Event Sourcing and CQRS for dummies

Kim Van Renterghem

Who Am I?

KEY FACTS

Kim Van Renterghem

Developer since 2008

Senior developer @ zf-scalar

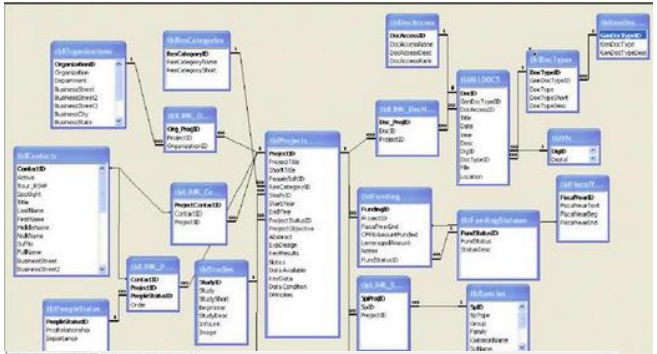
kim-van-renterghem

Traditional systems

CRUD

Whenever we change something, we lose the previous state

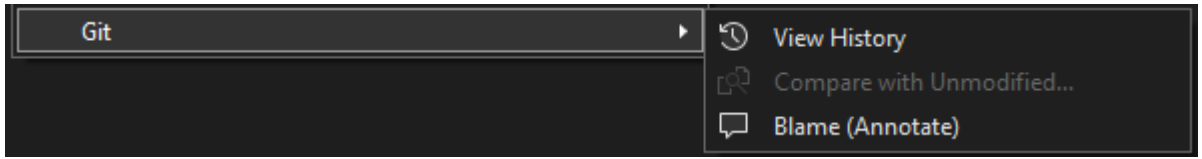

```
_id: "3497ea4b-9609-4ef9-b80e-b00215bd5758"
OrganizationId : "813b6fca-19b5-43ae-afdd-bffd035e5811"
Name : "Indus Zone"
Description : null
Type : "IndustrialZone"
▸ Address : Object
▸ Access : Object
UpdatedOn : 2023-12-13T09:08:17.173+00:00

_id: "edcc02f2-37b1-48a8-aa44-0eacb01db68e"
OrganizationId : "813b6fca-19b5-43ae-afdd-bffd035e5811"
Name : "chennai"
Description : null
Type : "CustomerSite"
▸ Address : Object
▸ Access : Object
UpdatedOn : 2023-12-13T09:08:17.173+00:00

_id: "8ffe7b6e-88bb-451a-a637-ca13f5bd4332"
OrganizationId : "813b6fca-19b5-43ae-afdd-bffd035e5811"
Name : "Cust.Site for Aprt"
Description : "updating to send update event"
Type : "CustomerSite"
▸ Address : Object
▸ Access : Object
UpdatedOn : 2023-12-13T09:08:17.173+00:00
```

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

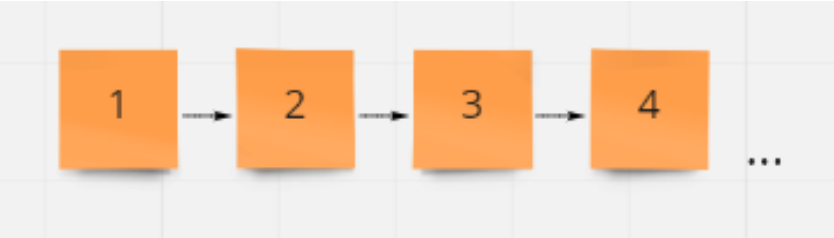
They show when something changes.

=> But not how or why.

Logs

They do not leave 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 of the past

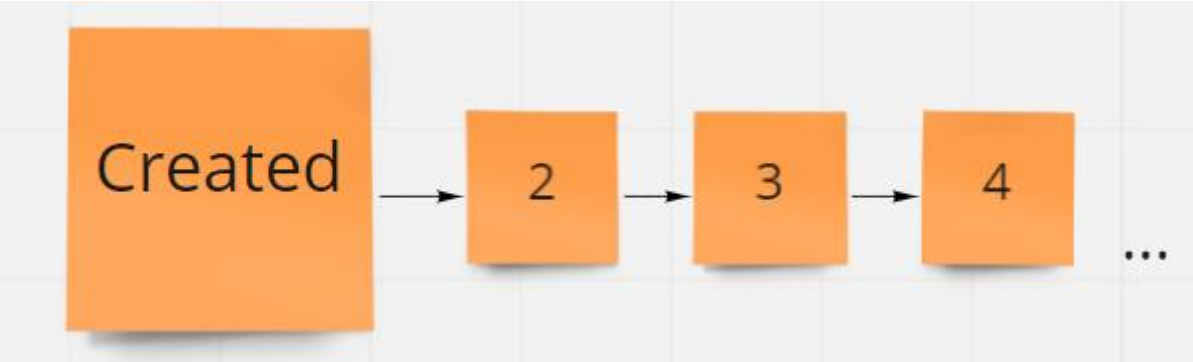
for example:

- Table reserved
- Two martinis ordered
- Martinis 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

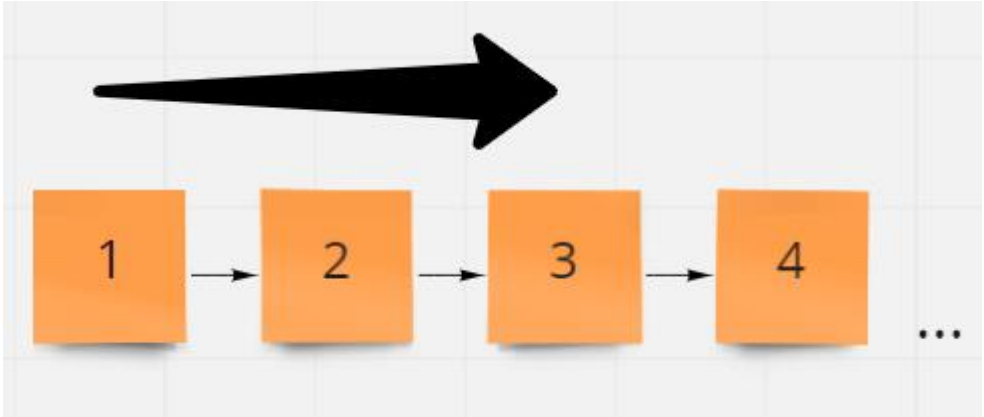
- Table reserved



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

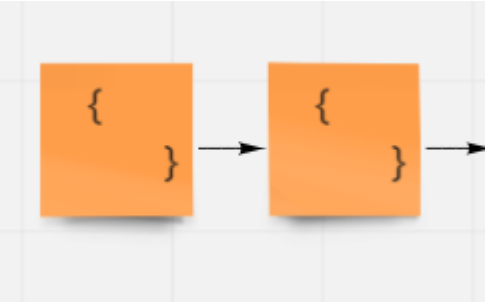
- Should be **add** only
  - 10l red wine ordered
  - 10l red wine order cancelled
  - 1l red wine ordered

  no removing or adjusting!



