledge and language.
- Different stakeholders also have different needs and insights.
  =>This can result into different bounded contexts.



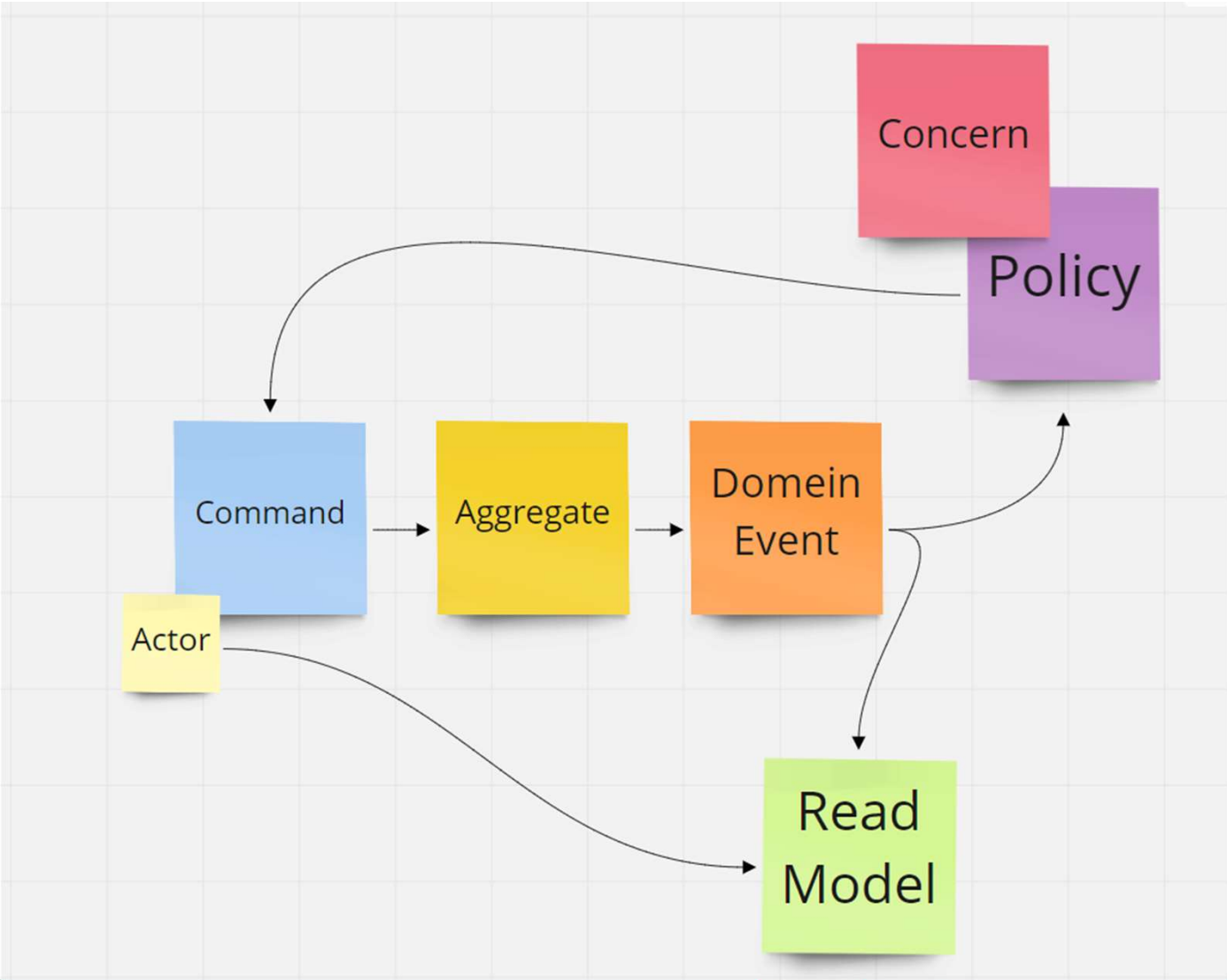
Event Storming

To better understand your domain.
This way you gain insights into the processes and phases of your application



