CQRS & ES

BERNARDO AMORIM

github.com/bamorim

CHECK OUT THE CODE AT

GITHUB.COM/BAMORIM/EX-CBSF2018

EVENT SOURCING

SAVE EVENTS

NOT THE FINAL STATE

AN EVENT IS SOMETHING THAT HAPPENED, A FACT

FundsAdded
account_id: 123
amount: 100

YOUR STATE IS A FIRST-LEVEL DERIVATIVE OF YOUR FACTS

AccountOpened

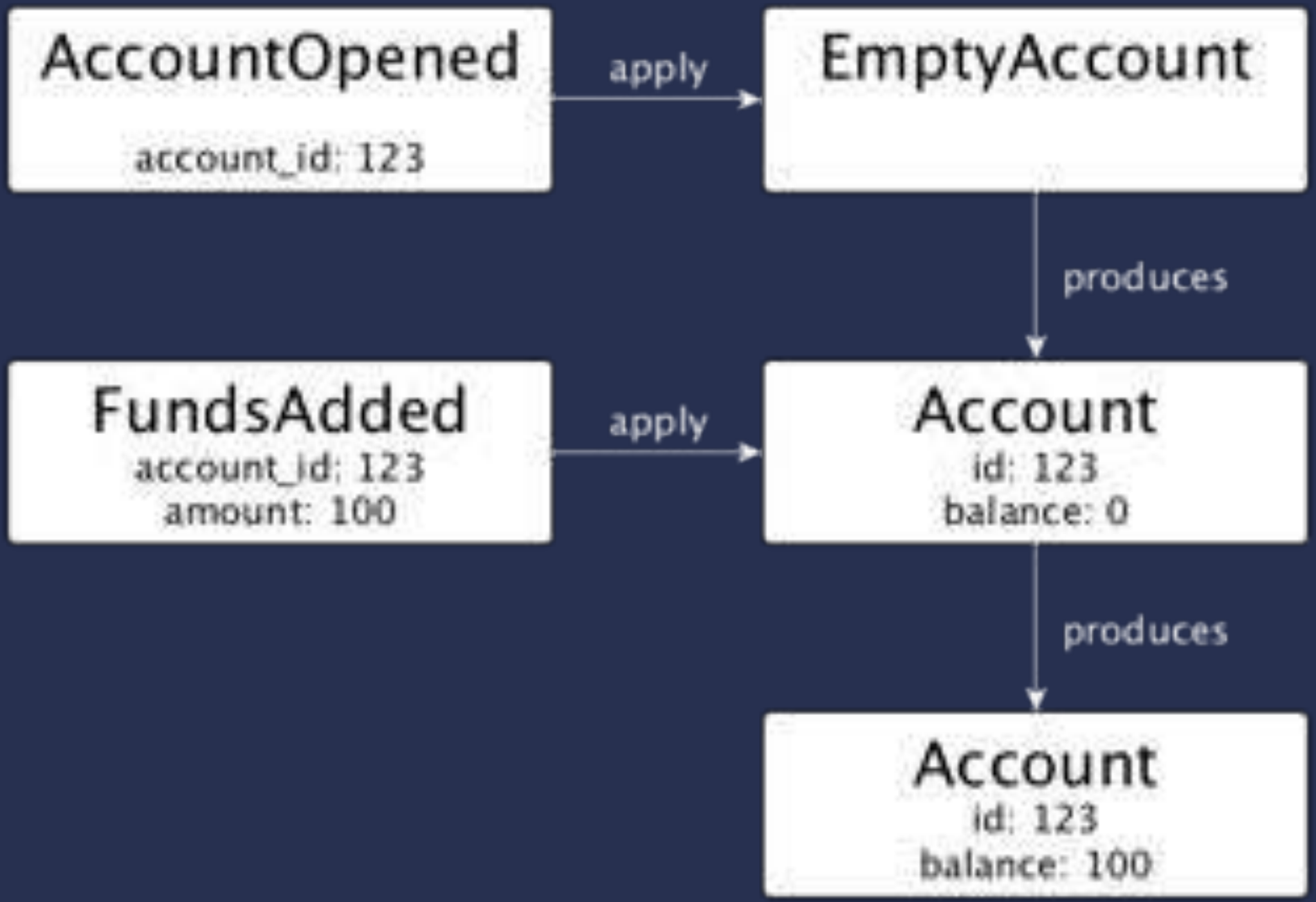
account_id: 123

FundsAdded