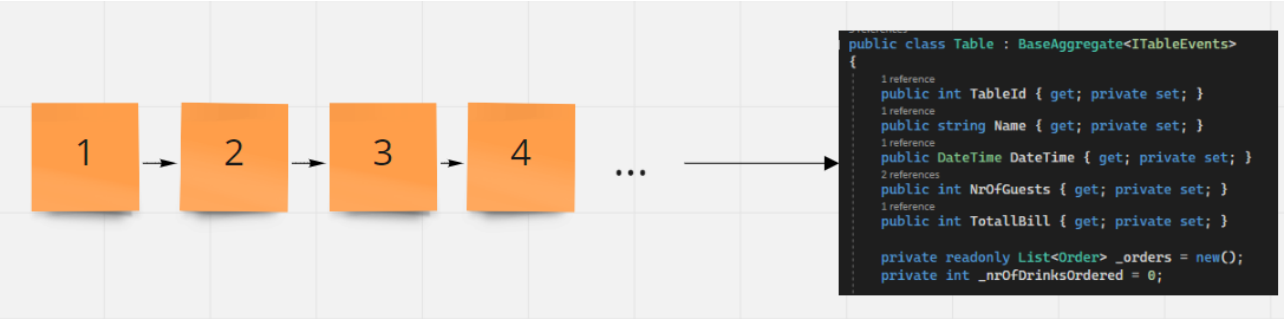
What if we now used our logs as "the only source of truth"

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



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

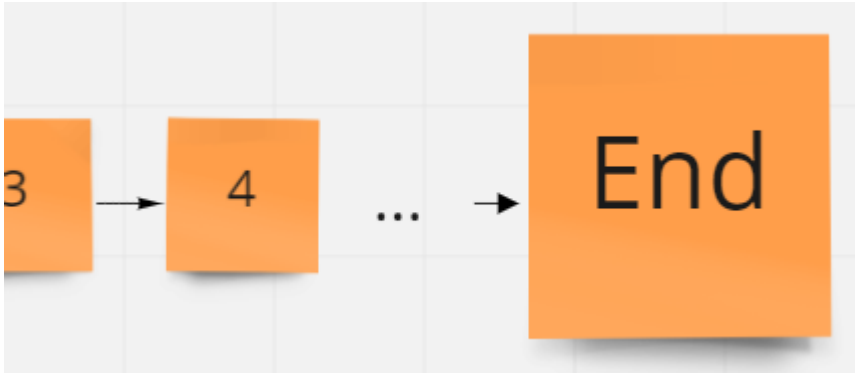
- Now you have time in your model and the sum of all transitions is the current state



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

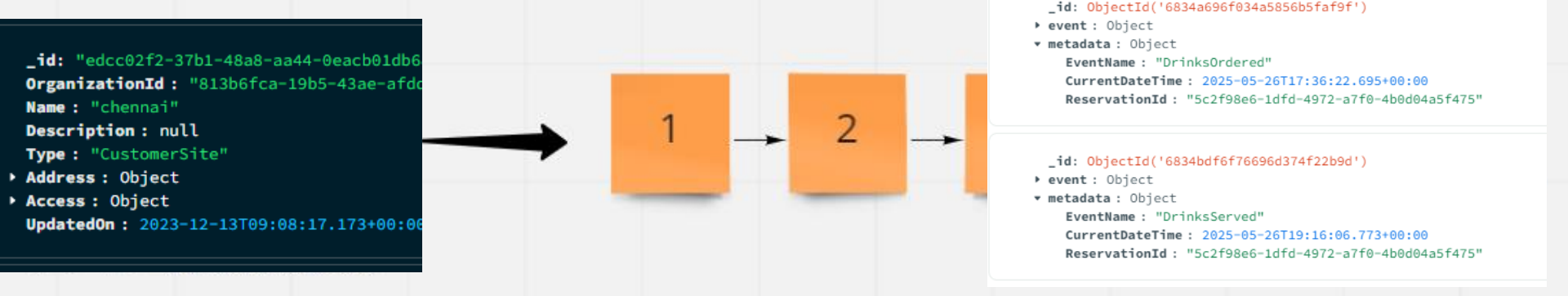
When stream ends, mark it with an event

- Dinner paid
- Guests left



This is what we call “Event sourcing”

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

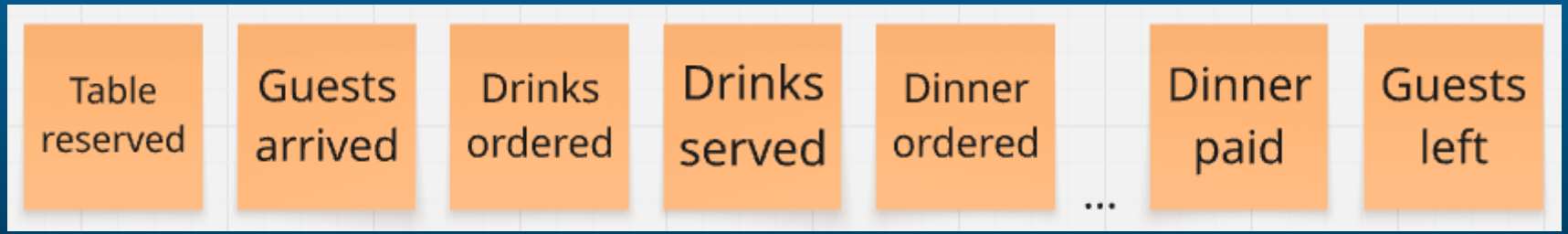


This is what we call “Event sourcing”

Domain events must always be (re)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)



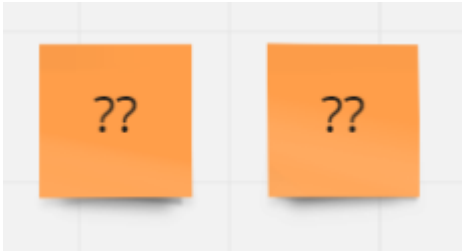
Where to begin?

