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

Domain-driven Design

Let's start at the beginning

impraise

Domain Driven Design

Bounded contexts

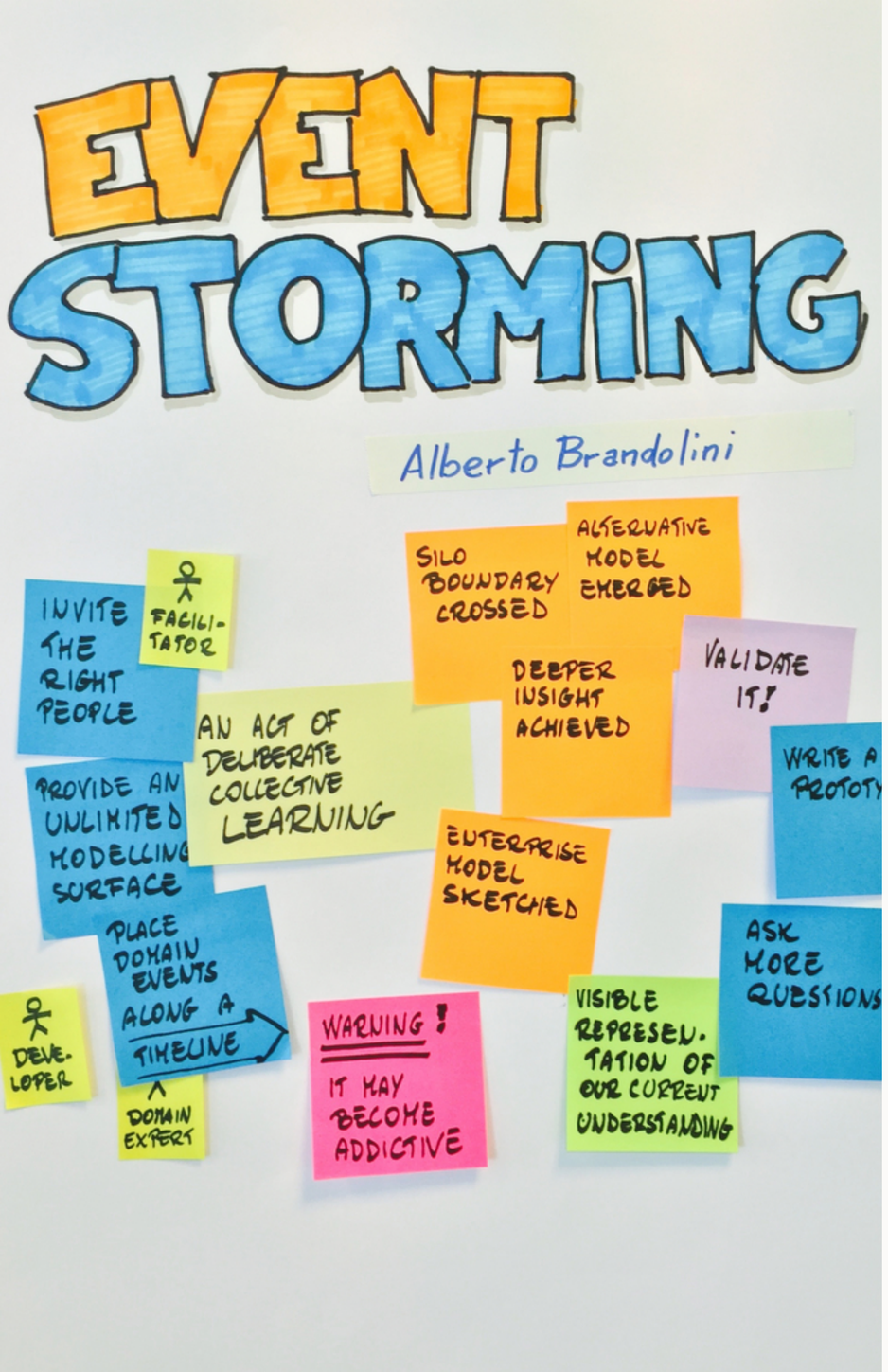
```
mix phx.gen.context Meetups meetup meetup
```

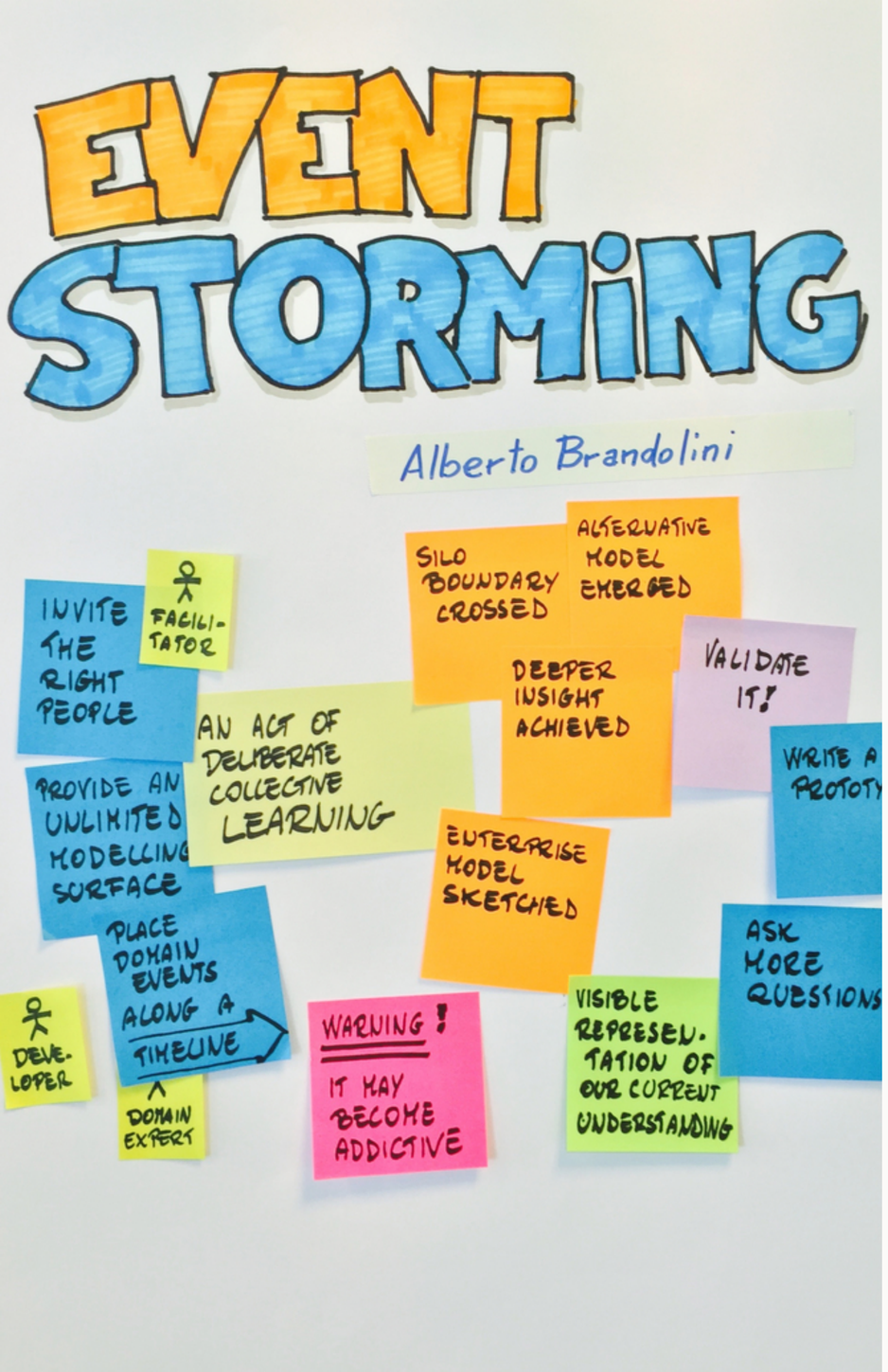
Aggregate

“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.

