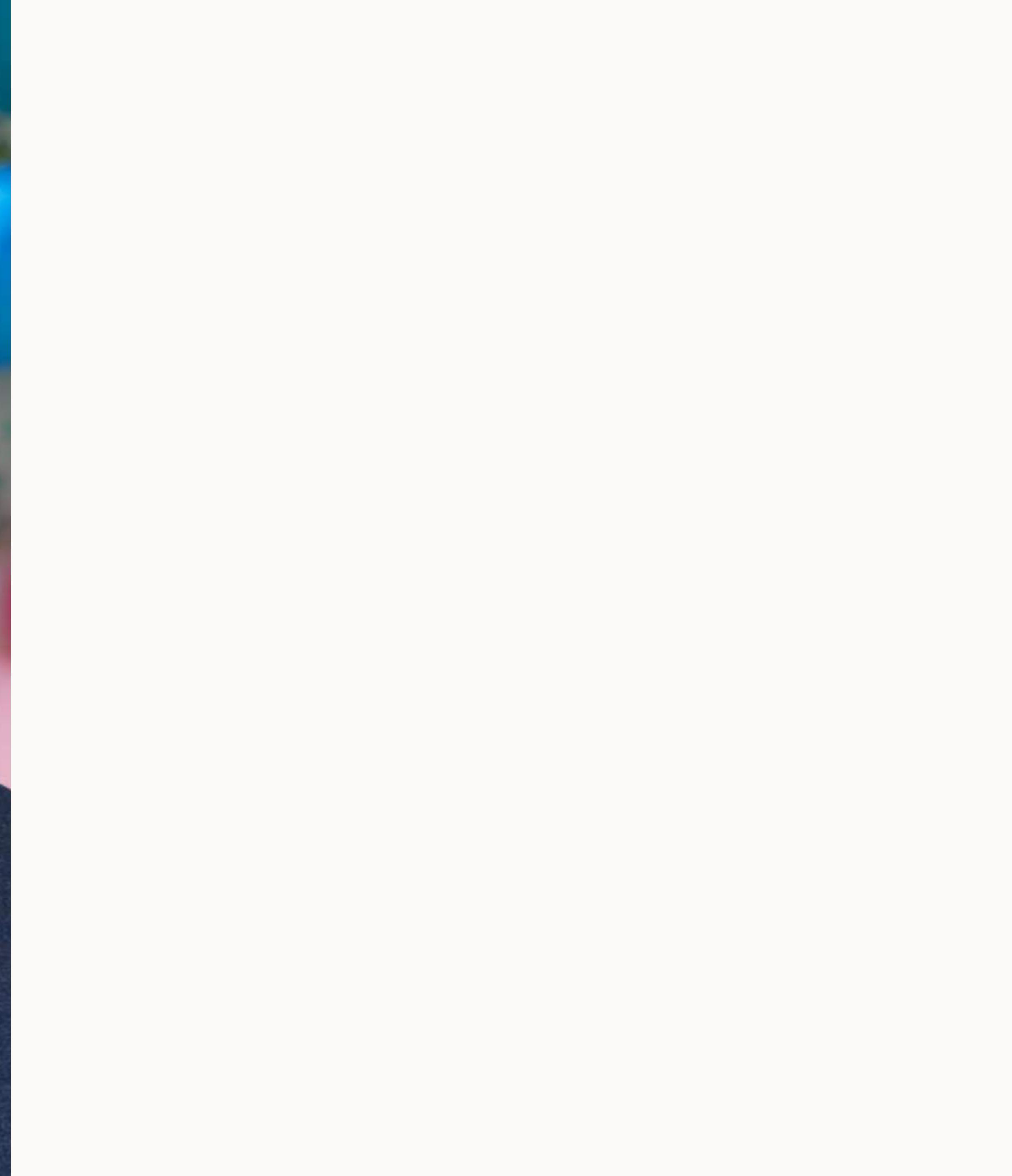


CQRS/ES & Elixir

Bertrand Dubaut | 10/09/2019

impraise





👋 I'm Bertrand



👋 I'm Bertrand

I build systems.



👋 I'm Bertrand

I build systems.

Sometimes with code.



👋 I'm Bertrand

I build systems.

Sometimes with code.

Sometimes with people.

“ ”