Event Storming

To explore your domain.

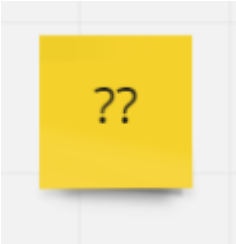
Event Storming

Because your events have yet to be determined and discovered.



Event Storming

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

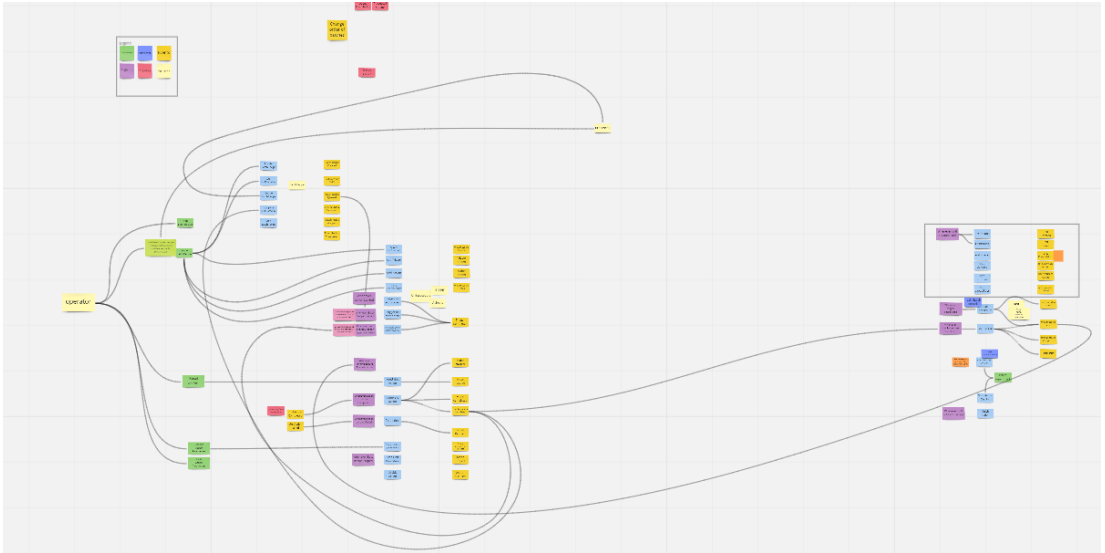


Event Storming

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.