account_id: 123
amount: 100

FundsAdded

account_id: 123
amount: 50


AccountOpened
account_id: 123
apply
EmptyAccount
produces
FundsAdded
account_id: 123
amount: 100
apply
Account
id: 123
balance: 0
produces
Account
id: 123
balance: 100

$$S_n = \text{apply}(S_{n-1}, E_n)$$

$$S_n = \mathrm{reduce}(E, S_0, \mathrm{apply})$$

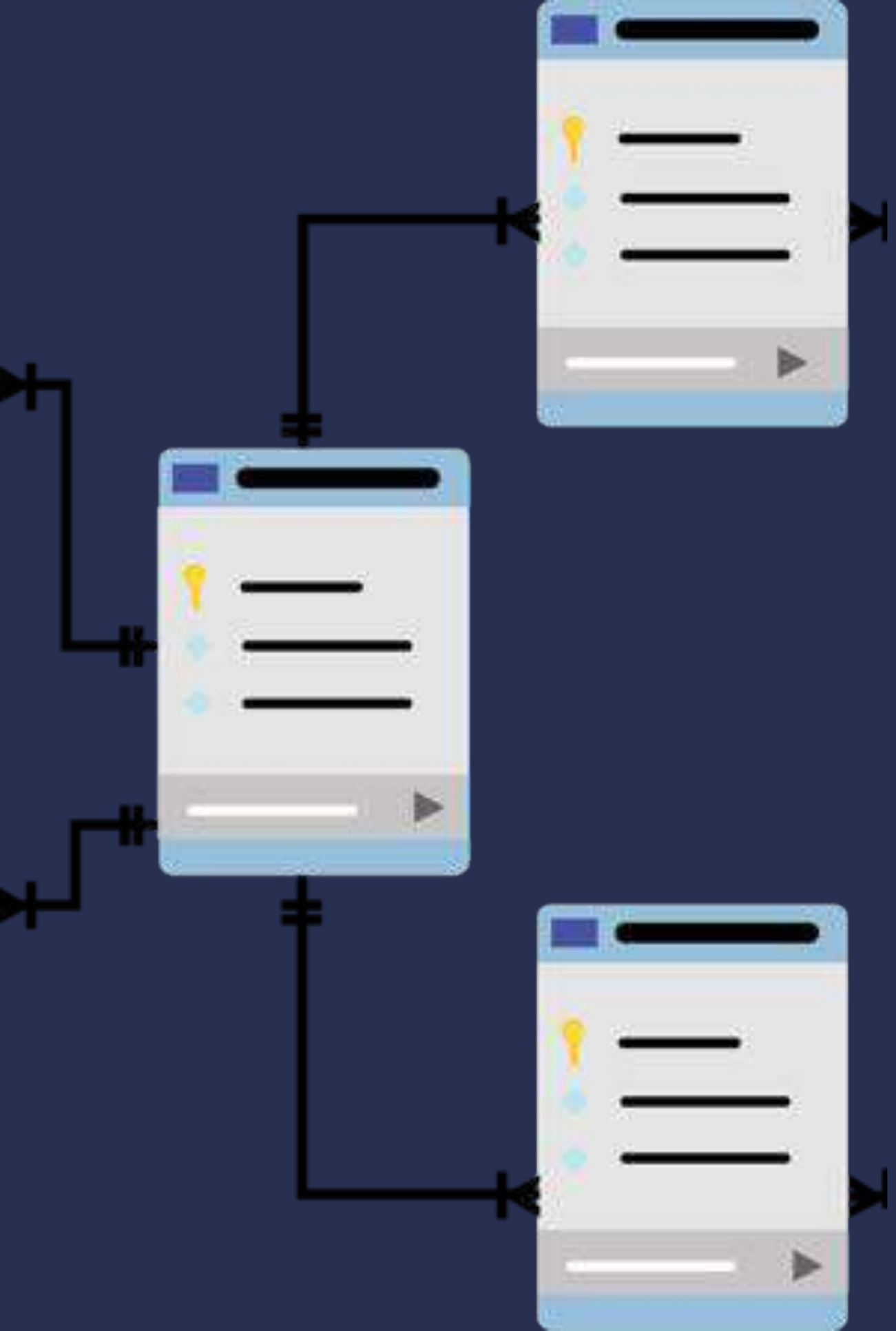
REAL WORLD EXAMPLES?