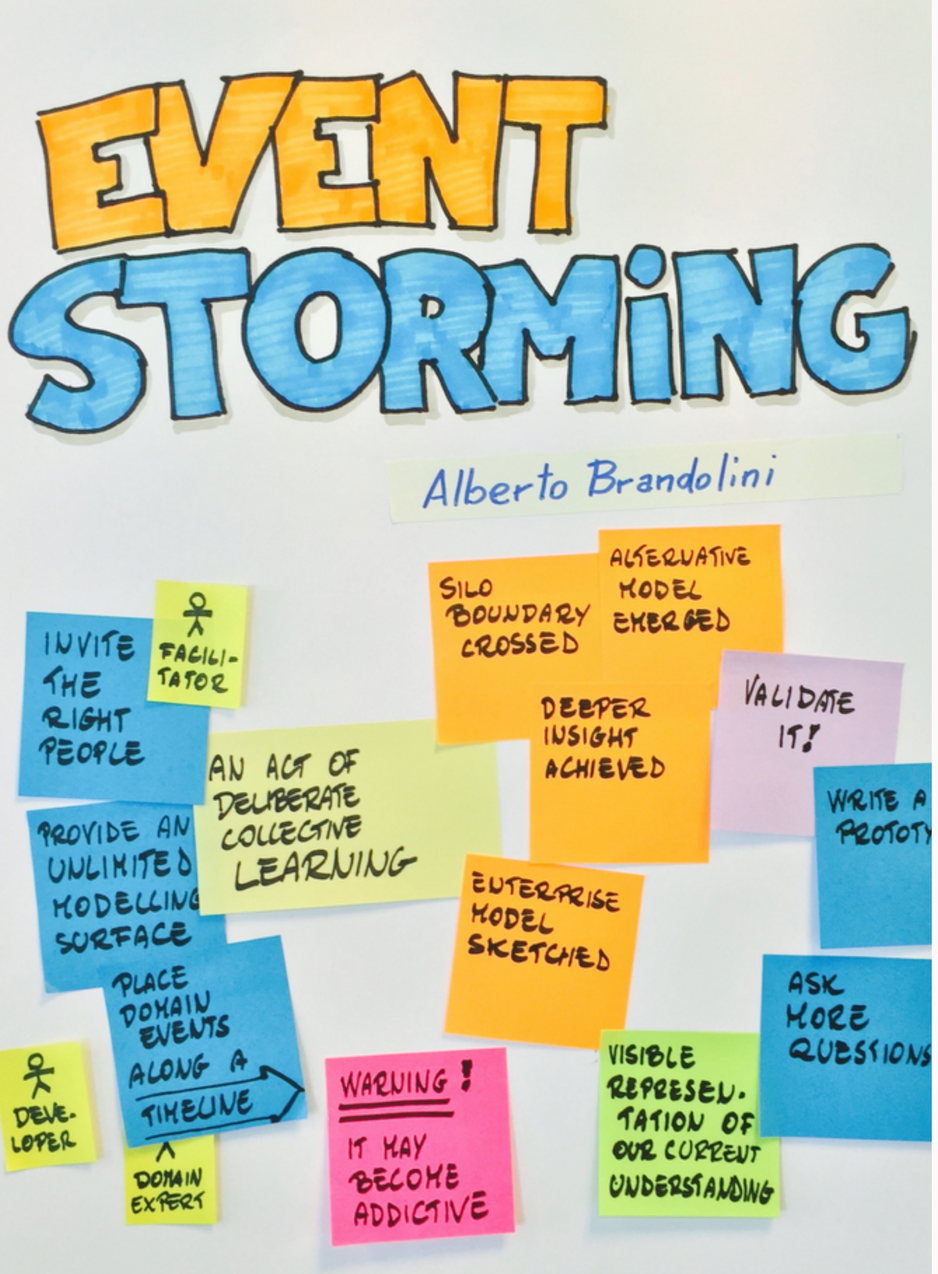

EVENT STORMING
Alberto Brandolini
INVITE THE RIGHT PEOPLE
FACILI-TATOR
AN ACT OF DELIBERATE COLLECTIVE LEARNING
PROVIDE AN UNLIMITED MODELLING SURFACE
PLACE DOMAIN EVENTS ALONG A TIMELINE
DEVE-LOPER
DOMAIN EXPERT
SILO BOUNDARY CROSSED
ALTERNATIVE MODEL EMERGED
DEEPER INSIGHT ACHIEVED
VALIDATE IT!
WRITE A PROTOTY
ENTERPRISE MODEL SKETCHED
WARNING!
IT MAY BECOME ADDICTIVE
VISIBLE REPRESEN-TATION OF OUR CURRENT UNDERSTANDING
ASK MORE QUESTIONS



Domain modelling tips

Buy Eric Evans' book

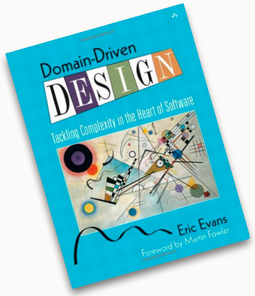
"Domain-Driven Design - Tackling complexity in the heart of software"