Grow your people,
grow your business

impraise

乾隆癸未
天下諸番
徵明永樂



Domain-driven Design

Let's start at the beginning

Domain Driven Design

Bounded contexts

`mix phx.gen.context Meetups meetup meetup`

Aggregate

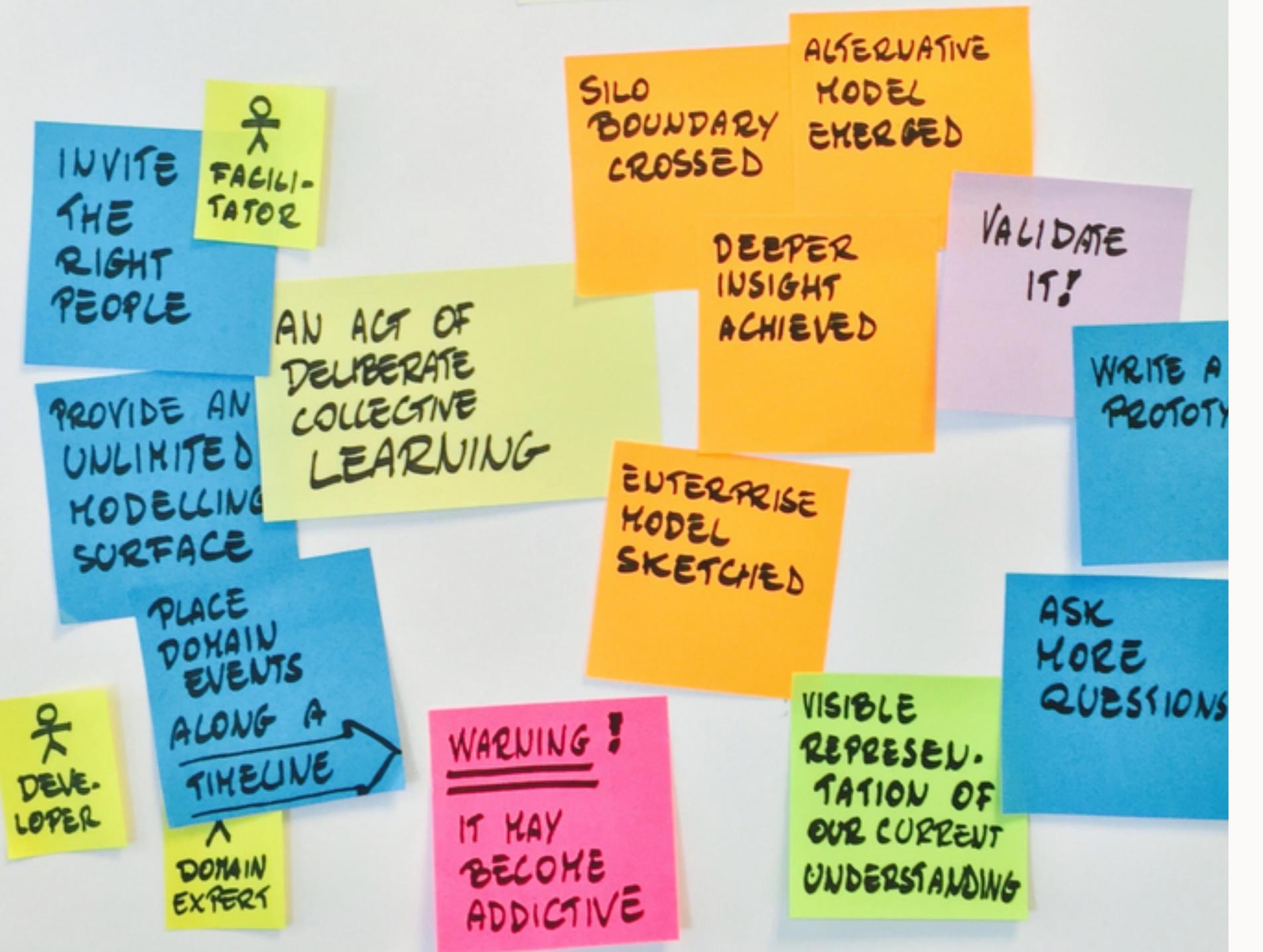
“Cluster of associated objects that we treat as unit”. In general, the aggregate is *invariant*. At the *root* of a *bounded context* is a single aggregate.

Entities & value objects

the building blocks of the aggregates. Entities are defined by their identity, value objects don’t.

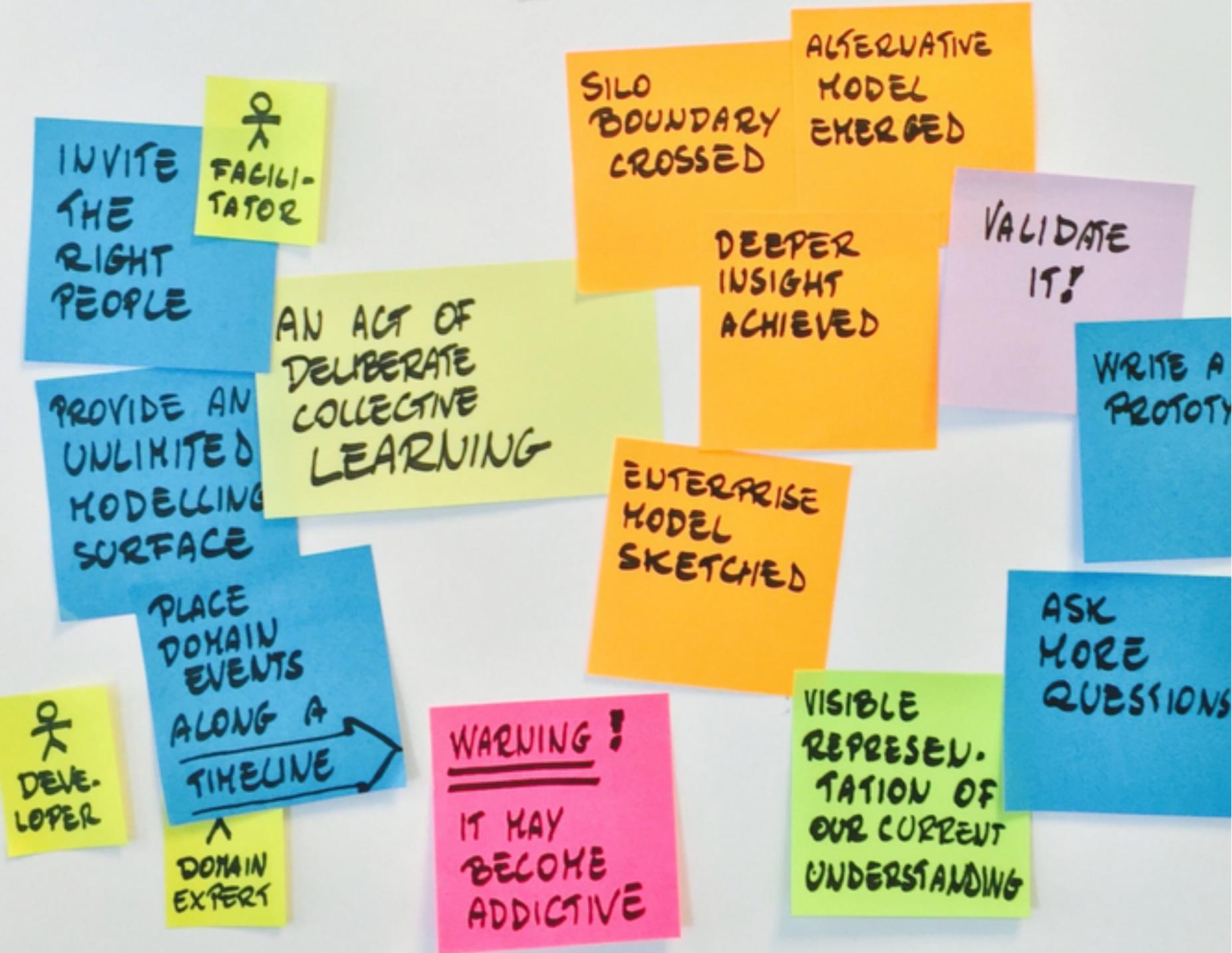
EVENT STORMING

Alberto Brandolini



EVENT STORMING

Alberto Brandolini



Domain modelling tips

Buy Eric Evans' book

“Domain-Driven Design - Tackling complexity in the heart of software”