Event Storming

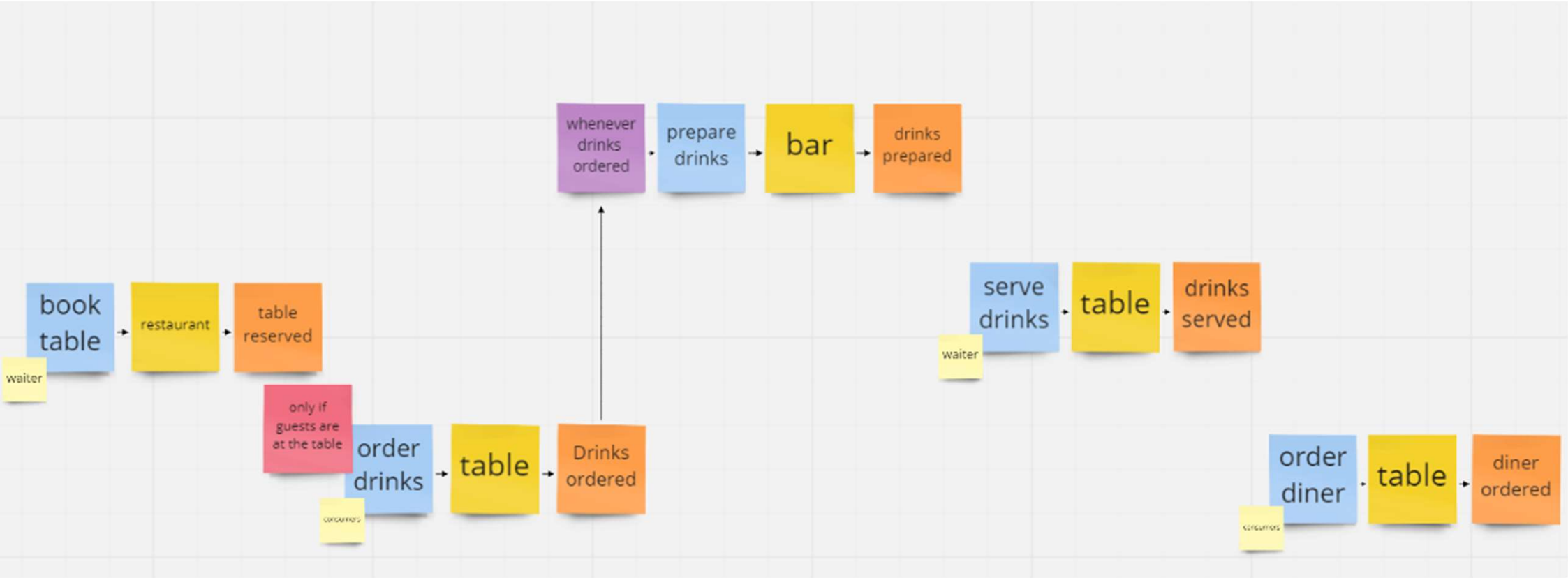
Process



Event Storming

- Is about interactions with the users/business.
- Domain Exploration.
- Using the language of the business.
- Interactions over time.
- System transformations.
  - You cannot pay before you have ordered.

Example


whenever drinks ordered
prepare drinks
bar
drinks prepared
book table
waiter
restaurant
table reserved
only if guests are at the table
order drinks
table
Drinks ordered
serve drinks
waiter
table
drinks served
order diner
table
diner ordered

Event Storming

- Remains a continuous process with every change.
- It keeps improving your process from your model.

Event Storming

- Let your code also reflect your model. Use the same names and format. This makes it easier to see the whole.

Constructors and commands

```
1 reference | 1/1 passing
public Table(int tableId, DateTime dateTime, string name, int nrOfGuests)
{
    RegisterHandlers();
    PublishNewEvent(new TableReserved // start event
    {
        TableId = tableId,
        Name = name,
        DateTime = dateTime,
        NrOfGuests = nrOfGuests
    });
}

2 references | 2/2 passing
public Table(IEnumerable<ITableEvents> events)
{
    RegisterHandlers();

    //play all the evetns
    events
        .ToList()
        .ForEach(PlayEvent);
}

//command
2 references | 2/2 passing
public void OrderDrinks(Order order)
{
    if (_nrOfDrinksOrdered >= 2 * NrOfGuests)
        throw new Exception("Too many drinks :-)");

    PublishNewEvent(new DrinksOrdered()
    {
        Order = order
    });
}
```