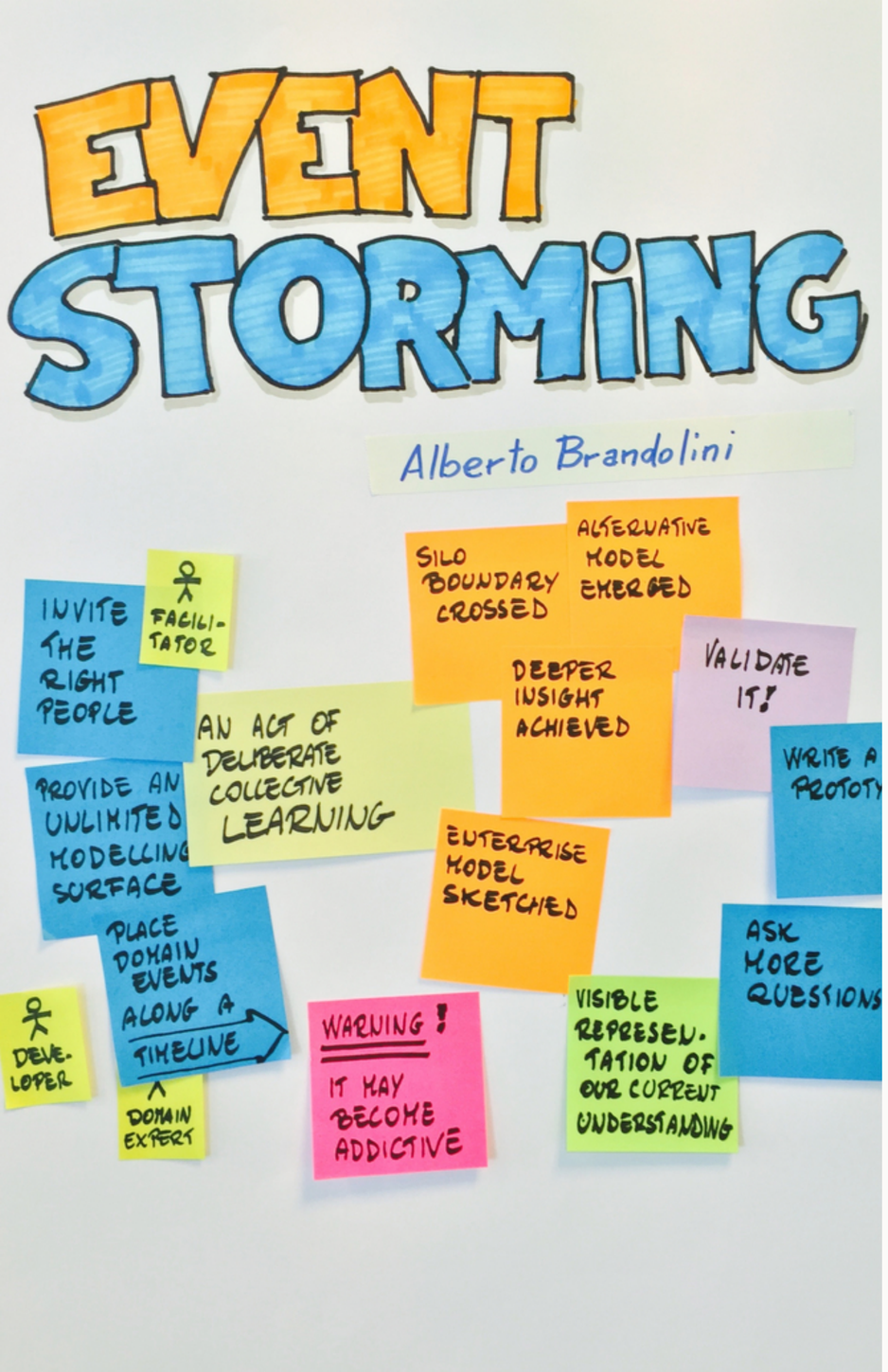


Domain modelling tips

Buy Eric Evans' book

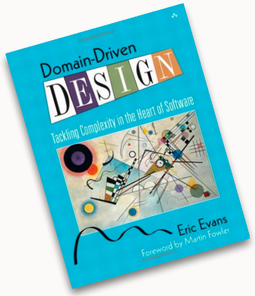
"Domain-Driven Design - Tackling complexity in the heart of software"





Domain modelling tips

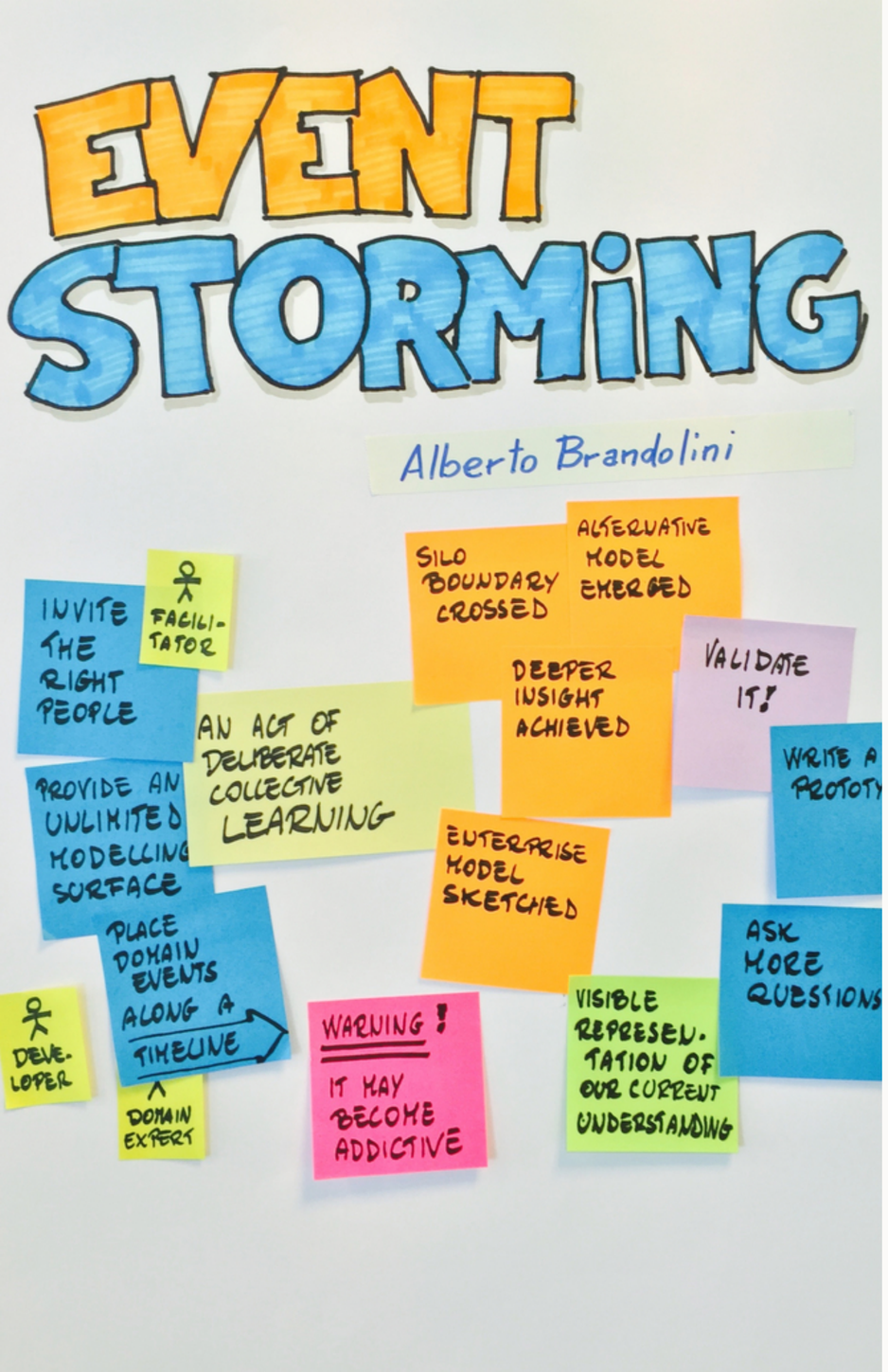
Buy Eric Evans' book



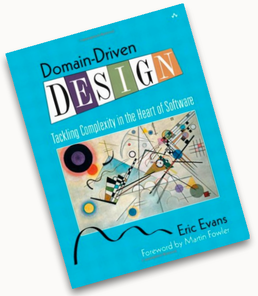
"Domain-Driven Design - Tackling complexity in the heart of software"