EVENT STORMING

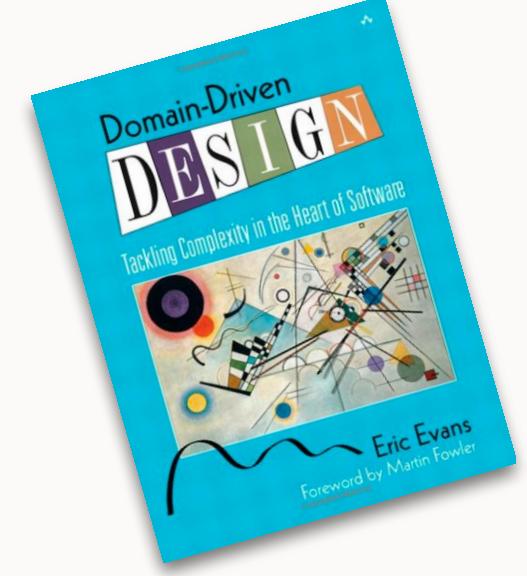
Alberto Brandolini



Domain modelling tips

Buy Eric Evans' book

“Domain-Driven Design - Tackling complexity in the heart of software”



EVENT STORMING

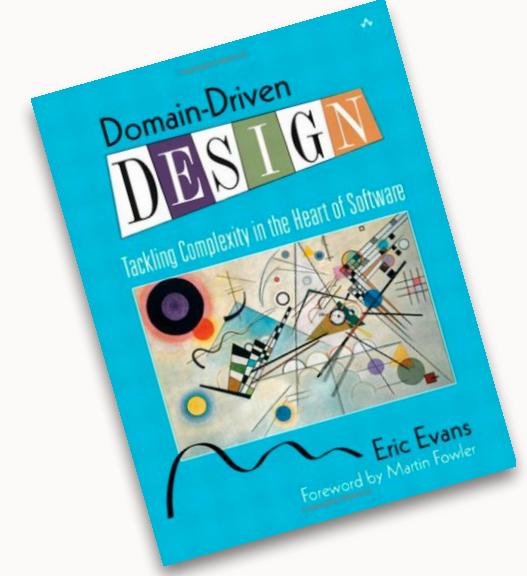
Alberto Brandolini



Domain modelling tips

Buy Eric Evans' book

“Domain-Driven Design - Tackling complexity in the heart of software”



Involve EVERYONE

People with questions, People with answers. From IT, Marketing, Sales, Support, etc.

EVENT STORMING

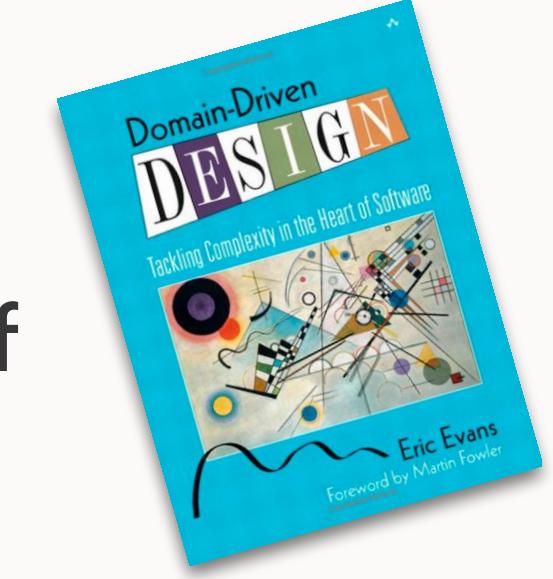
Alberto Brandolini



Domain modelling tips

Buy Eric Evans' book

“Domain-Driven Design - Tackling complexity in the heart of software”



Involve EVERYONE

People with questions, People with answers. From IT, Marketing, Sales, Support, etc.

Event storming can be a powerful tool

Get post-its. Then get more post-its. Put relevant business events over a timeline, then isolate your commands, read models, and your domain model appears in front of your very eyes.

The background of the slide features a stylized, abstract illustration of a horse's head and neck. The horse is white with brown and black markings, including a dark mane and tail. It is set against a dark, swirling background of blue, purple, and green lines that resemble flowing water or energy. The overall aesthetic is artistic and modern.

CQRS

A natural evolution of DDD

impraise



UI

UI

Commands



Commands

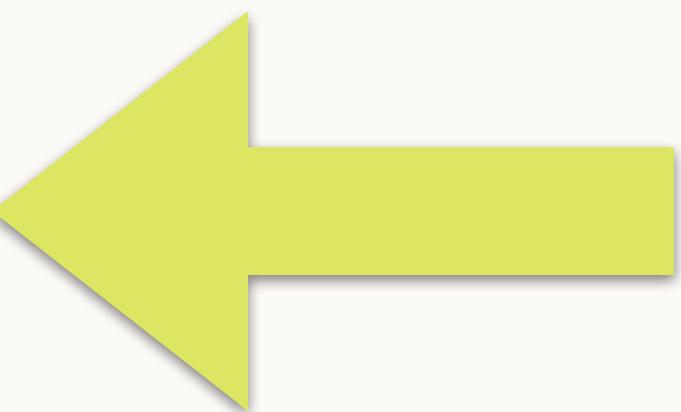
(write operations)