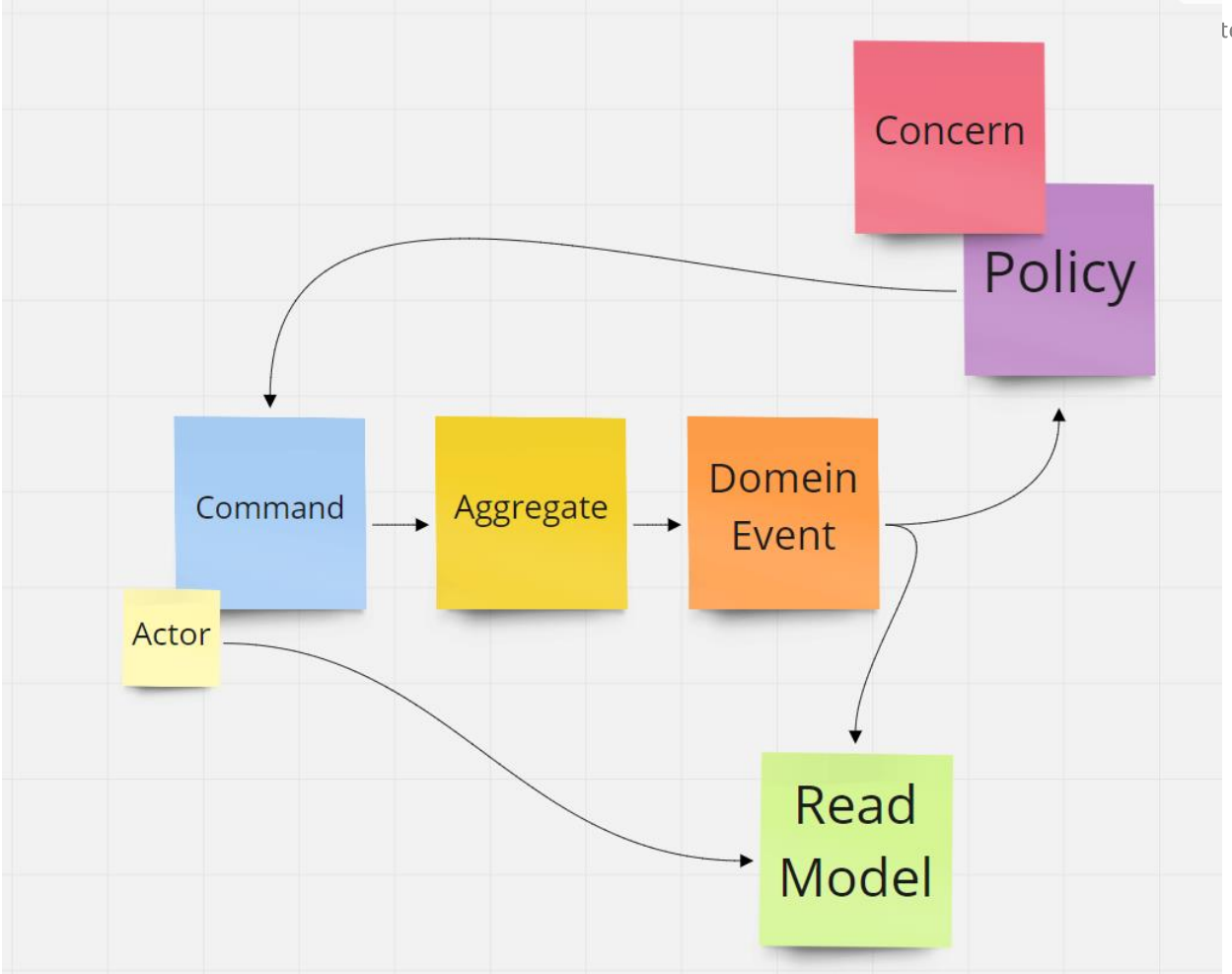
Event Storming

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



Event Storming

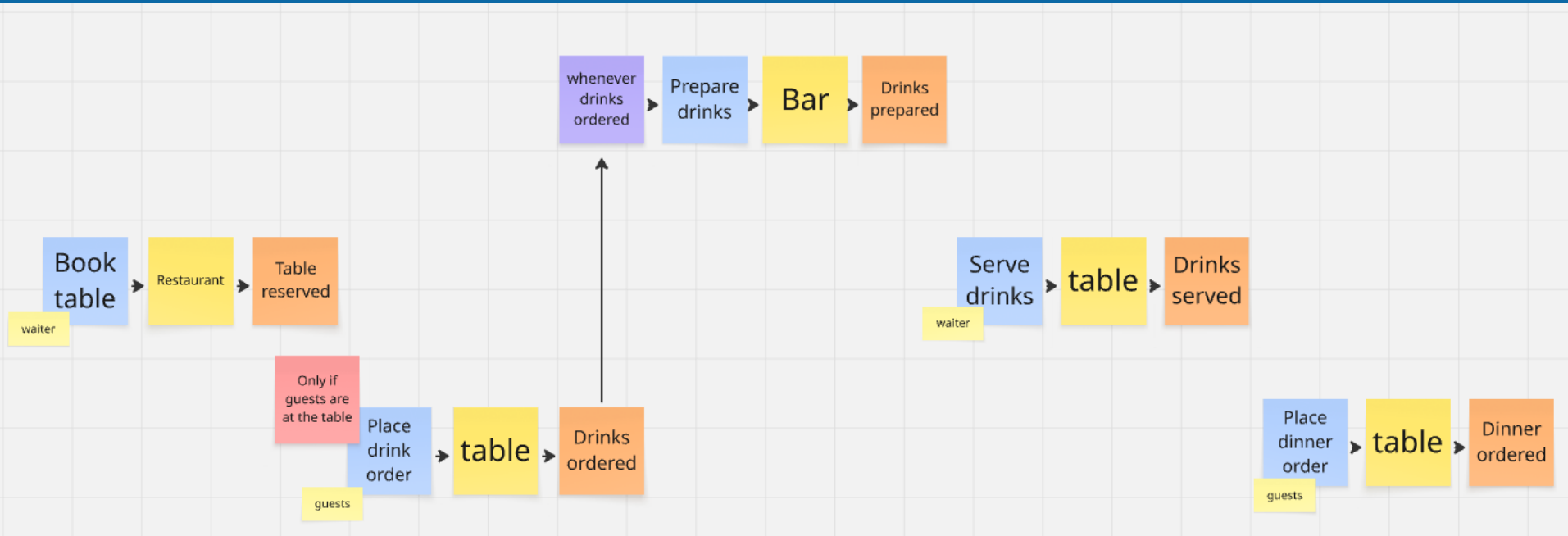
Process



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



Event Storming

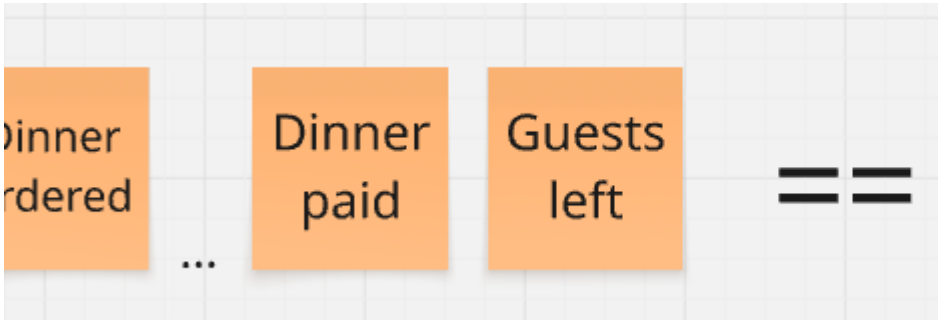
- Remains a continuous process with every change.
- It keeps improving your process from your model.

Event Storming

- Set up the space
  - Prepare a large wall or online board, sticky notes, and markers.
- Collect domain events
  - Let participants write and place events (past tense) on the timeline.
- Organize and group
  - Remove duplicates, sequence events, and group them by flow or theme.
- Add supporting elements
  - Use colored notes for commands (blue), actors/systems (yellow), and views (green).
- Explore and refine
  - Identify issues, gaps, and possible bounded contexts through discussion.

Event Storming

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




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

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

1 reference
private void DrinksServedHandler(DrinksServed served)
{
    var existingOrder = _orders.FirstOrDefault(o :Order => o.OrderId == served.Order);

    _orders.Remove(existingOrder);
}
```

Constructors and Commands

```
2 references
public Table(int tableId, DateTime dateTime, string name, int nrOfGuests)
{
    RegisterHandlers();
    PublishNewEvent(new TableReserved( ReservationId: Guid.NewGuid() , tableId, name, dateTime, nrOfGuests));
}

3 references
public Table(IEnumerable<TableEvent> events)
{
    RegisterHandlers();

    //play all the evens
    events // IEnumerable<TableEvent>
        .ToList() // List<TableEvent>
        .ForEach(PlayEvent);
}
```

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

    PublishNewEvent(new DrinksOrdered(order));
}
1 reference
public void ServeDrinks(Guid order)
{
    var existingOrder = _orders.FirstOrDefault(o :Order => o.OrderId == order);

    if (existingOrder == null)
    {
        throw new Exception("This order dos not exists.");
    }
    PublishNewEvent(new DrinksServed(order));
}
```

Events

```
7 references | - changes | -authors, -changes
public record TableReserved(
        int TableId,
        string Name,
        DateTime DateTime,
        int NrOfGuests
    ) : ITableEvents;
```

```
8 references | - changes | -authors, -changes
public record DrinksOrdered(Order Order) : ITableEvents;

12 references | - changes | -authors, -changes
public record Order(
    string ProductName,
    int ProductId,
    int Quantity,
    int Price,
    string Comment);
```

Handlers

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

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

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

1 reference
private void DrinksServedHandler(DrinksServed served)
{
    var existingOrder = _orders.FirstOrDefault(o :Order => o.OrderId == served.Order);
    _orders.Remove(existingOrder);
}
```

At Work?

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.

```
0 references
public class OrderDrinksCommandHandler(TablesStore tablesStore)
        : IRequestHandler<OrderDrinksCommand>
{
    0 references
    public async Task Handle(OrderDrinksCommand request, Cancellation
    {
        var table = await tablesStore.Get(request.ReservationId);

        table.OrderDrinks(request.Order);

        await tablesStore.SaveAsync(table);
    }
}
```

Testing?

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

    events.Should().ContainSingle();

    events[0] // ITableEvents
        .Should() // ObjectAssertions
        .BeEquivalentTo(expectation: new TableReserved(
            TableId: 6,
            Name: "kim vr",
            DateTime: new DateTime(year: 2021, month: 12, day: 31),
            NrOfGuests: 2));
}
```

Testing?