Involve EVERYONE

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



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

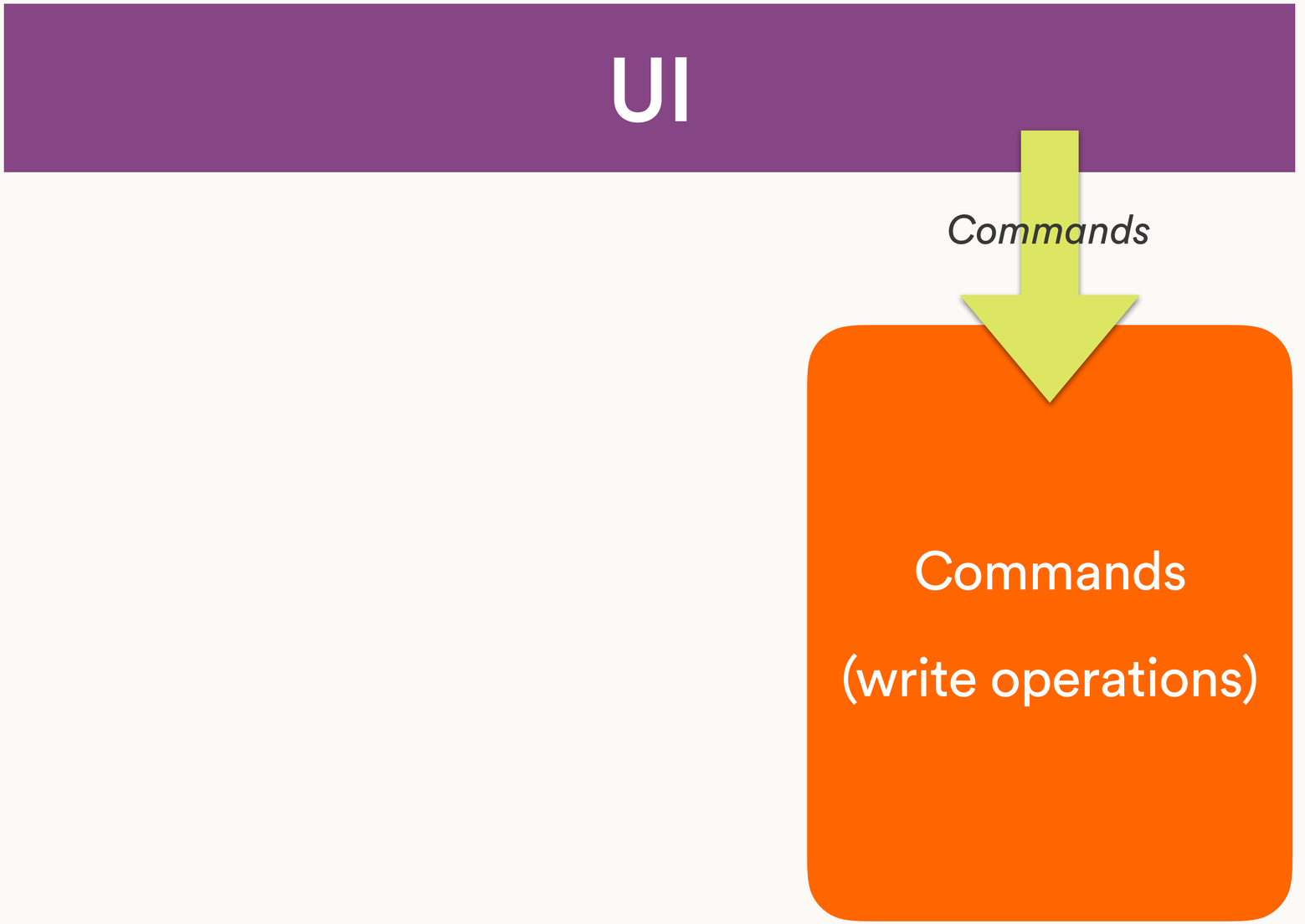
Get post-its. Then get more post-its. Put relevant business events over a timeline, then isolate your commands, read models, and your domain model appears in front of your very eyes.