git

State: Files

Events: Commits

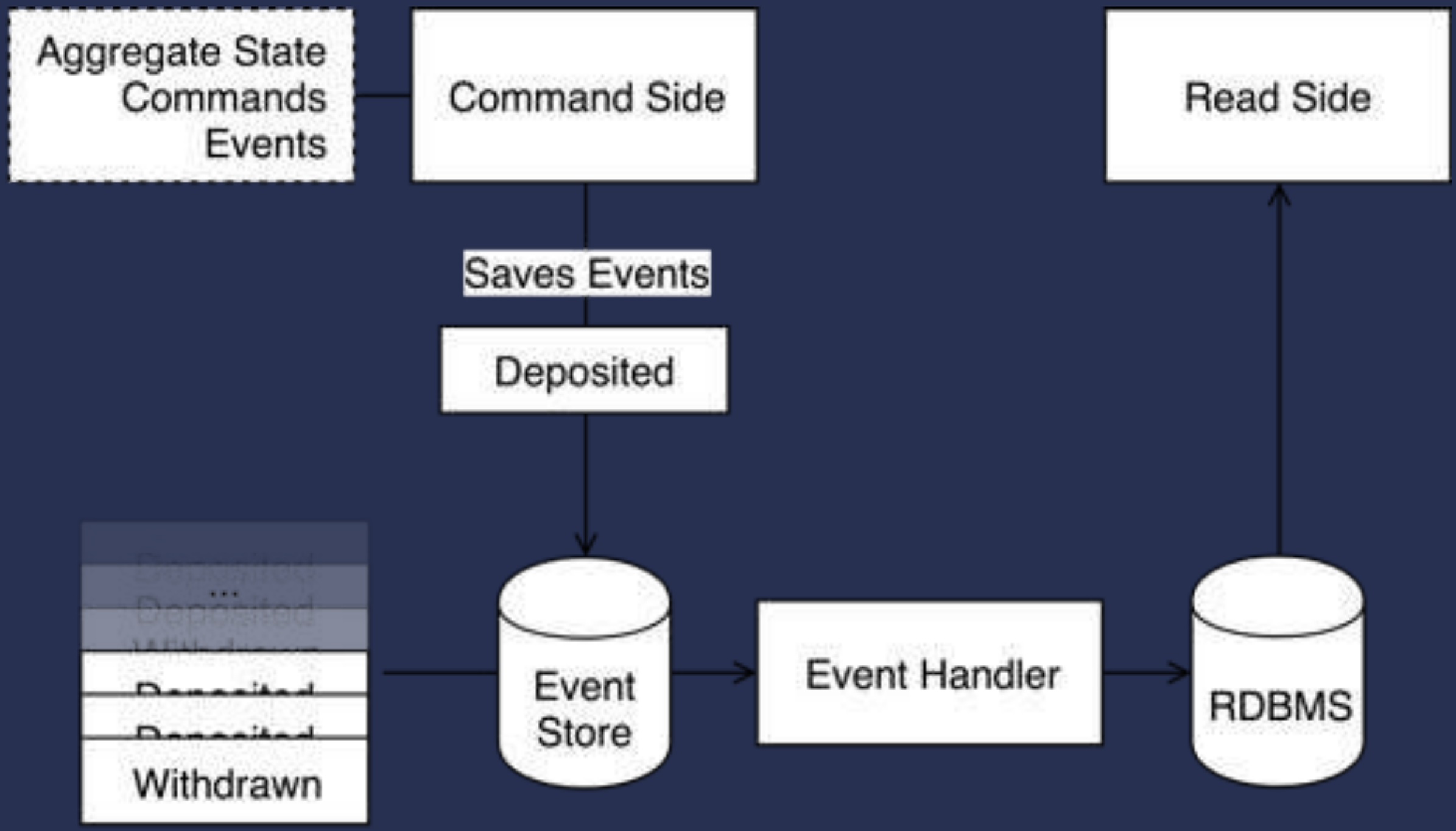
QUERYING



HOW ABOUT SAVING IN A RELATIONAL DATABASE?

CQRS

COMMAND-QUERY RESPONSIBILITY SEGREGATION


Aggregate State
Commands
Events
Command Side
Read Side
Saves Events
Deposited
...
Withdrawn
Event
Store
Event Handler
RDBMS

GITHUB.COM/COMMANDED/

COMMANDED

COMMANDED HELPS YOU BUILD THE

EVENT SOURCED COMMAND SIDE

OF YOUR APPLICATION

First you need to define your domain model consisting of

- Commands
- Events
- Aggregates

COMMANDS

- A request
- Imperative
- e.g. AddFunds(account_id: 1, amount: 10)
- May fail

EVENTS

- A fact
- In the past
- e.g. FundsAdded(account_id: 1, amount: 10)

AGGREGATES

- enforce business invariants
- has some state
- state changes with events
- may emit events in response to commands

```elixir
@spec execute(state, command)
      :: {:ok, [event]}
       | {:error, term()}

@spec apply(state, event) :: state
```

Then you...

- create a command router
- send commands to the router

QUERYING

- you cannot read your aggregates
- you can listen to events and build projections

LET'S CREATE OUR CRYPTO CURRENCY

WITH CQRS
AND EVENT SOURCING

UNWIND

WE WANT TO

```
defmodule Coins do
  # Commands
  def mine_coin(account_id, nonce)
  def send_coins(from_user, to_user, amount)

  # Query
  def richest()
end
```

RULES

MINING COINS

- Nonce should satisfy $f(\text{user}, \text{nonce}) = 0$
- Nonce should be bigger than last used nonce
- f will be a poor's man proof of work

SENDING COINS

- Source account should have enough coins

LET'S DO IT

MINING

DEFINE THE COMMAND

```elixir
defmodule MineCoin do
  defstruct [
    :account_id,
    :nonce
  ]
end
```

AND THE EVENT

```