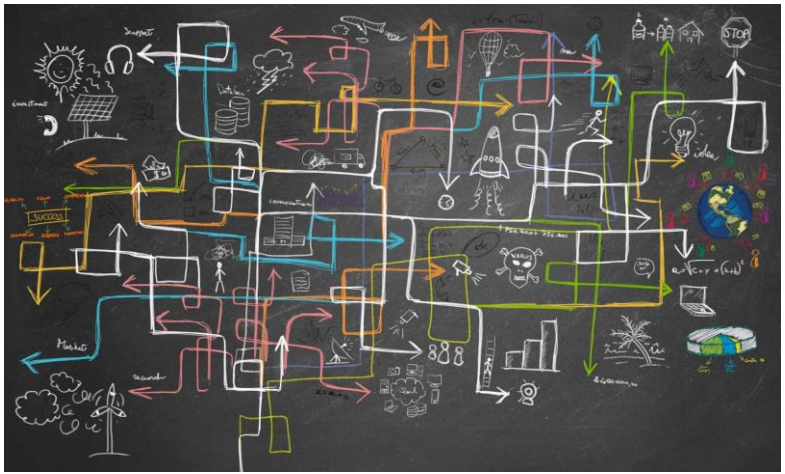
```
public async ValueTask Should_only_allow_2_times_a_drinks_order()
{
    var events = new TableEvent[]
    {
        new TableReserved(...),
        new DrinksOrdered(...),
        new DrinksOrdered(...),
    };
    var newEvents = new List<TableEvent>();

    var aggregate = new Table(events);
    var act :Action = () => aggregate.OrderDrinks(
        new Order(
                Guid.NewGuid(),
            ProductName: "Martini",
            ProductId: 8,
            Quantity: 2,
            Price: 14,
            Comment: ""
        ));

    act.Should() // ActionAssertions
        .Throw<Exception>()
        .WithMessage(expectedWildcardPattern: "Too many drinks :-)");

    await aggregate.PlayAllEvents(writeEvent: async e :TableEvent =>
    {
        await Task.FromResult(0);
        newEvents.Add(e);
    }); // ValueTask

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

```
public async ValueTask Should_be_able_to_order_a_drink()
{
    var events = new ITableEvents[]
    {
        new TableReserved(
            TableId: 6,
            Name: "kim vr",
            DateTime: new DateTime(year: 2021, month: 12, day: 31),
            NrOfGuests: 2
        )
    };

