Event sourcing for everyone

Jenna Blumenthal

Agenda

- What & why
- Key components of an event sourced system
- Example 

Account

id	email	is_active	plan_tier	created_at	updated_at

Account#register

```
account = Account.find_or_initialize_by(params[:email])
Account.is_active = true
account.save!
```

id	email	is_active	plan_tier
1	ruby@mail.com	true	free

Account#change_plan

```
account.update(plan_tier: “paid”)
account.save!
```

id	email	is_active	plan_tier
1	ruby@mail.com	true	paid

Account#disable

```
account.update(is_active: false)
account.save!
```

id	email	is_active	plan_tier
1	ruby@mail.com	false	paid

Account#register

```
account = Account.find_or_initialize_by(params[:email])
Account.is_active = true
account.save!
```

id	email	is_active	plan_tier
1	ruby@mail.com	true	free

Account#disable

```
account.update(is_active: false)
account.save!
```

id	email	is_active	plan_tier
1	ruby@mail.com	false	paid

Data loss

Logging

```
def change_plan(new_plan_tier:)
  Logger.info("Plan changed from #{plan_tier} to #{new_plan_tier}")
  self.plan_tier = new_plan_tier
end
```

All the columns

id	is_active	disabled_at	disabled_reason
1	false	2018-05-28 15:06:04	payment_required

Account Audit Table

```ruby
class Account < ApplicationRecord
  after_commit :create_account_audit_log
  ...

  def create_account_audit_log
    AccountAudit.create!(
      email: email,
      is_active: is_active,
      plan_tier: plan_tier,
      account_id: id
    )
  end
end
```

Account

id	email	is_active	plan_tier
1	ruby@mail.com	false	paid

AccountAudit

id	account_id	email	is_active	plan_tier
1	1	ruby@mail.com	true	free
2	1	ruby@mail.com	true	paid
3	1	ruby@mail.com	false	paid
4	1	ruby@mail.com	true	free
5	1	ruby@mail.com	false	free

Why did the state change?

“Capture all changes to an application state as a sequence of events.”

— Martin Fowler

Event

as a first class citizen

- Contains all info needed to change from one state to the next
- Immutable, append-only
- Events are persisted, current state is disposable
- Basis for universal language

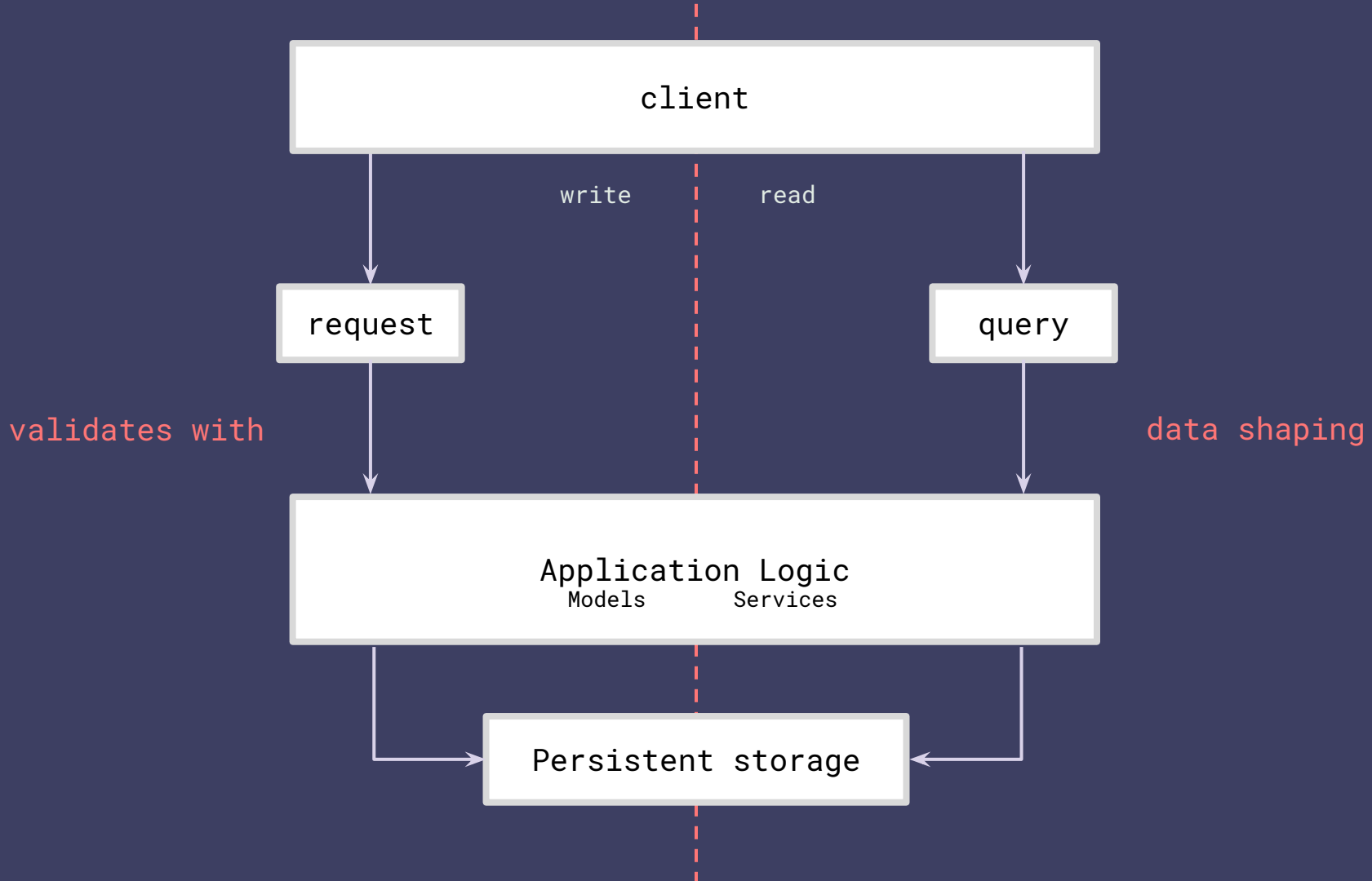
(CQRS: an interlude)