defmodule CoinMined do
  defstruct [
    :account_id,
    :nonce
  ]
end
```

DEFINE YOUR AGGREGATE

```elixir
defmodule Account do
  defstruct [
    :last_nonce
  ]

  def apply(_, %CoinMined{} = evt) do
    %Account{last_nonce: evt.nonce}
  end
end
```

```elixir
defmodule Account do
  def execute(%{last_nonce: ln}, %MineCoin{nonce: n})
    when n < ln, do: {:error, :used_nonce}

  def execute(_, %MineCoin{} = cmd) do
    if Proof.proof(cmd.account_id, cmd.nonce) do
      %CoinMined{
        account_id: cmd.account_id,
        nonce: cmd.nonce
      }
    else
      {:error, :invalid_nonce}
    end
  end
end
```

```elixir
defmodule Account do
  def execute(%{last_nonce: ln}, %MineCoin{nonce: n})
    when n < ln, do: {:error, :used_nonce}

  def execute(_, %MineCoin{} = cmd) do
    if Proof.proof(cmd.account_id, cmd.nonce) do
      %CoinMined{
        account_id: cmd.account_id,
        nonce: cmd.nonce
      }
    else
      {:error, :invalid_nonce}
    end
  end
end
```

```elixir
defmodule Account do
  def execute(%{last_nonce: ln}, %MineCoin{nonce: n})
    when n < ln, do: {:error, :used_nonce}

  def execute(_, %MineCoin{} = cmd) do
    if Proof.proof(cmd.account_id, cmd.nonce) do
      %CoinMined{
        account_id: cmd.account_id,
        nonce: cmd.nonce
      }
    else
      {:error, :invalid_nonce}
    end
  end
end
```

```elixir
defmodule Account do
  def execute(%{last_nonce: ln}, %MineCoin{nonce: n})
    when n < ln, do: {:error, :used_nonce}

  def execute(_, %MineCoin{} = cmd) do
    if Proof.proof(cmd.account_id, cmd.nonce) do
      %CoinMined{
        account_id: cmd.account_id,
        nonce: cmd.nonce
      }
    else
      {:error, :invalid_nonce}
    end
  end
end
```