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

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

    newEvents.Should().ContainSingle();
    newEvents[0] // ITableEvents
        .Should() // ObjectAssertions
        .BeEquivalentTo(
            expectation: new DrinksOrdered(
                new Order(
                    ProductName: "Martini",
                    ProductId: 6,
                    Quantity: 2,
                    Price: 6,
                    Comment: ""
                )
```

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.

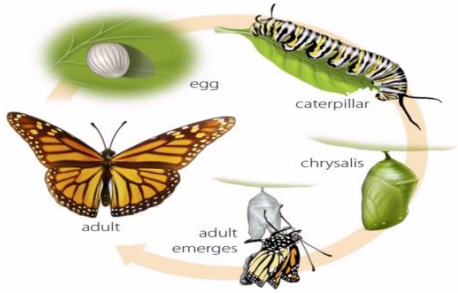
When to use Event Sourcing?

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.

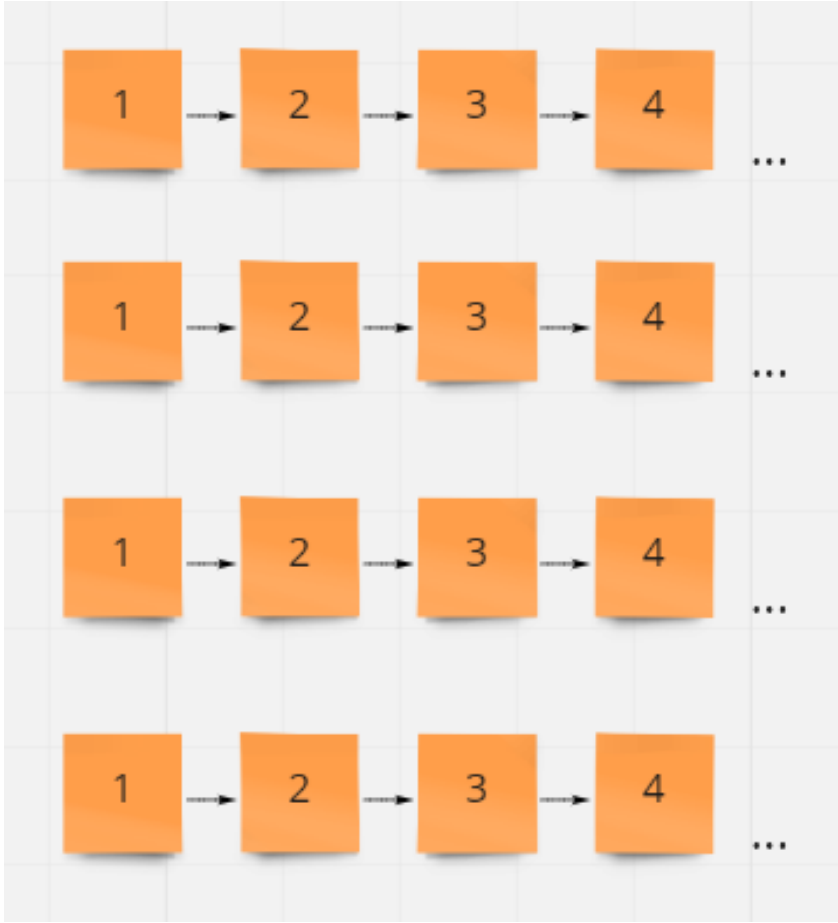


Pro and contra

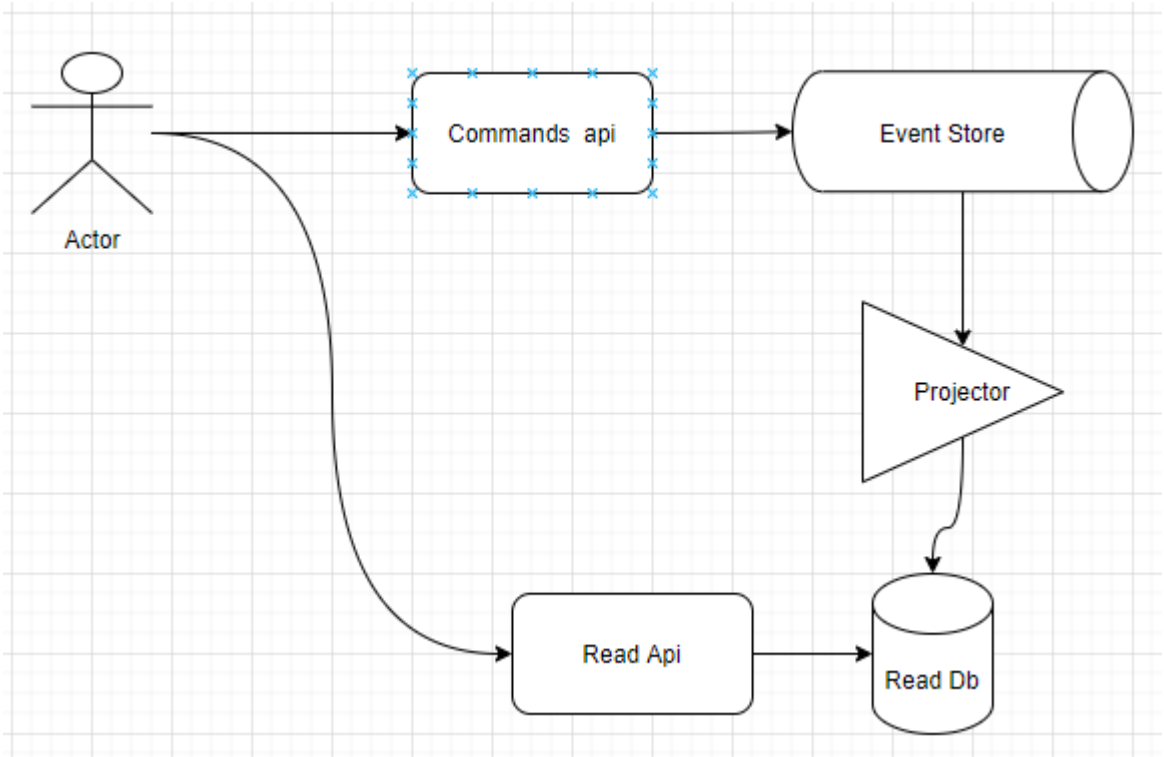
- To have time in your system
- To query over time
- To move your complexity in the aggregates
- **When mastered**, your system becomes modular

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

Select * from Tables where firstOrder != appetizer and ...

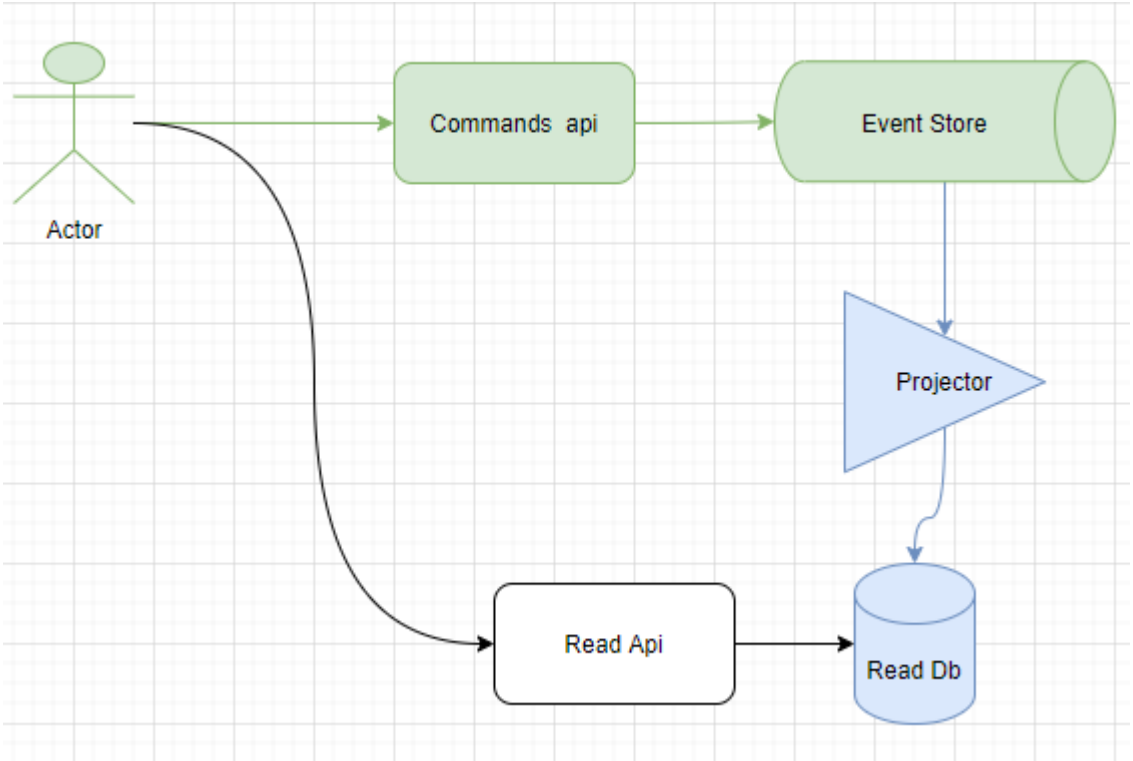


Command Query Responsibility Segregation



CQRS

Commands



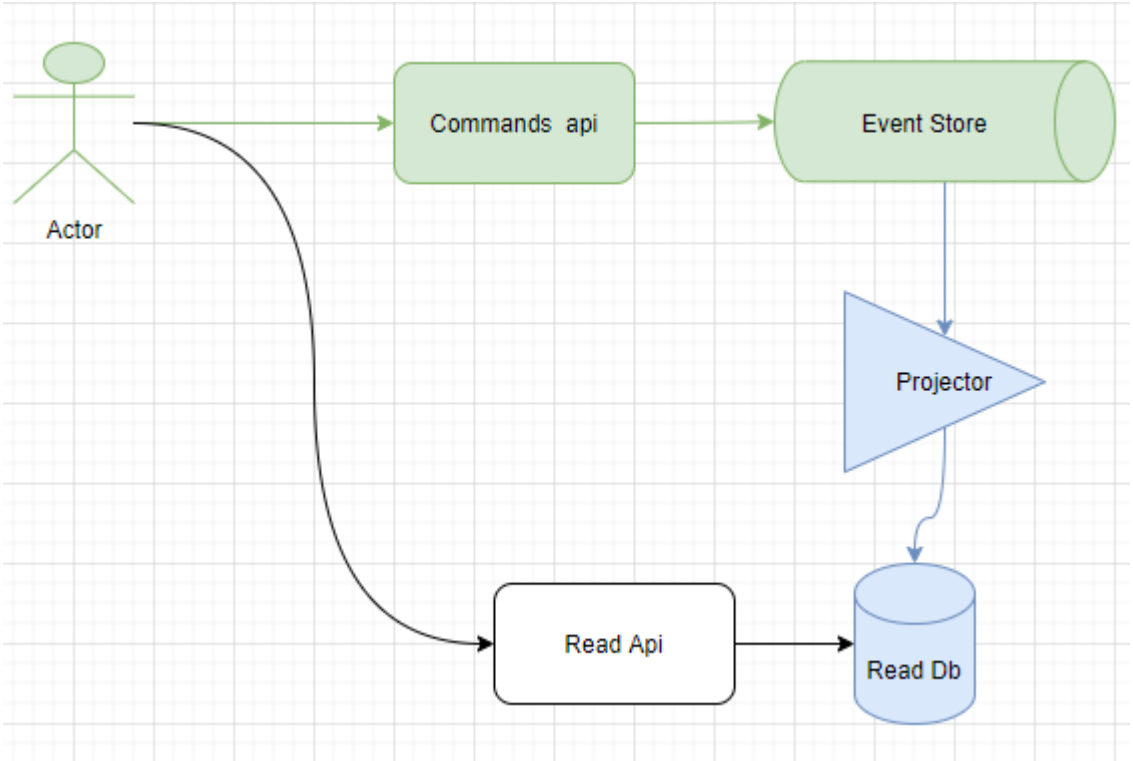
CQRS