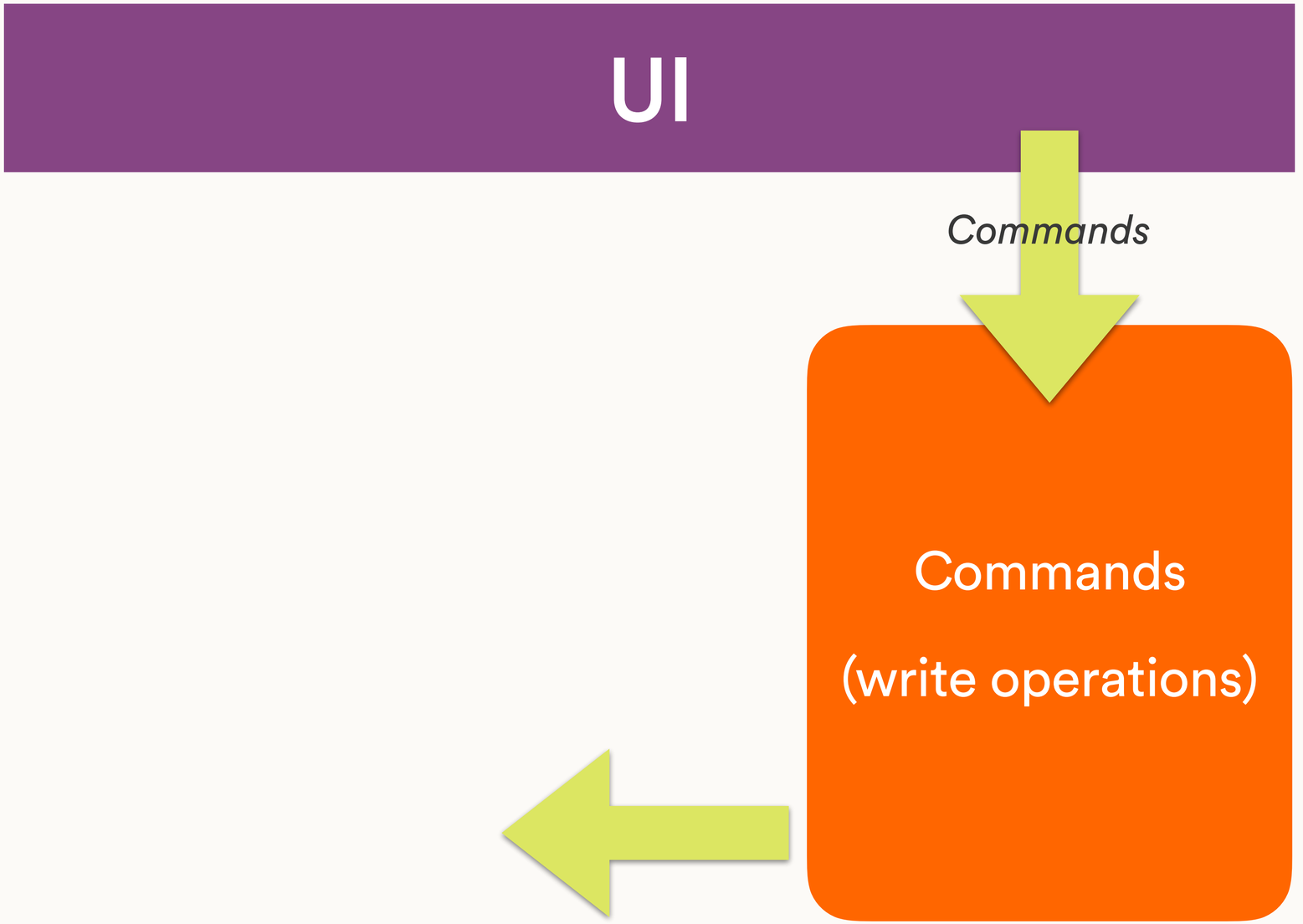
CQRS

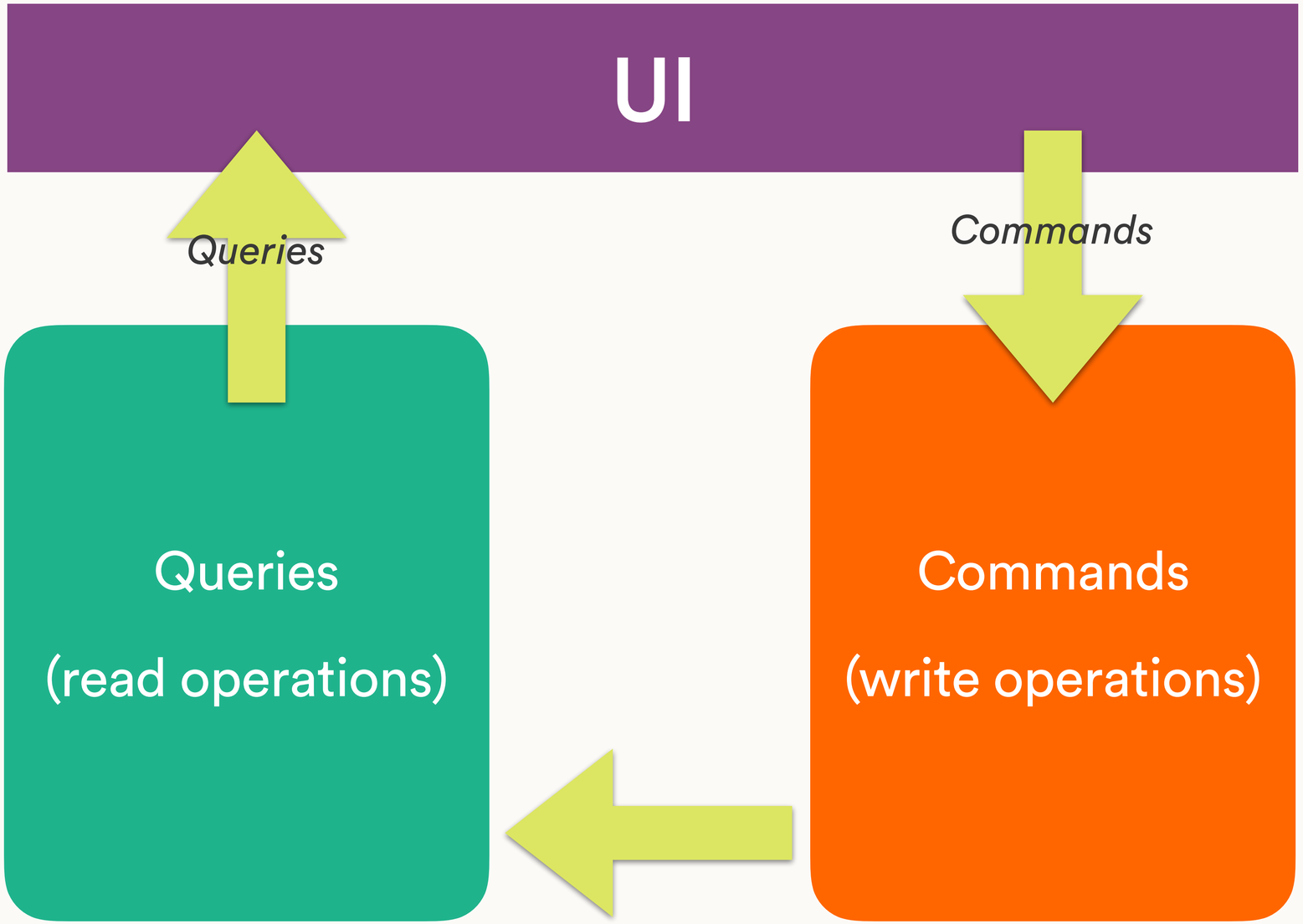
A natural evolution of DDD

impraise

UI


UI
Commands
Commands
(write operations)


UI
Commands
Commands
(write operations)


UI
Queries
Commands
Queries
(read operations)
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

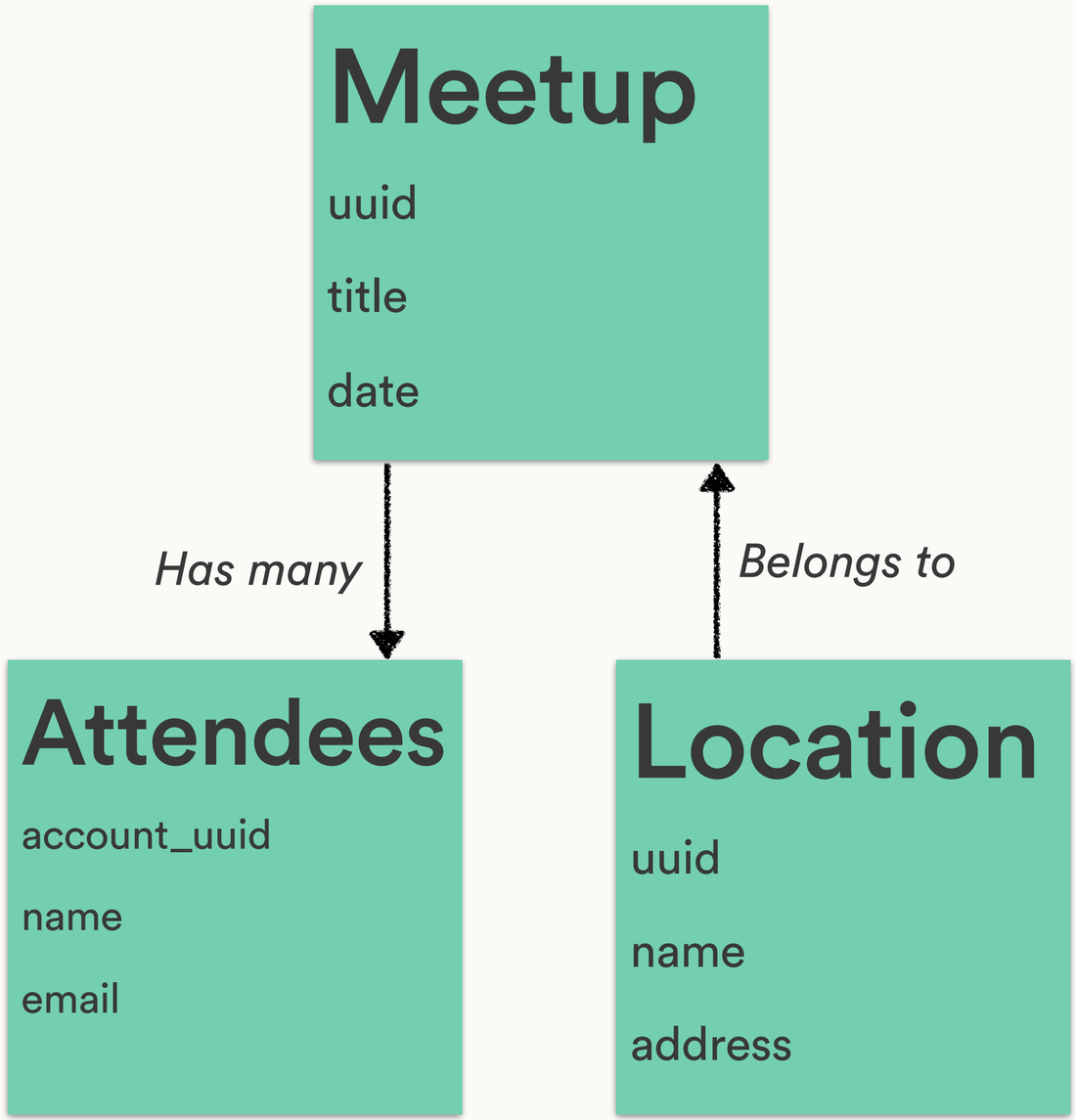
Extra complexity

You now have 2 models to understand and maintain.