PROOF OF WORK (BONUS)

```elixir
defmodule Proof do
  def proof(aid, nonce) do
    String.starts_with?(
      hash(hash(aid) <> hash(to_string(nonce))),
      <<0>>
    )
  end

  defp hash(x), do: :crypto.hash(:sha256, x)
end
```

CREATE YOUR ROUTER

```
defmodule Router do
  use Commanded.Commands.Router

  dispatch(
    MineCoin,
    to: Account,
    identity: :account_id
  )
end
```

DISPATCH COMMANDS

```elixir
defmodule Coins do
  def mine_coin(account_id, nonce) do
    %MineCoin{
      account_id: account_id,
      nonce: nonce
    }
    |> Router.dispatch
  end
end
```

```
iex> Coins.mine_coin("me", 1)
{:error, :invalid_nonce}

iex> Coins.mine_coin("me", 190)
:ok
```

```
iex> Coins.mine_coin("me", 190)
{:error, :used_nonce}

iex> Coins.mine_coin("me", 443)
:ok
```

```
iex> EventStore.stream_forward("me") |> Enum.to_list
[
  %EventStore.RecordedEvent{
    data: %CoinMined{account_id: "me", nonce: 190},
    stream_uuid: "me",
    stream_version: 1
    #...
  },
  %EventStore.RecordedEvent{
    data: %CoinMined{account_id: "me", nonce: 443},
    stream_uuid: "me",
    stream_version: 2
    #...
  }
]
```

QUERYING

GITHUB.COM/COMMANDED/

COMMANDED-ECTO-PROJECTIONS

SAY WE HAVE A SCHEMA...

```elixir
defmodule Schemas.Account do
  use Ecto.Schema

  @primary_key false

  schema "accounts" do
    field(:account_id, :string)
    field(:balance, :integer)
  end
end
```

CREATE A PROJECTOR

```
defmodule AccountProjector do
  use Commanded.Projections.Ecto,
    name: "AccountProjector"

  project %CoinMined{} = evt do
    inc_balance(multi, evt.account_id, 1)
  end
end
```

CREATE A PROJECTOR

```
defp inc_balance(multi, account_id, amount) do
  Ecto.Multi.insert(
    multi,
    :increase_balance,
    %Schemas.Account{
      account_id: account_id,
      balance: amount
    },
    conflict_target: :account_id,
    on_conflict: [inc: [balance: amount]]
  )
end
```

DEFINE THE API

```
defmodule Coins do
  def richest do
    import Ecto.Query

    Schemas.Account
    |> order_by(desc: :balance)
    |> limit(1)
    |> Repo.one
  end
end
```