Events

```
namespace EventSourcingDemo
{
    6 references
    public class TableReserved : ITableEvents
    {
        5 references | 3/3 passing
        public int TableId { get; init; }
        5 references | 3/3 passing
        public DateTime DateTime { get; init; }
        5 references | 3/3 passing
        public string Name { get; init; }
        5 references | 3/3 passing
        public int NrOfGuests { get; init; }
    }
}
```

```
public class DrinksOrdered : ITableEvents
{
    7 references | 2/2 passing
    public Order Order { get; init; }
}
```

```
public class Order
{
    5 references | 2/2 passing
    public string ProductName { get; init; }
    5 references | 2/2 passing
    public int ProductId { get; init; }
    6 references | 2/2 passing
    public int Quantity { get; init; }
    6 references | 2/2 passing
    public int Price { get; init; }
    0 references
    public string Comment { get; init; }
}
```

Register and Handlers

```
// register all events handlers
2 references
private void RegisterHandlers()
{
    RegisterHandler<TableReserved>(TableReservedHandler);
    RegisterHandler<DrinksOrdered>(DrinksOrderedHandler);
}

//play the event
1 reference
private void DrinksOrderedHandler(DrinksOrdered @event)
{
    _orders.Add(@event.Order);
    _nrOfDrinksOrdered += @event.Order.Quantity;
    TotallBill += @event.Order.Price;
}

1 reference
private void TableReservedHandler(TableReserved @event)
{
    TableId = @event.TableId;
    Name = @event.Name;
    DateTime = @event.DateTime;
    NrOfGuests = @event.NrOfGuests;
}
```

In practice?

For each command, the aggregate gets in the correct state by loading all events from the current stream.

Then execute your commands and add your events to the current stream.

```
//get aggregate in repository
var aggregate = new Table(events);

//execute command
aggregate.OrderDrinks(new Order...);

//commit events in repository
await aggregate.PlayAllEvents(async e =>...);
```

Testing?

```
[Fact]
| 0 references
public async void Should_Emit_TableReservedEvent_On_Create()
{
    var aggregate = new Table(6, new DateTime(2021, 12, 31), "kim vr", 2);
    var events = new List<ITableEvents>();
    await aggregate.PlayAllEvents(async e =>
    {//because normally this is an async write to the db
        await Task.FromResult(0);
        events.Add(e);
    });

    events.Should().ContainSingle();

    events.First()
        .Should().Equals(new TableReserved
        {
            TableId = 6,
            DateTime = new System.DateTime(2021, 12, 31),
            Name = "kim vr",
            NrOfGuests = 2
        });
}
```

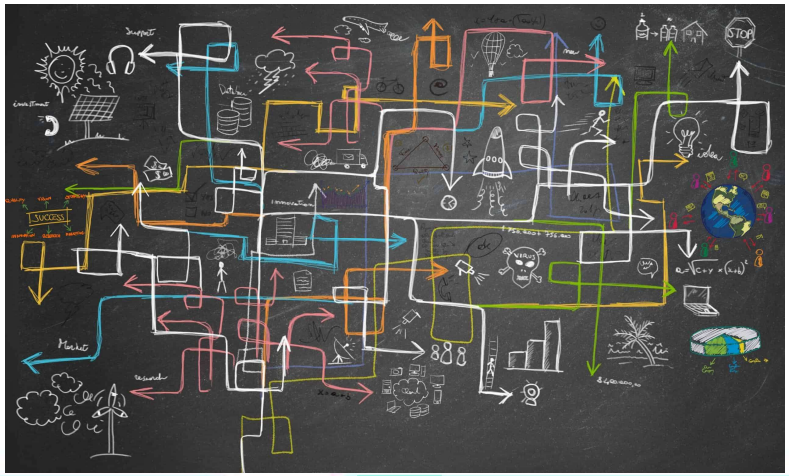
```
[Fact]
| 0 references
public async void Should_be_able_to_order_a_drink()
{
    var events = new ITableEvents[]
    {
        new TableReserved
        {
            TableId = 6,
            DateTime = new DateTime(2021, 12, 31),
            Name = "kim vr",
            NrOfGuests = 2
        }
    };

    var aggregate = new Table(events);
    aggregate.OrderDrinks(new Order
    {
        ProductId = 6,
        Price = 6,
        ProductName = "Martini",
        Quantity = 2
    });

    var newEvents = new List<ITableEvents>();
    await aggregate.PlayAllEvents(async e =>
    {
        await Task.FromResult(0);
        newEvents.Add(e);
    });

    newEvents.Should().ContainSingle();
    newEvents.First().Should().Equals(new DrinksOrdered
    {
        Order = new Order
        {
            ProductId = 6,
            Price = 6,
            ProductName = "Martini",
            Quantity = 2
        }
    });
}
```

```
[Fact]
0 references
public async void Should_only_allaw_2_times_a_drinks_order()
{
    var events = new ITableEvents[]
    {
        new TableReserved...,
        new DrinksOrdered {
            Order = new Order...
        },
        new DrinksOrdered {
            Order = new Order...
        }
    };
    var newEvents = new List<ITableEvents>();

    var aggregate = new Table(events);
    Action act = () => aggregate.OrderDrinks(new Order...);
    act.Should()
        .Throw<Exception>()
        .WithMessage("Too many drinks :-)");

    await aggregate.PlayAllEvents(async e =>...);

    newEvents.Any().Should().BeFalse();
}
```