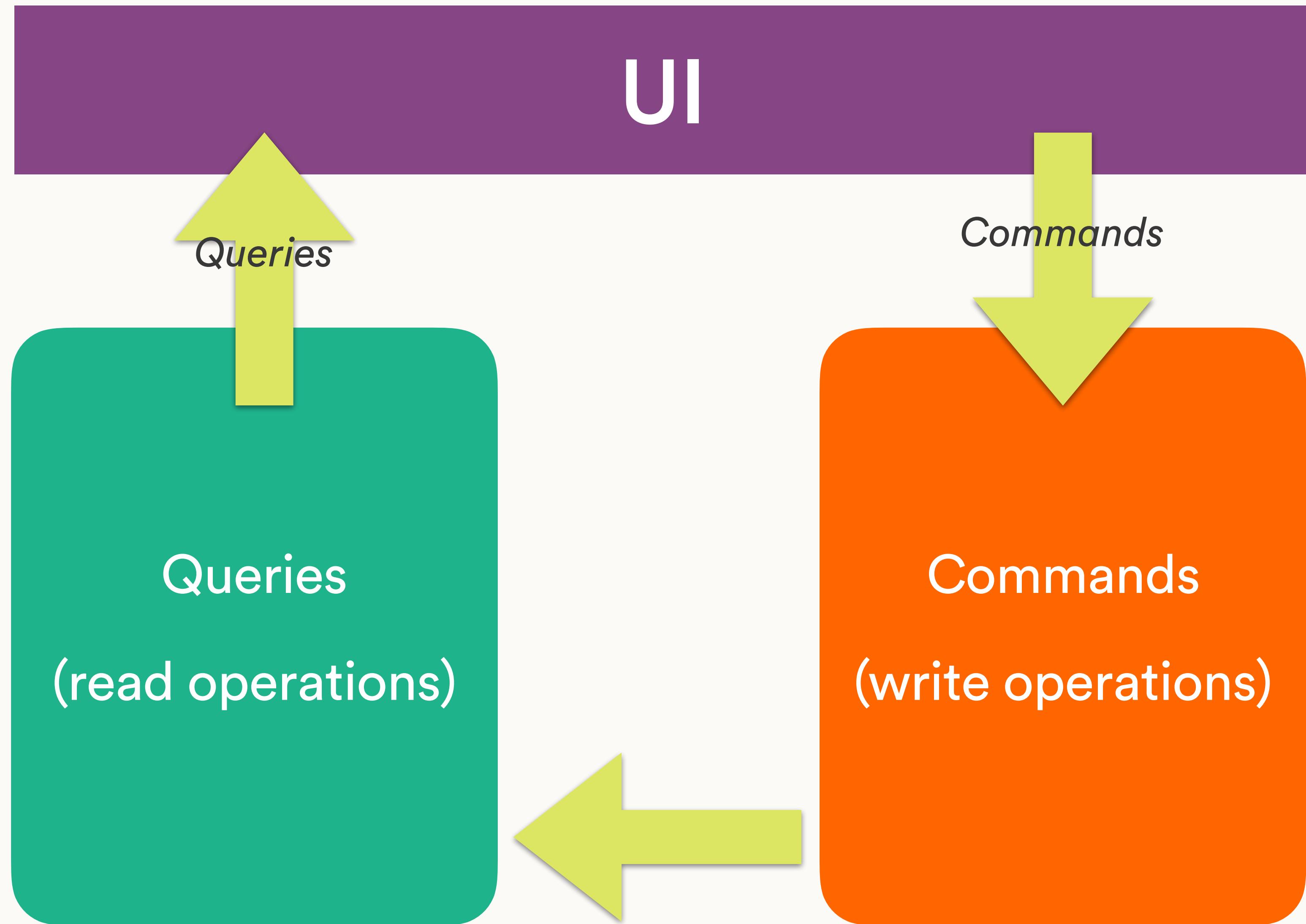
UI

Commands

Commands

(write operations)





Command-Query Responsibility Segregation

Command-Query Responsibility Segregation

Scalability

Scale the writes and the reads
independently.

Command-Query Responsibility Segregation

Scalability

Scale the writes and the reads independently.

Flexibility

Write and re-write the read models based on your business needs

Command-Query Responsibility Segregation

Scalability

Scale the writes and the reads independently.

Flexibility

Write and re-write the read models based on your business needs

Extra complexity

You now have 2 models to understand and maintain.