```
[HttpPost(template: "order-drinks")]
[ProducesResponseType(statusCode: StatusCodes.Status200OK)]
[ProducesResponseType(statusCode: StatusCodes.Status400BadRequest)]
0 references
public async Task<IActionResult> OrderDrinks([FromBody] OrderDrinksCo
{
    await mediator.Send(command);
    return Ok();
}
```

```
3 references
public record OrderDrinksCommand(
    Guid ReservationId,
    Order Order
) : IRequest;

0 references
public class OrderDrinksCommandHandler(TablesStore tablesStore)
        : IRequestHandler<OrderDrinksCommand>
{
    0 references
    public async Task Handle(OrderDrinksCommand request, CancellationToken cancellationToken)
    {
        var table = await tablesStore.Get(request.ReservationId);

        table.OrderDrinks(request.Order);

        await tablesStore.SaveAsync(table);
    }
}
```

CQRS

Projector

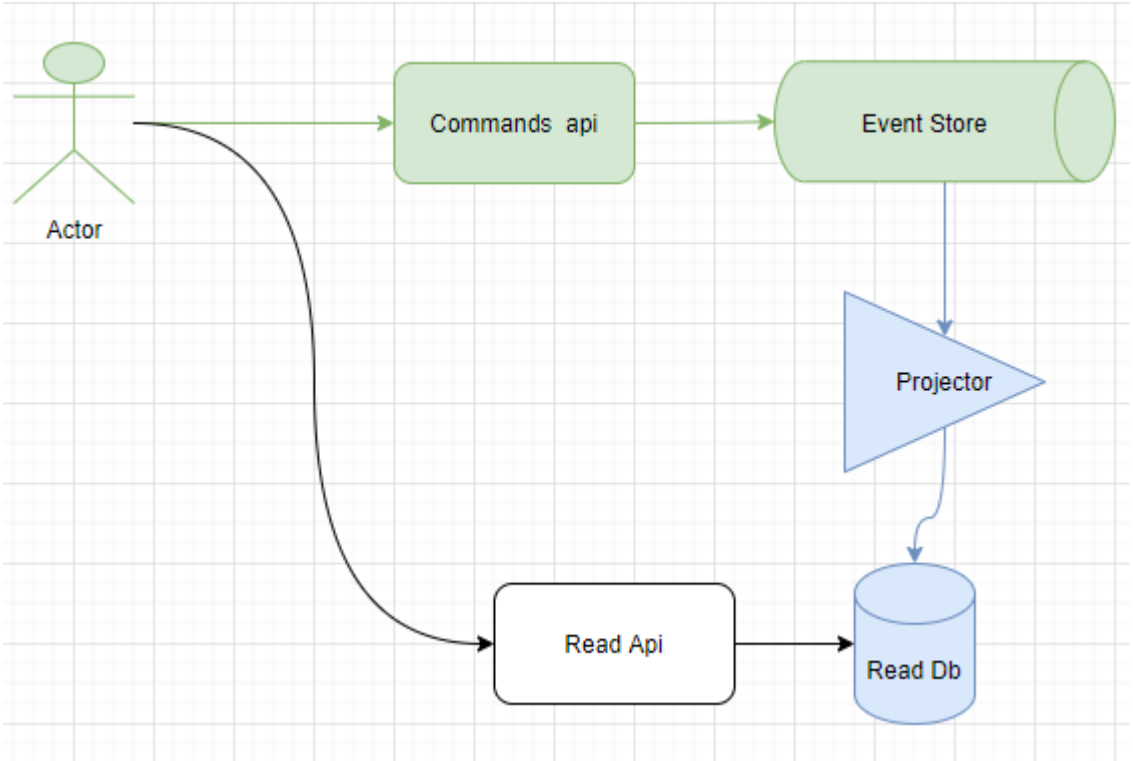


CQRS

```
0 references
public class OrderProjector(OrderCollection orderCollection) :
        INotificationHandler<PublicEvents.DrinksOrdered>,
        INotificationHandler<PublicEvents.DrinksServed>
{
    0 references
    public async Task Handle(PublicEvents.DrinksOrdered notification, Cancellat
    {
        await orderCollection.AddOrderAsync(notification);
    }