When to use Event Sourcing?

When to use Event Sourcing?

- If you have many complex transitions and interactions.



When to use Event Sourcing?

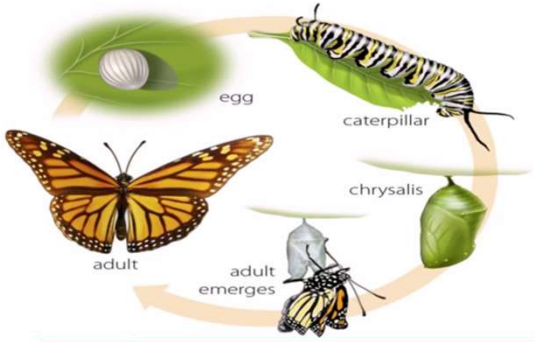
- If you need audits. Or must be able to prove how you arrived at a state.

When to use Event Sourcing?

- If you want to be able to debug what happened in production.

When to use Event Sourcing?

- If you need time in your model.
  => For example: if first a and then c happens then x does something. If first b and then a happens then x is not allowed or does something completely different.

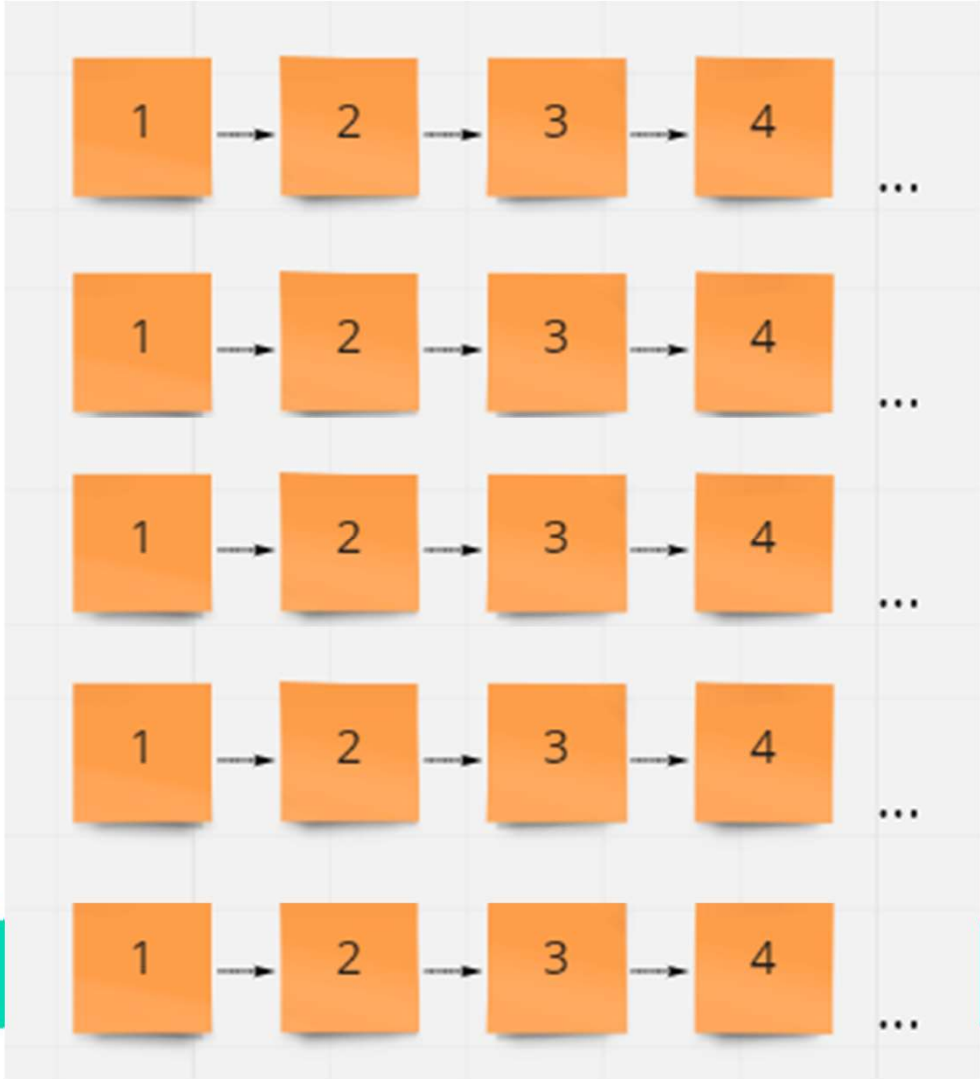


Pro and contra

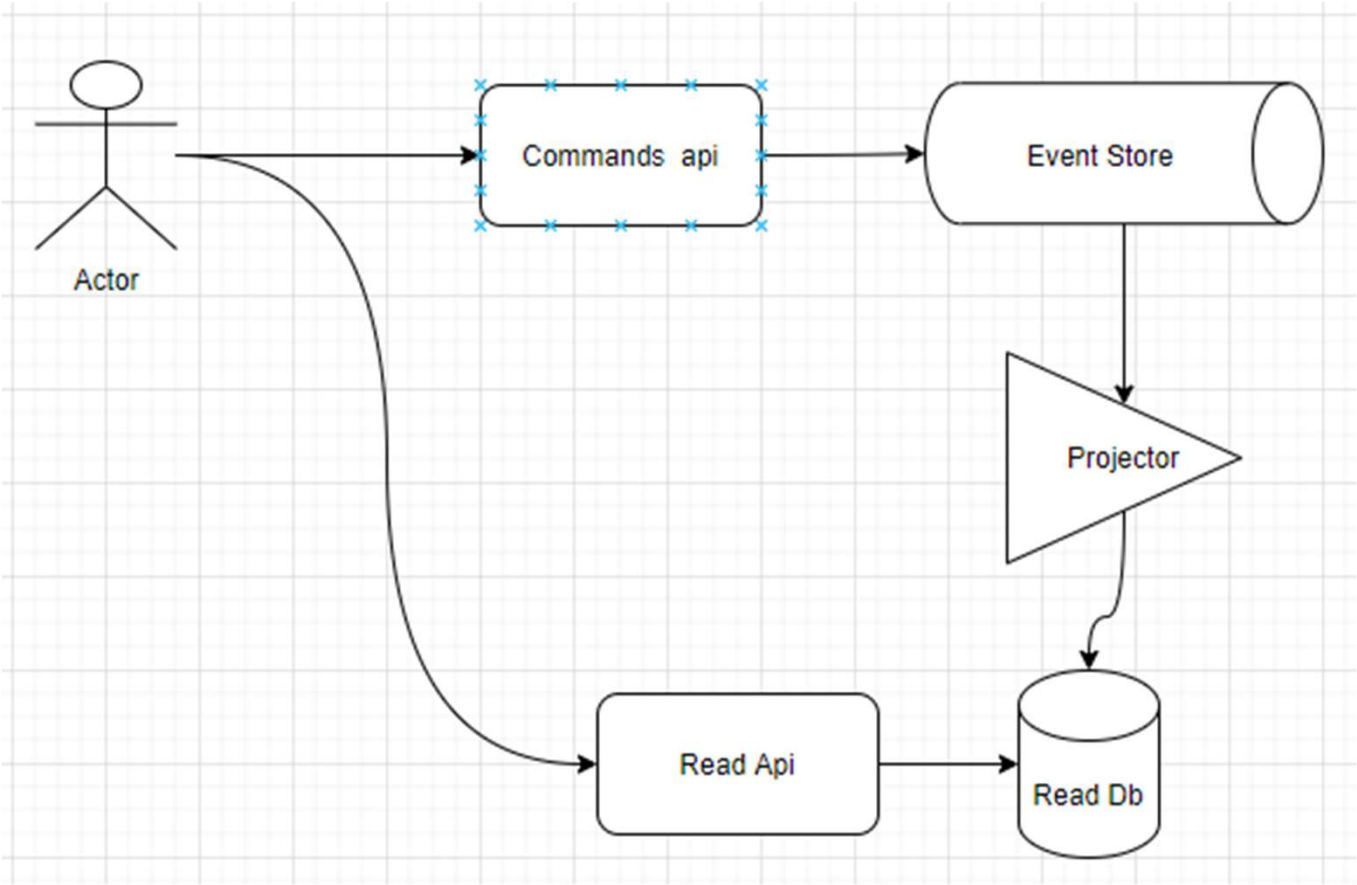
- To have time in your system
- To query over time
- To move your complexity in the aggregates
- **If mastered** it, your system becomes modular