Event Sourcing

Not quite a new concept

Meetup

uuid

title

date

Has many

Attendees

account_uuid

name

email

Belongs to

Location

uuid

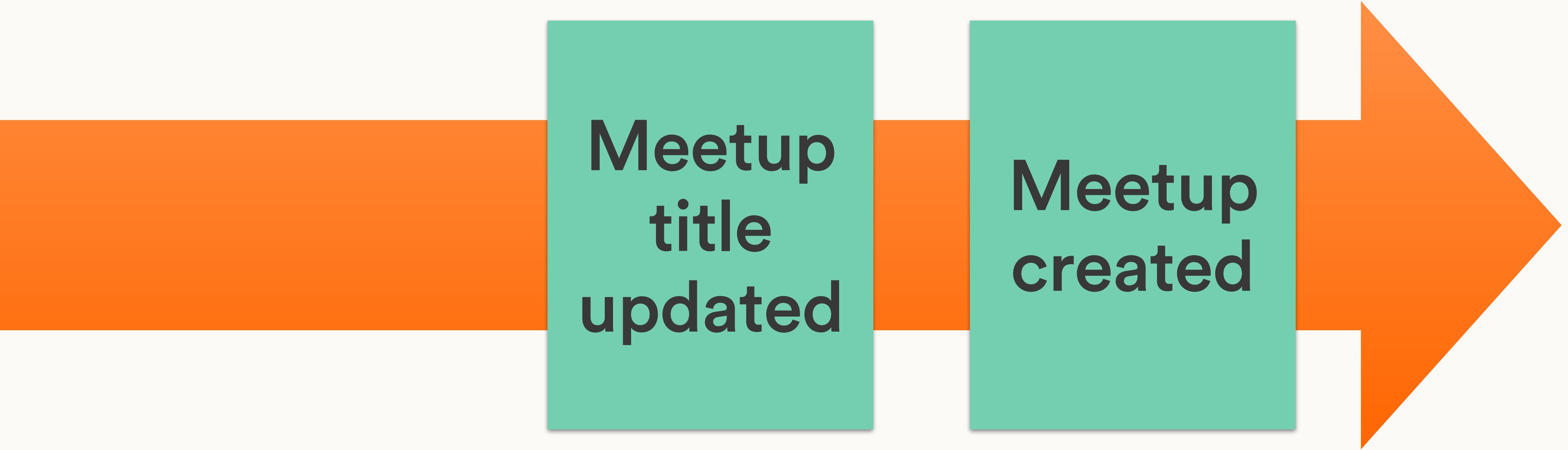
name

address



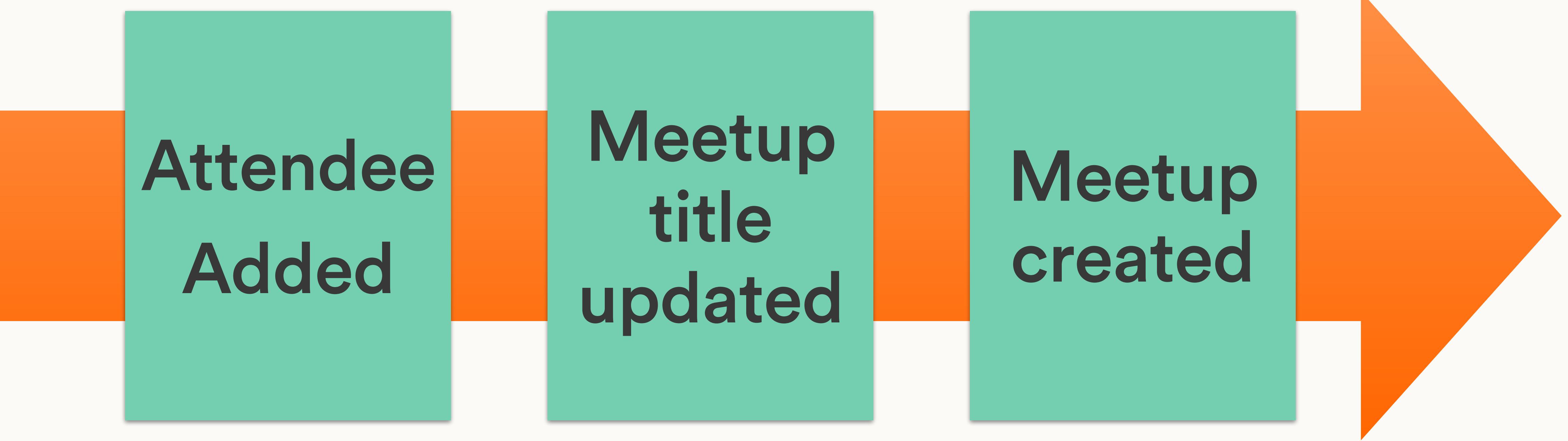


**Meetup
created**



Meetup
title
updated

Meetup
created



**Attendee
Added**

**Meetup
title
updated**

**Meetup
created**

“”

**Event Sourcing ensures that
all changes to application
state are stored as a sequence
of events**

Martin Fowler - <https://www.martinfowler.com/eaaDev/EventSourcing.html>

Event Sourcing

Event Sourcing

Immutable

Event Sourcing

Immutable

Thy shall never delete thing from the event store.
It is append-only

Event Sourcing

Immutable

Thy shall never delete thing from the event store.
It is append-only

Single source of truth

Event Sourcing

Immutable

Thy shall never delete thing from the event store.
It is append-only

Single source of truth

$$\text{current_state} = \sum \text{events}$$

Event Sourcing

Immutable

Thy shall never delete thing from the event store.
It is append-only

Single source of truth

$$\text{current_state} = \sum \text{events}$$

Be careful with snapshots

Event Sourcing

Immutable

Thy shall never delete thing from the event store.
It is append-only

Single source of truth

$$\text{current_state} = \sum \text{events}$$

Be careful with snapshots

Fine-tune your snapshotting logic (race conditions)

Event Sourcing

Event Sourcing

Time travel

Project data from the event store to any point in time in the history of the system. Business will love you for it

Event Sourcing

Time travel

Project data from the event store to any point in time in the history of the system. Business will love you for it

Audit log

The event store is an immutable, append-only datastore

Event Sourcing

Time travel

Project data from the event store to any point in time in the history of the system. Business will love you for it

Audit log

The event store is an immutable, append-only datastore

Eventual consistency

There is UX workarounds to deal with it, but this is going to be something to accept.

Event Sourcing

Time travel

Project data from the event store to any point in time in the history of the system. Business will love you for it

Audit log

The event store is an immutable, append-only datastore

Eventual consistency

There is UX workarounds to deal with it, but this is going to be something to accept.



Eating &c					
1832					
Oct 1	Ginger Beer		" "		
5	A brace of Grouse a la f.	16	" }		
"	Packing &c of do	3	" }	10	
Dec 11	Dinner at Club		" 2		
"	Coffee		" "		
12	Breakfast		" 1		
13	Breakfast		" 1		
"	Tea		" "		
14	Breakfast		" 1		
15	Breakfast		" 1		
1833					
Jan 20	Tea at Union Club		" 1		
29	Breakfast		" 1		
"	Soup		" 1		
Feb 19	Soda Water		" "		
23	Oranges		" "		
March 22	Two Jupubes	3	" 1		
April 30	Bundle of Asparagus		" "		
May 1 st	Breakfast	16	" 1		
"	Waiter	6	" 1		
14	Ices &c		" "		
June 1	Ices		" "		

CQRS/ES: when is it a good idea?

Reads > Writes, Writes > Reads

When the amount of reads you need to perform is far greater than the amount of writes. Or vice-versa.

When you have a complex domain to model

It's easier to reason about a series of events when modelling a complex domain (insurance policy changes, e-commerce, banking transactions...)