    0 references
    public async Task Handle(PublicEvents.DrinksServed notification, Cancellati
    {
        await orderCollection.SetToServed(notification.OrderId);
    }
}
```

CQRS

Query



CQRS

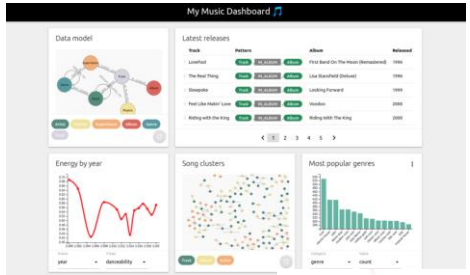
```
[HttpGet]
[ProducesResponseType(statusCode: StatusCodes.Status200OK)]
0 references
public async Task<IEnumerable<Application.Query.Order>> GetAllOrders([F
{
    return await mediator.Send(new GetOrders());
}
```

```
0 references
public class GetOrdersHandler(OrderCollection orderCollection) :
        IRequestHandler<GetOrders, IEnumerable<Order>>
{
    0 references
    public async Task<IEnumerable<Order>> Handle(GetOrders request, Canc
    {
        return await orderCollection.GetAsync();
    }
}
```

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

```
12 references
public record DrinksOrdered(Order Order) : TableEvent
{
    2 references
    public void Accept(EventVisitor visitor)
    {
        visitor.Visit(ordered: this);
    }
}
```

```
4 references
public async Task SaveAsync(Table table)
{
    var visitor = new MongoEventVisitor(table.ReservationId.ToString(), table);

    await table.PlayAllEvents(writeEvent: async e :TableEvent =>
    {
        var (doc :BsonDocument, @event :INotification) = visitor.Transform(e);
        await _collection.InsertOneAsync(doc);

        await _mediator.Publish(@event);
    }); // ValueTask
}
```

```
namespace EventSourcingDemo.PublicEvents;

7 references
public record DrinksOrdered(Guid OrderId, Guid ReservationId, Order Order, int TableId, string Name) : INotification;
```

CQRS

Corresponding records are retrieved adjusted and stored in the DB

CQRS

Events should no longer be ordered.

1. If you received a debit of €100 for a credit card once.
2. You can't find the credit card number.
3. Then just create it with an amount of €0.
4. Debit the card for €100, where it turns negative.
5. If then you create comes in for €200. then it will end at €100.

CQRS

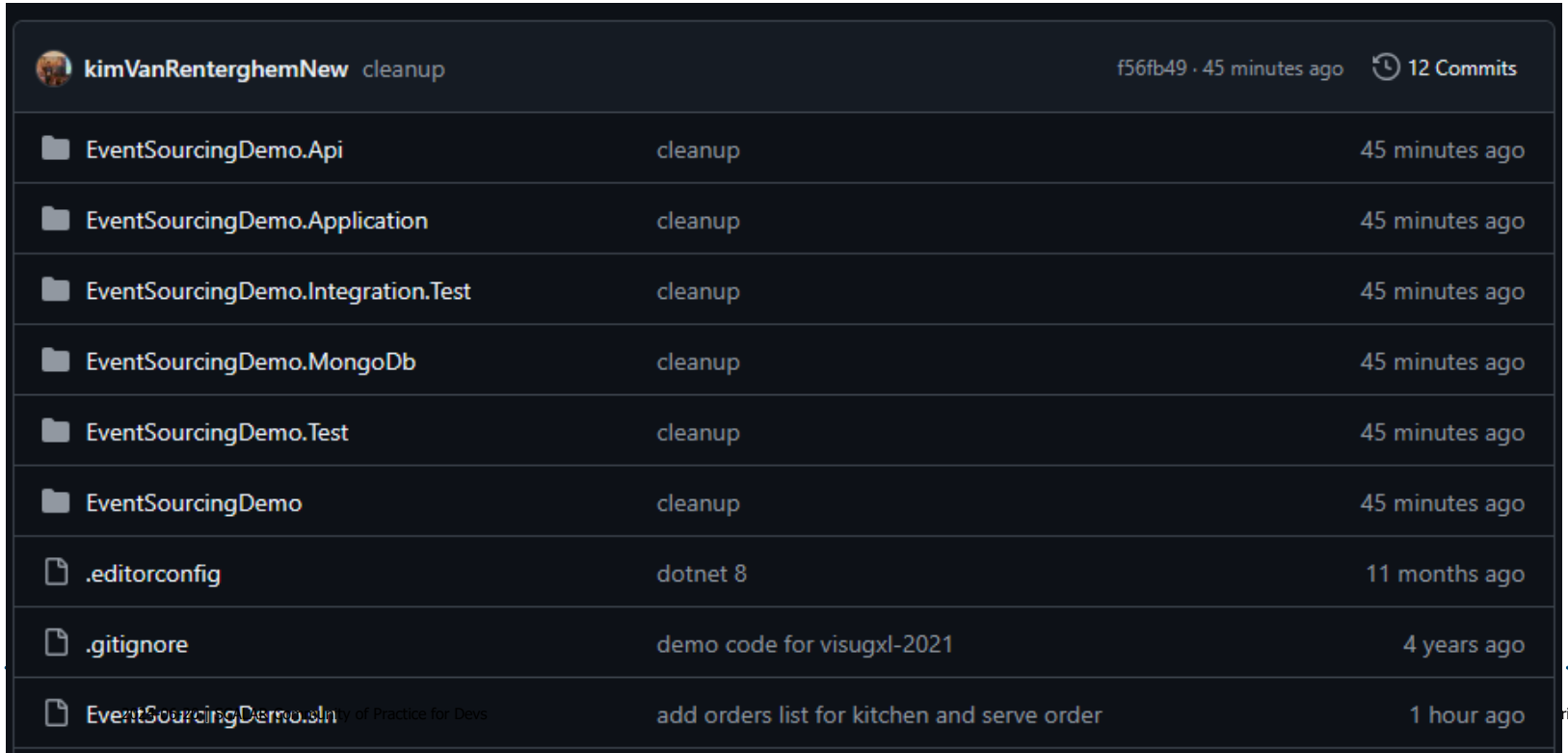
Advantage?

- 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/EventSourcingAndCQRS



kimVanRenterghemNew cleanup	f56fb49 · 45 minutes ago	12 Commits
EventSourcingDemo.Api	cleanup	45 minutes ago
EventSourcingDemo.Application	cleanup	45 minutes ago
EventSourcingDemo.Integration.Test	cleanup	45 minutes ago
EventSourcingDemo.MongoDb	cleanup	45 minutes ago
EventSourcingDemo.Test	cleanup	45 minutes ago
EventSourcingDemo	cleanup	45 minutes ago
.editorconfig	dotnet 8	11 months ago
.gitignore	demo code for visugxl-2021	4 years ago
EventSourcingDemo.sln	add orders list for kitchen and serve order	1 hour ago

Demo

Questions?