- It requires a lot of discipline from the team.
- Your system becomes more complex
- Your system feels less familiar

Select * from Tables where firstOrder != apperitif and …

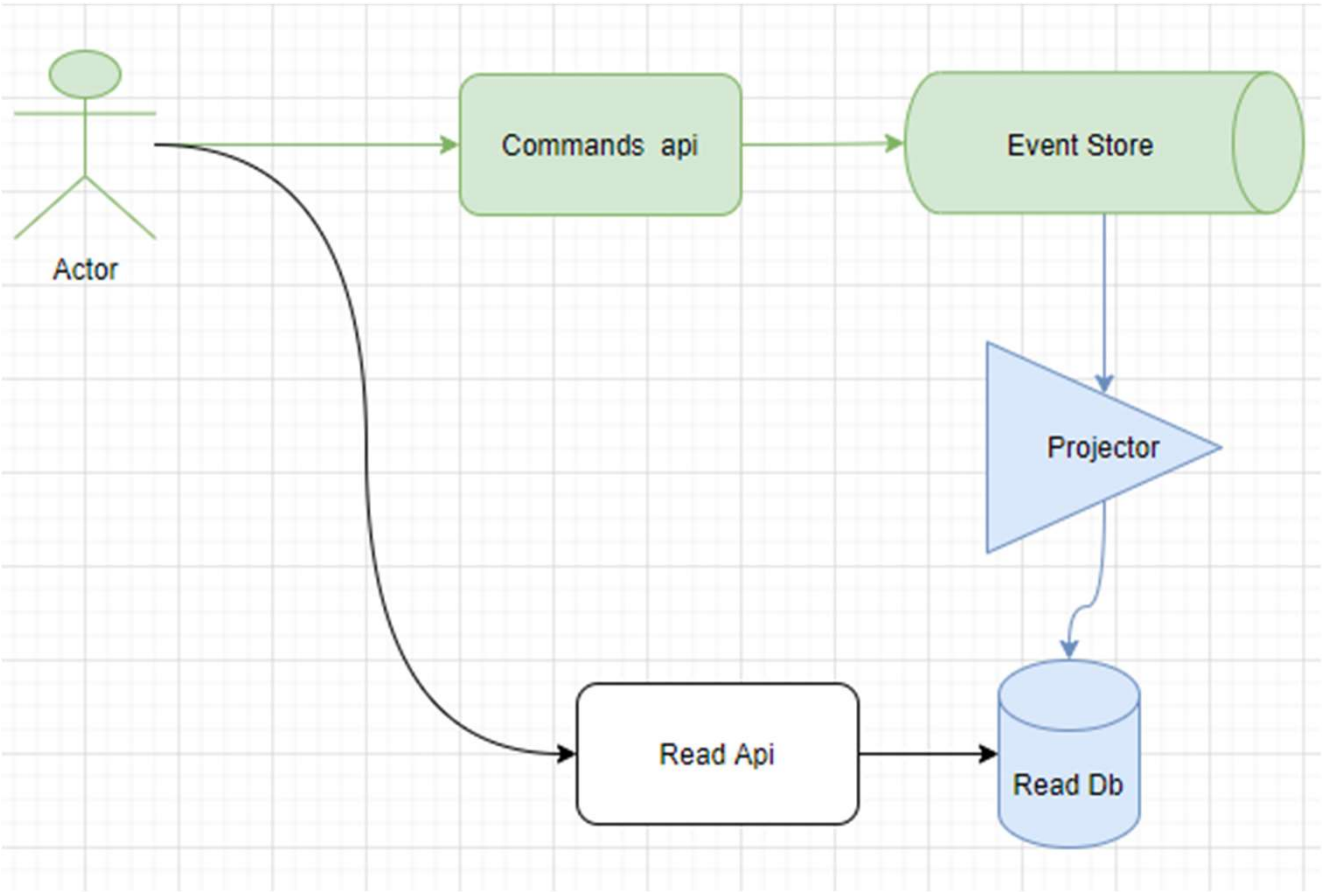


CQRS



CQRS

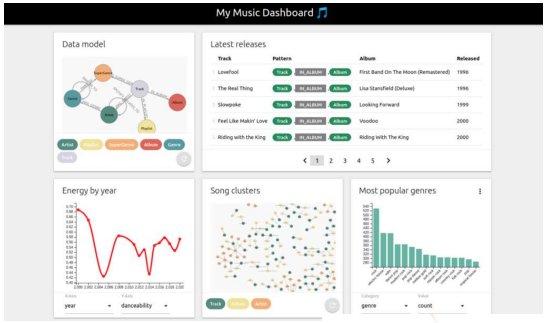
What do we already have



CQRS

Where to use a read model?

Everything you need to visualize



- A list
- A detail e.g. all reserved tables
- Everything that is processed in the kitchen
- What are the three most common orders
- What is the average price per minute and what are the top 5 orders

CQRS

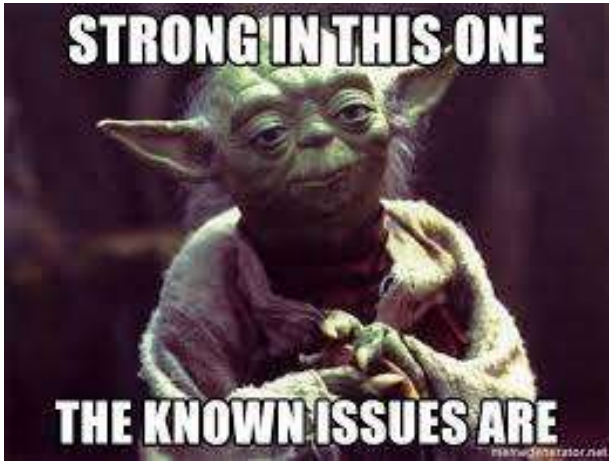
Every domain event that is needed for your read model is translated into a public event

```
public class DrinksOrdered : ITableEvents
{
    7 references | 2/2 passing
    public Order Order { get; init; }
}
```

```
namespace EventSourcingDemo.PublicEvents
{
    0 references
    internal class DrinksOrdered
    {
        0 references
        public Order Order { get; set; }
        0 references
        public int TableId { get; init; }
        0 references
        public string Name { get; init; }
    }
}
```

CQRS

Corresponding records are retrieved adjusted and stored in the DB



CQRS

Events should no longer be ordered.

1. If you receive a debit of €100 for a credit card once.