When your system can't afford to lose data

Your industry is heavily regulated, or you're dealing with critical data for your customers

CQRS/ES & Elixir

Command |> Aggregate |> Projection(s)

Dashboard

Queue Name	Length		Rate
	Current	Peak	
Index Committer	0	0	
MainQueue	1	5	
MonitoringQueue	0	1	
Projection Core	+ 0	0	
Projections Master	0	3	
Storage Chaser	0	0	
StorageReaderQueue	+ 0	1	
StorageWriterQueue	0	2	
Subscriptions	0	1	
Timer	7	9	
Workers	+ 0	0	

Connections (0)

Connection	Client Connection Name	Type
No Connections		

Choose an Event Store database

Eventstore.com

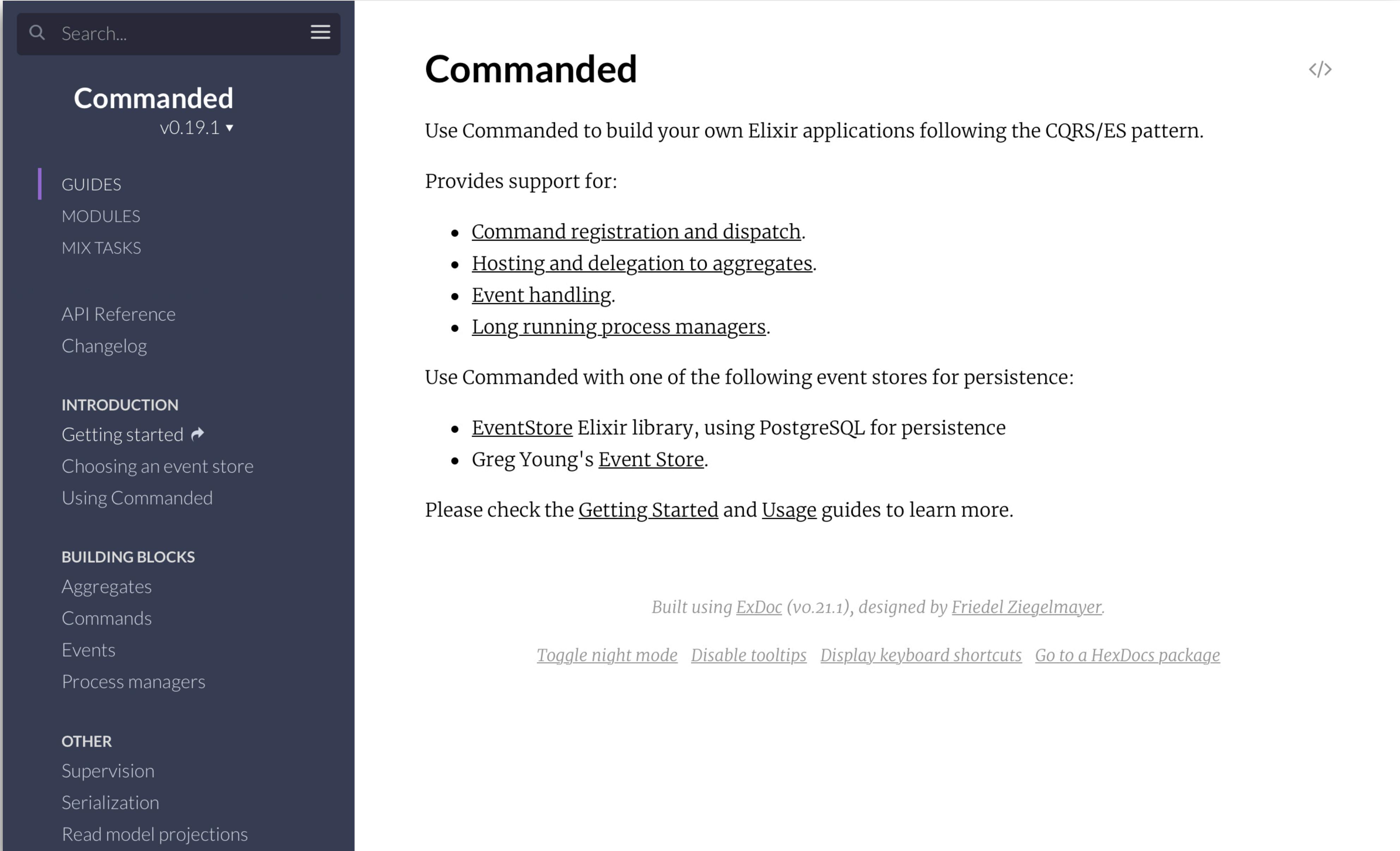
Company founded by Greg Young, it's a database built specifically for event sourcing.

Plain old Postgres 

Postgres is more than enough to have as an event store. There is an event store library implemented on top of the *Postgrex Client*.

```
{:eventstore, "~> 0.17.0"}
```

Commanded, a framework for CQRS/ES in Elixir



The screenshot shows the Commanded HexDocs page. The left sidebar has a dark blue background with white text. It includes a search bar at the top, followed by the title "Commanded" and version "v0.19.1". Below this are sections for "GUIDES", "MODULES", "MIX TASKS", "API Reference", and "Changelog". Under "INTRODUCTION", there are links for "Getting started", "Choosing an event store", and "Using Commanded". Under "BUILDING BLOCKS", there are links for "Aggregates", "Commands", "Events", and "Process managers". Under "OTHER", there are links for "Supervision", "Serialization", and "Read model projections". The main content area on the right has a white background. It features a large heading "Commanded" and a sub-heading "Use Commanded to build your own Elixir applications following the CQRS/ES pattern." It lists "Provides support for:" with four bullet points: "Command registration and dispatch.", "Hosting and delegation to aggregates.", "Event handling.", and "Long running process managers.". It also mentions "Use Commanded with one of the following event stores for persistence:" with two bullet points: "EventStore Elixir library, using PostgreSQL for persistence" and "Greg Young's Event Store.". A note at the bottom says "Please check the [Getting Started](#) and [Usage](#) guides to learn more." At the very bottom, it says "Built using [ExDoc](#) (v0.21.1), designed by Friedel Ziegelmayer." and provides links for "Toggle night mode", "Disable tooltips", "Display keyboard shortcuts", and "Go to a HexDocs package".