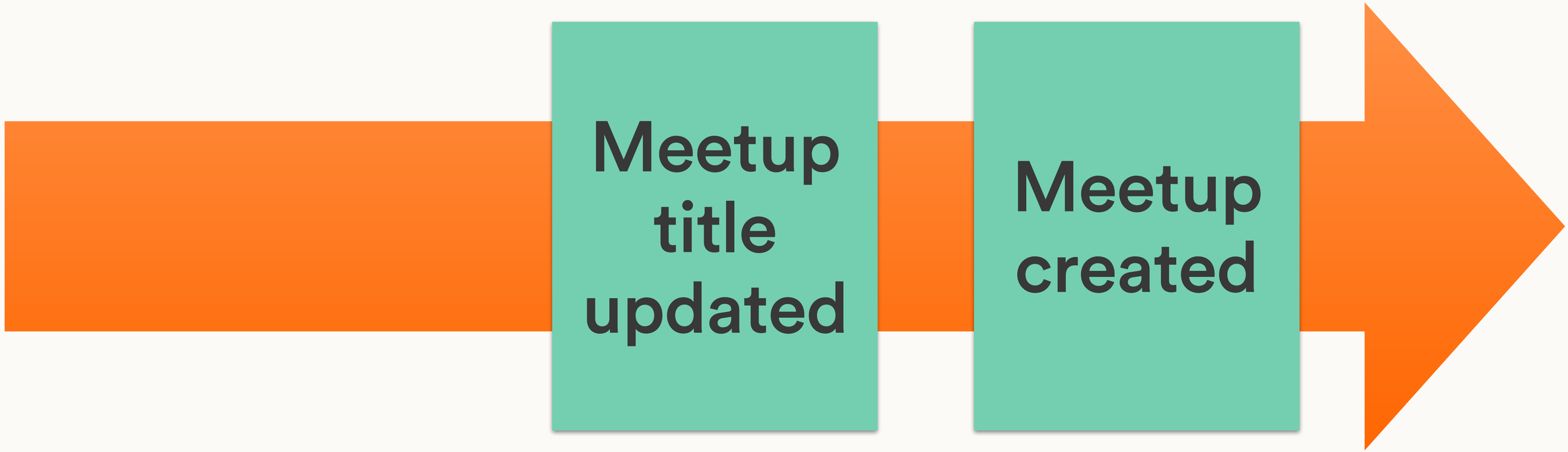
Event Sourcing

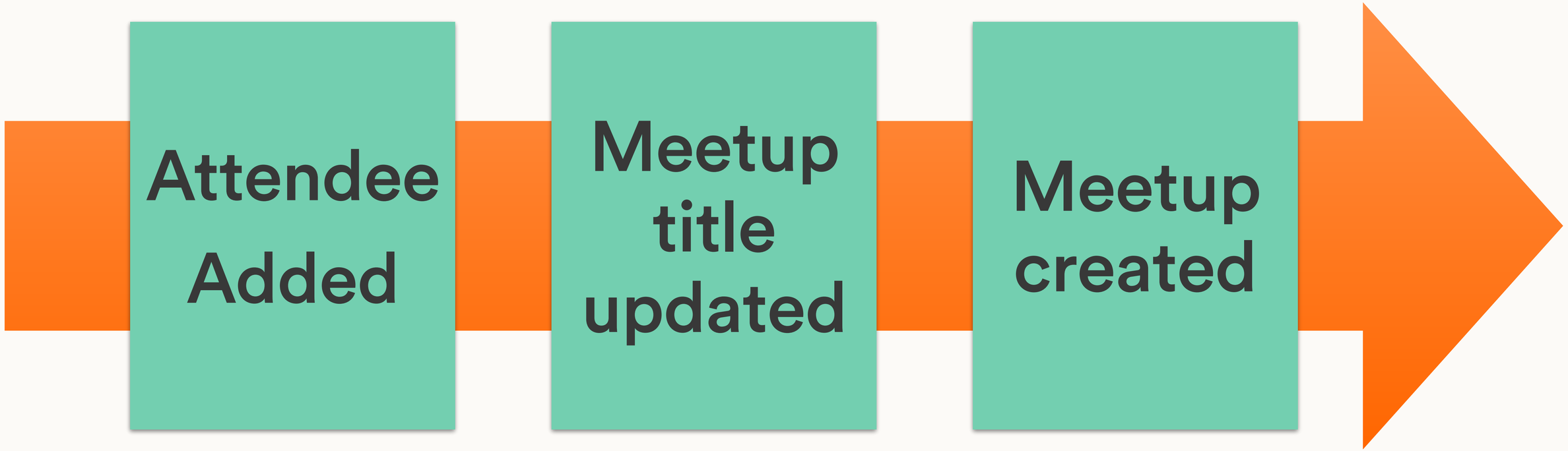
Not quite a new concept

impraise


Meetup
uuid
title
date
Has many
Belongs to
Attendees
account_uuid
name
email
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