2. You can't find the credit card no.
3. Then just create it with an amount of €0.
4. Debit the card for €100, where it turns negative.
5. If then your create comes in for €200. then it will ends at €100.

CQRS

Advantage?

- To deploy your read and write db separately.
- Scale separately.
  - Filing taxes at the last minute will cause much less inconvenience.
- If there are many transactions, your read db will catch up slowly. But it doesn't block your command side.
- To create a completely new read model and feed it with all historical event.
  - New report contains all historical data after processing.

Those same public events can also be sent to other services. There they are processed by policies.

Code @
https://github.com/kimVanRenterghemNew/visugxl-2021EventSourcing

Kim Van Renterghem demo code for visugxl-2021		77fa13e 1 minute ago	3 commits
EventSourcingDemo	demo code for visugxl-2021		1 minute ago
ventSourcingDemo.Test	demo code for visugxl-2021		1 minute ago
.editorconfig	demo code for visugxl-2021		1 minute ago
.gitignore	demo code for visugxl-2021		1 hour ago
EventSourcingDemo.sln	demo code for visugxl-2021		1 minute ago
LICENSE	Initial commit		12 hours ago
README.md	Initial commit		12 hours ago

Thank you!

codit

cegeka
IN CLOSE COOPERATION

Capgemini

team4talent

involved

ORDINA

Xpirit

AXXES

Microsoft