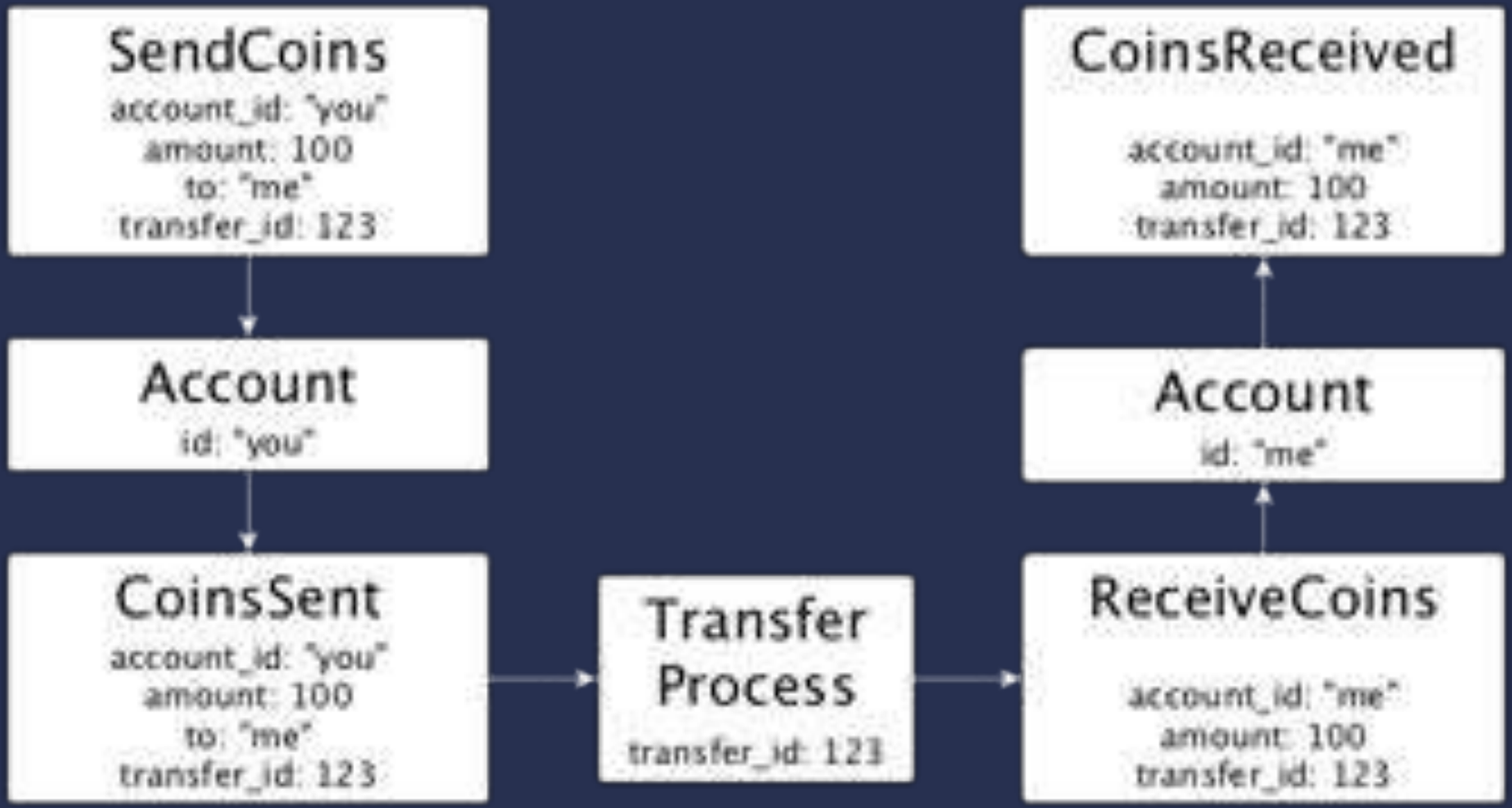
```
iex> Coins.richest |>
...> Map.take([:account_id, :balance])
%{account_id: "me", balance: 1}

iex> [488, 1442, 1597] |>
...> Enum.map(&(Coin.mine_coin("you" &1)))
[:ok, :ok, :ok]

iex> Coins.richest |>
...> Map.take([:account_id, :balance])
%{account_id: "you", balance: 3}
```

HOW TO MAKE TRANSFERS

THE PROBLEM WITH SENDING COINS IS THAT WE TOUCH TWO AGGREGATES

PROCESS MANAGER

A PROCESS MANAGER RECEIVE EVENTS & SEND COMMANDS


SendCoins
account_id: "you"
amount: 100
to: "me"
transfer_id: 123
Account
id: "you"
CoinsSent
account_id: "you"
amount: 100
to: "me"
transfer_id: 123
Transfer Process
transfer_id: 123
CoinsReceived
account_id: "me"
amount: 100
transfer_id: 123
Account
id: "me"
ReceiveCoins
account_id: "me"
amount: 100
transfer_id: 123