$$current\_state = \sum events$$

Event Sourcing

Immutable

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

Single source of truth

$$current\_state = \sum events$$

Be careful with snapshots

Event Sourcing

Immutable

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

Single source of truth

$$current\_state = \sum events$$

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

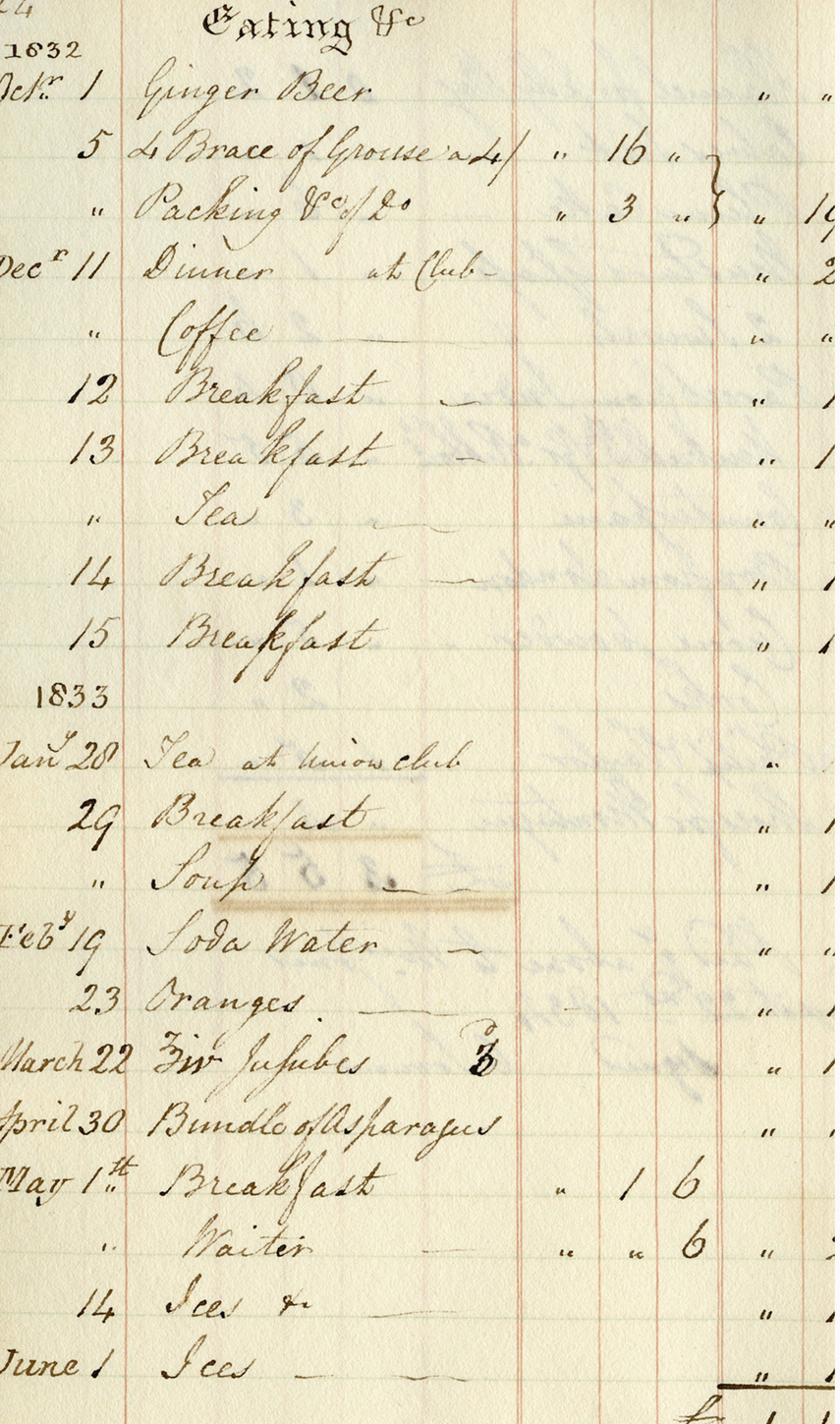
Project data from the event store to any point in time in the history of the system. Business will love you for it

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.



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

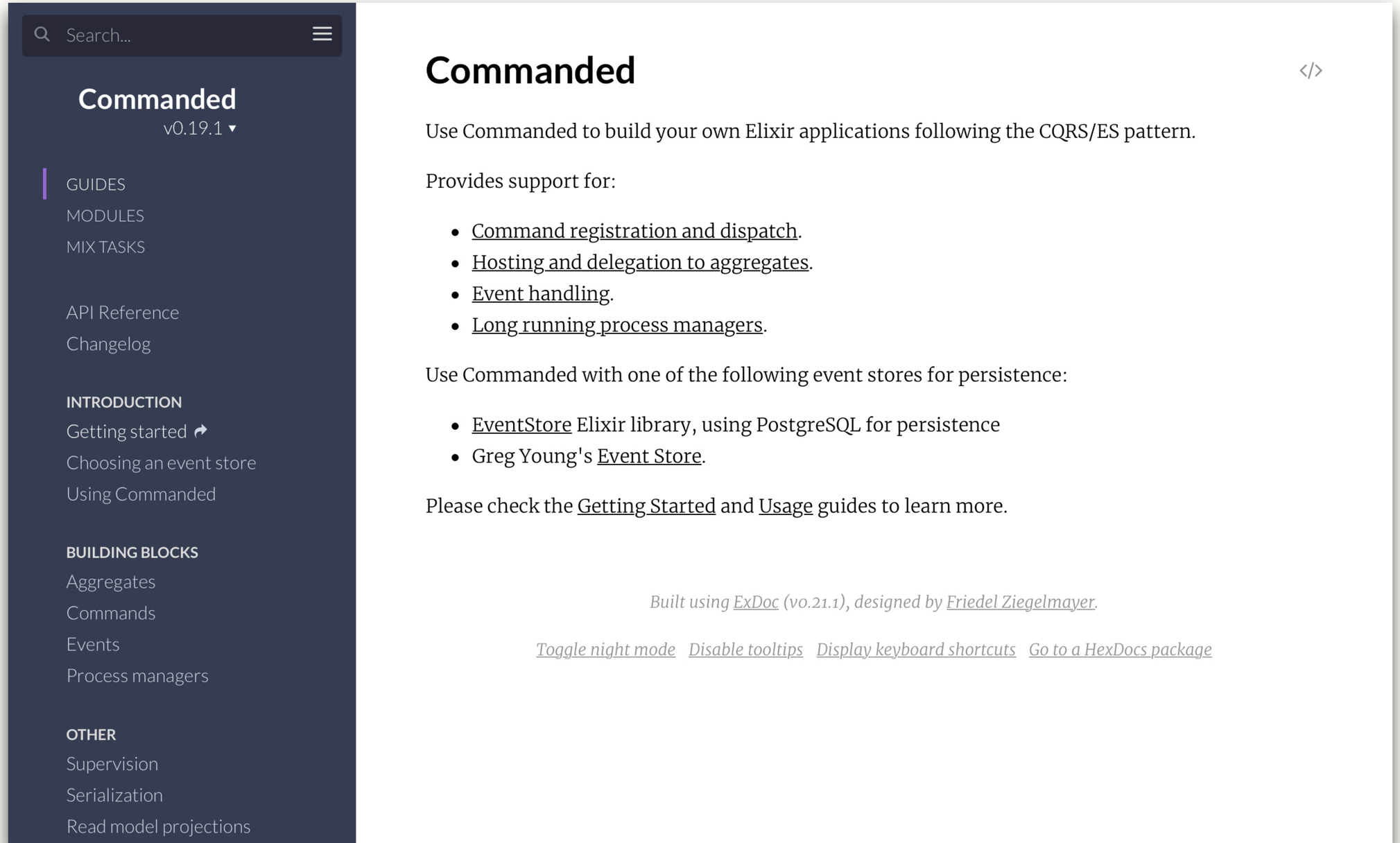
Company founded by Greg Young, it's a database built specifically for event sourcing.