Commanded

Use Commanded to build your own Elixir applications following the CQRS/ES pattern.

Provides support for:

- [Command registration and dispatch.](#)
- [Hosting and delegation to aggregates.](#)
- [Event handling.](#)
- [Long running process managers.](#)

Use Commanded with one of the following event stores for persistence:

- [EventStore](#) Elixir library, using PostgreSQL for persistence
- [Greg Young's Event Store.](#)

Please check the [Getting Started](#) and [Usage](#) guides to learn more.

Built using [ExDoc](#) (v0.21.1), designed by [Friedel Ziegelmayer](#).

[Toggle night mode](#) [Disable tooltips](#) [Display keyboard shortcuts](#) [Go to a HexDocs package](#)



```
defmodule CqrsDemo.Meetups.Aggregates.Meetup do
  defstruct [:uuid, :title, :date, :attendees, :host]
  alias __MODULE__
  alias CqrsDemo.Meetups.Commands.CreateMeetup
  alias CqrsDemo.Meetups.Events.MeetupCreated

  # Executing commands
  def execute(%Meetup{uuid: nil}, %CreateMeetup{uuid: nil})
    %MeetupCreated{
      uuid: uuid,
      title: title,
      date: date
    }
  end

  def execute(%Meetup{}, %CreateMeetup{}), do: {:error, "Meetup already exists"}

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, title: title, date: date})
    %Meetup{
      uuid: uuid,
      title: title,
      date: date
    }
  end
end
```



```
defmodule CqrsDemo.Meetups.Aggregates.Meetup do
  defstruct [:uuid, :title, :date, :attendees, :host]
  alias __MODULE__
  alias CqrsDemo.Meetups.Commands.CreateMeetup
  alias CqrsDemo.Meetups.Events.MeetupCreated

  # Executing commands
  def execute(%Meetup{uuid: nil}, %CreateMeetup{uuid: nil})
    %MeetupCreated{
      uuid: uuid,
      title: title,
      date: date
    }
  end

  def execute(%Meetup{}, %CreateMeetup{}), do: {:error, "Meetup already exists"}

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, title: title, date: date})
    %Meetup{
      uuid: uuid,
      title: title,
      date: date
    }
  end
end
```

Aggregates



```
defmodule CqrsDemo.Meetups.Aggregates.Meetup do
  defstruct [:uuid, :title, :date, :attendees, :host]
  alias __MODULE__
  alias CqrsDemo.Meetups.Commands.CreateMeetup
  alias CqrsDemo.Meetups.Events.MeetupCreated

  # Executing commands
  def execute(%Meetup{uuid: nil}, %CreateMeetup{uuid: nil, title: title, date: date})
    %MeetupCreated{
      uuid: uuid,
      title: title,
      date: date
    }
  end

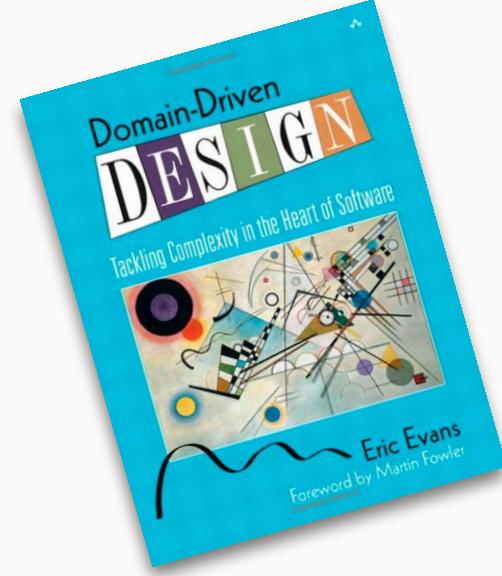
  def execute(%Meetup{}, %CreateMeetup{}), do: error()

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, title: title, date: date})
    %Meetup{
      uuid: uuid,
      title: title,
      date: date
    }
  end
end
```

Aggregates

It's a cluster of entities

It's a GenServer process. It can have a lifecycle.





```
defmodule CqrsDemo.Meetups.Aggregates.Meetup do
  defstruct [:uuid, :title, :date, :attendees, :host]
  alias __MODULE__
  alias CqrsDemo.Meetups.Commands.CreateMeetup
  alias CqrsDemo.Meetups.Events.MeetupCreated

  # Executing commands
  def execute(%Meetup{uuid: nil}, %CreateMeetup{uuid: nil, title: title, date: date}) do
    MeetupCreated{
      uuid: uuid,
      title: title,
      date: date
    }
  end

  def execute(%Meetup{}, %CreateMeetup{}), do: {:error, "Meetup already exists"}

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, title: title, date: date}) do
    Meetup{
      uuid: uuid,
      title: title,
      date: date
    }
  end
end
```

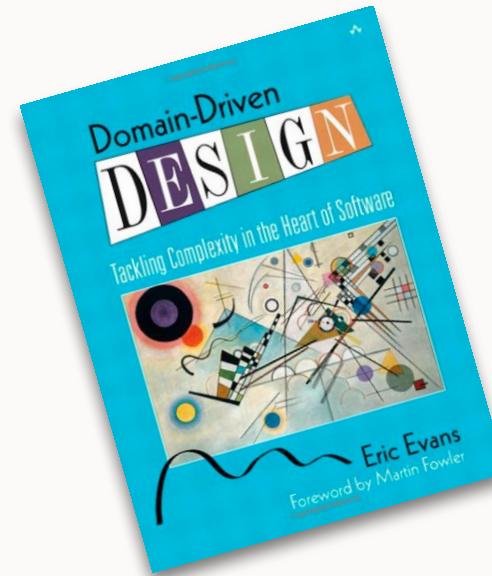
Aggregates

It's a cluster of entities

It's a GenServer process. It can have a lifecycle.