SO WE ADD SOME COMMANDS...

```elixir
defmodule SendCoins do
  defstruct [
    :account_id,
    :to,
    :amount,
    :transfer_id
  ]
end
```

```elixir
defmodule ReceiveCoins do
  defstruct [
    :account_id,
    :amount,
    :transfer_id
  ]
end
```

AND SOME EVENTS...

```elixir
defmodule CoinsSent do
  defstruct [
    #..
  ]
end

defmodule CoinsReceived do
  defstruct [
    #...
  ]
end
```

REDEFINE OUR AGGREGATE STATE

```elixir
defmodule Account do
  defstruct [
    :last_nonce,
    :balance
  ]

  # ...
end
```

REDEFINE OUR AGGREGATE STATE

```elixir
def apply(state, %CoinMined{} = evt) do
  %Account{ state | last_nonce: evt.nonce }
  |> inc_balance(1)
end
```

```elixir
def apply(s, %CoinsSent{} = evt),
  do: inc_balance(state, -evt.amount)

def apply(s, %CoinsReceived{} = evt),
  do: inc_balance(state, evt.amount)
```

ENFORCE INVARIANTS

```
def execute(
  %{balance: b}, %SendCoins{amount: a}
) when a > b, do: {:error, :not_enough_coins}
```

```
def execute(_, %SendCoins{} = cmd) do
  %CoinsSent{
    account_id: cmd.account_id,
    to: cmd.to,
    amount: cmd.amount,
    transfer_id: cmd.transfer_id
  }
end
```

EMIT EVENT

```
def execute(
  %{balance: b}, %SendCoins{amount: a}
) when a > b, do: {:error, :not_enough_coins}
```

```
def execute(_, %SendCoins{} = cmd) do
  %CoinsSent{
    account_id: cmd.account_id,
    to: cmd.to,
    amount: cmd.amount,
    transfer_id: cmd.transfer_id
  }
end
```

DEFINE OUR API

```
defmodule Coins do
  def send_coins(from, to, amount) do
    %SendCoins{
      account_id: from,
      to: to,
      amount: amount,
      transfer_id: UUID.uuid4
    }
    |> Router.dispatch
  end
end
```

NOW DEFINE THE PROCESS MANAGER

```elixir
defmodule Coins.SendCoinsProcess do
  use Commanded.ProcessManagers.ProcessManager,
    name: "SendCoinsProcess",
    router: Router

  defstruct []

  def apply(state, _event), do: state
end
```

```elixir
def interested?(
  %E.CoinsSent{transfer_id: tid}
) do
  {:start, tid}
end
```

```elixir
def interested?(
  %E.CoinsReceived{transfer_id: tid}
) do
  {:stop, tid}
end
```

```elixir
def handle(_, %E.CoinsSent{} = evt) do
  %C.ReceiveCoins{
    transfer_id: evt.transfer_id,
    account_id: evt.to,
    amount: evt.amount
  }
end
```

THEN YOU...

- update your **projector** to react to these events
- update your **router** to accept these commands
- add the process manager to your **supervisor tree**

```
iex> Coins.send_coins("you", "me", 99999)
{:error, :not_enough_coins}

iex> Coins.send_coins("you", "me", 3)
:ok
```

```
iex> Coins.richest |>
...> Map.take([:account_id, :balance])
%{account_id: "me", balance: 5}
```

```
iex> EventStore.stream_forward("me") |>  ...
[
  %CoinMined{nonce: 190},
  %CoinMined{nonce: 443},
  %CoinsReceived{from: "you"}
]
```

```
iex> EventStore.stream_forward("me") |>  ...
[
  %CoinMined{nonce: 488},
  %CoinMined{nonce: 1442},
  %CoinMined{nonce: 1597},
  %CoinsSent{to: "me"}
]
```

NK YOU

QUESTIONS?