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



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

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

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

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, t
    %Meetup{
      uuid: uuid,
      title: title,
      date: date
    }
  end
end
```

Aggregates

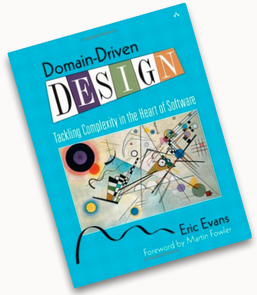
```elixir
defmodule CqrsDemo.Meetups.Aggregates.Meetup do
  defstruct [:uuid, :title, :date, :attendees, :hos
  alias __MODULE__
  alias CqrsDemo.Meetups.Commands.CreateMeetup
  alias CqrsDemo.Meetups.Events.MeetupCreated

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

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

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, t
    %Meetup{
      uuid: uuid,
      title: title,
      date: date
    }
  end
end
```

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

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

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

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, t
    %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.

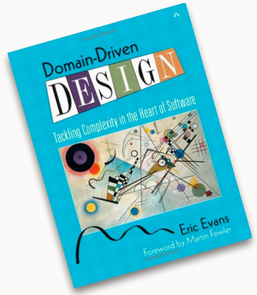


```elixir
defmodule CqrsDemo.Meetups.Aggregates.Meetup do
  defstruct [:uuid, :title, :date, :attendees, :hos
  alias __MODULE__
  alias CqrsDemo.Meetups.Commands.CreateMeetup
  alias CqrsDemo.Meetups.Events.MeetupCreated

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

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

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, t
    %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.

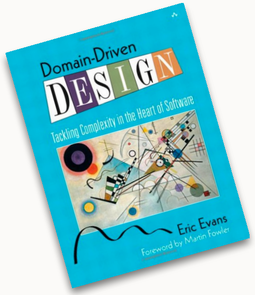


It handles Commands

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

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

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

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

  # Mutating the aggregate's state
  def apply(%Meetup{}, %MeetupCreated{uuid: uuid, t
    %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.

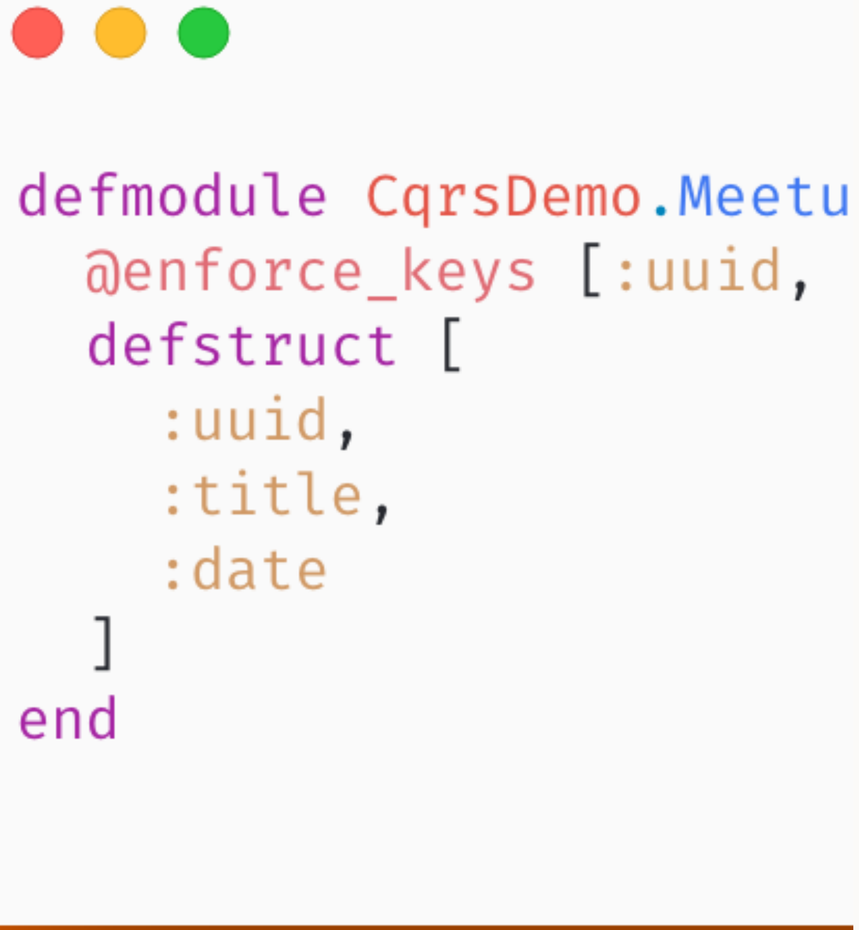
It persists events to the Event Store

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

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

Commands

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



```elixir
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

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

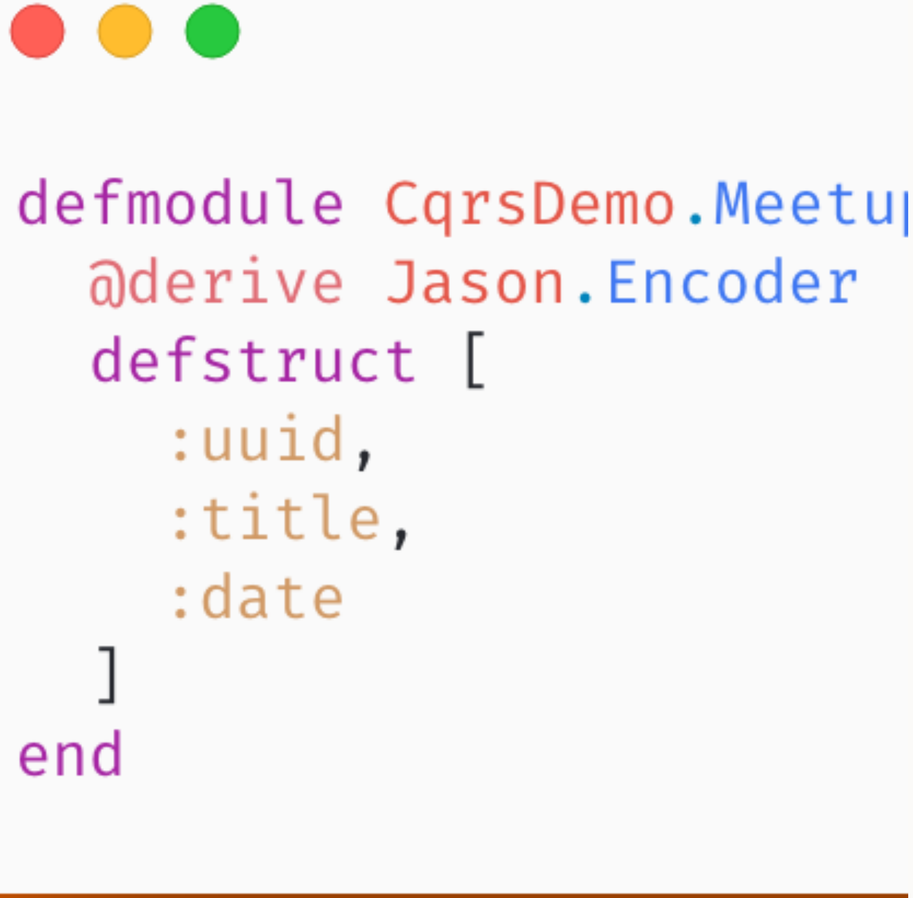
Commands

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



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

Domain Events

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

Domain Events

Elixir struct

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

```elixir
defmodule CqrsDemo.Meetu
  @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

```
defmodule CqrsDemo.Meetu
  @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

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


defmodule ExampleHandler
use Commanded.Event.Ha
def handle(%AnEvent{ ..
... process the ev
:ok
end
end

Event handlers & process managers

```elixir
defmodule ExampleHandler
  use Commanded.Event.Ha

  def handle(%AnEvent{ ..
    # ... process the ev
    :ok
  end
end
```

Event handlers & process managers

Event handlers

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

Event handlers & process managers

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

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