It handles Commands

The execute/2 function returns 1 or more domain events.





```
defmodule CqrsDemo.Meetups.Aggregates.Meetup do
  defstruct [:uuid, :title, :date, :attendees, :host]
  alias __MODULE__
  alias CqrsDemo.Meetups.Commands.CreateMeetup
  alias CqrsDemo.Meetups.Events.MeetupCreated

  # Executing commands
  def execute(%Meetup{uuid: nil}, %CreateMeetup{uuid: nil, title: title, date: date}) do
    %MeetupCreated{
      uuid: uuid,
      title: title,
      date: date
    }
  end

  def execute(%Meetup{}, %CreateMeetup{}), do: {:error, "Meetup already exists"}

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, title: title, date: date}) do
    %Meetup{
      uuid: uuid,
      title: title,
      date: date
    }
  end
end
```

Aggregates

It's a cluster of entities

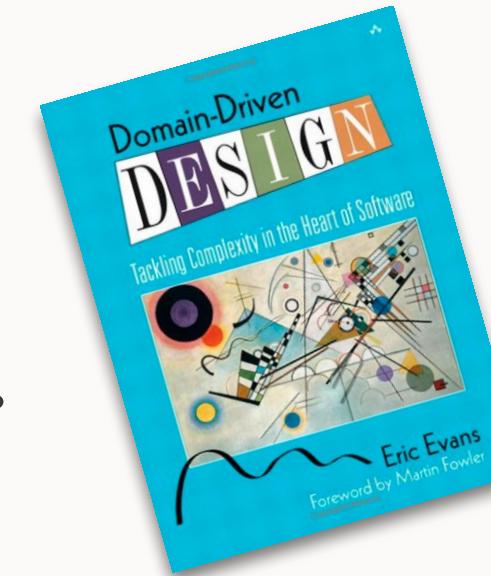
It's a GenServer process. It can have a lifecycle.

It handles Commands

The execute/2 function returns 1 or more domain events.

It persists events to the Event Store

The apply/2 function returns the updated state of the aggregate.





```
defmodule CqrsDemo.Meetu
  @enforce_keys [:uuid,
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```



Commands

```
defmodule CqrsDemo.Meetu
  @enforce_keys [:uuid,
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```



```
defmodule CqrsDemo.Meetu
  @enforce_keys [:uuid,
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```

Commands

It's just a struct

It holds the action & its input to be handled by the aggregate. The Router is responsible to dispatch commands to the correct aggregate



```
defmodule CqrsDemo.Meetu
  @enforce_keys [:uuid,
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```

Commands

It's just a struct

It holds the action & its input to be handled by the aggregate. The Router is responsible to dispatch commands to the correct aggregate

We have 2 consistency guarantees

We can choose between :eventual and :strong consistency



```
defmodule CqrsDemo.Meetup
  @derive Jason.Encoder
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```



Domain Events

```
defmodule CqrsDemo.Meetup
  @derive Jason.Encoder
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```



Domain Events

Elixir struct

```
defmodule CqrsDemo.Meeting
  @derive Jason.Encoder
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```



```
defmodule CqrsDemo.Meeting
  @derive Jason.Encoder
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```

Domain Events

Elixir struct

Just a struct, encoded in JSON.

Always make sure that JSON serialisation is supported



Domain Events

```
defmodule CqrsDemo.Meeting
  @derive Jason.Encoder
  defstruct [
    :uuid,
    :title,
    :date
  ]
end
```

Elixir struct

Just a struct, encoded in JSON.

Always make sure that JSON serialisation is supported

Otherwise they won't be de-serialisable by the event handlers or projectors



```
defmodule ExampleHandler
  use Commanded.Event.Handler

  def handle(%AnEvent{ ... })
    # ... process the event ...
    :ok
  end
end
```



Event handlers & process managers

```
defmodule ExampleHandler
  use Commanded.Event.Handler

  def handle(%AnEvent{ ... })
    # ... process the event ...
    :ok
  end
end
```



Event handlers & process managers

```
defmodule ExampleHandler
  use Commanded.Event.Handler

  def handle(%AnEvent{ ... })
    # ... process the event ...
    :ok
  end
end
```

Event handlers

Event handlers apply any side effect caused by a domain event (projection, sending an email, etc.).



Event handlers & process managers

```
defmodule ExampleHandler
  use Commanded.Event.Handler

  def handle(%AnEvent{ ... })
    # ... process the event
    :ok
  end
end
```

Event handlers

Event handlers apply any side effect caused by a domain event (projection, sending an email, etc.).

Process managers

Process managers dispatch commands between aggregates in response to events.



Event handlers & process managers

```
defmodule ExampleHandler
  use Commanded.Event.Handler

  def handle(%AnEvent{ ... })
    # ... process the event
    :ok
  end
end
```

Event handlers

Event handlers apply any side effect caused by a domain event (projection, sending an email, etc.).

Process managers

Process managers dispatch commands between aggregates in response to events.

Both are supervised processes

Be considerate about the retry strategies (too many restarts can cause the app to crash)

DEMO!

#YOLO

impraise



We're hiring!

Come see at
jobs.impraise.com

Elixir Engineer 

Join us in our mission to grow teams & managers
into their best self

DevOps engineer 

Help us practice DevOps & GitOps across the
whole company

We're looking for guinea- pigs



- Managers
- Team members

User interviews

Help us test our prototypes by joining our pool of test users

Beta testers

Give us feedback on our work so we can improve it

<https://tinyurl.com/impraise>



We're looking for guinea- pigs



- Managers
- Team members

User interviews

Help us test our prototypes by joining our pool of test users

Beta testers

Give us feedback on our work so we can improve it

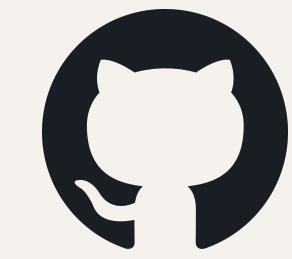
<https://tinyurl.com/impraise>



Sources:

1. www.commanded.io
2. eventstore.org
3. martinfowler.com
4. eventstorming.com

Thank you!



@bdubaut