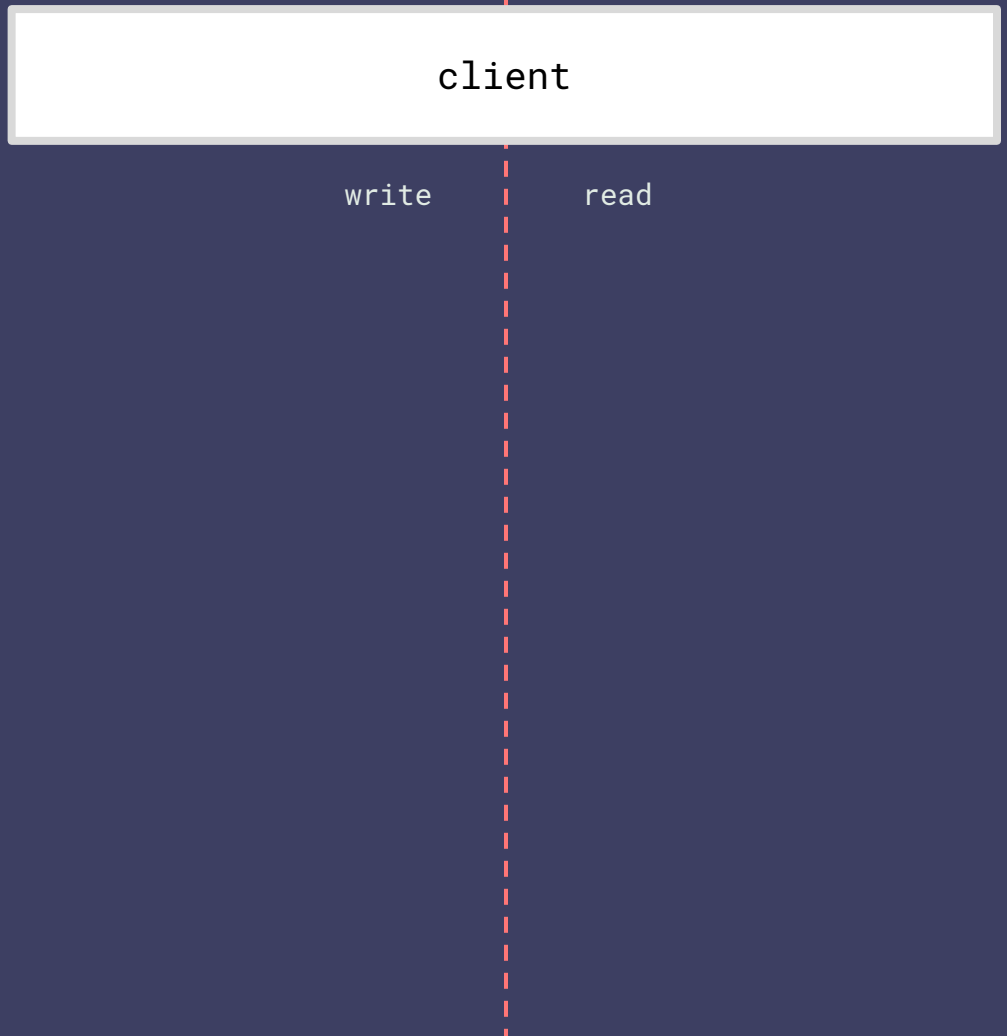
write

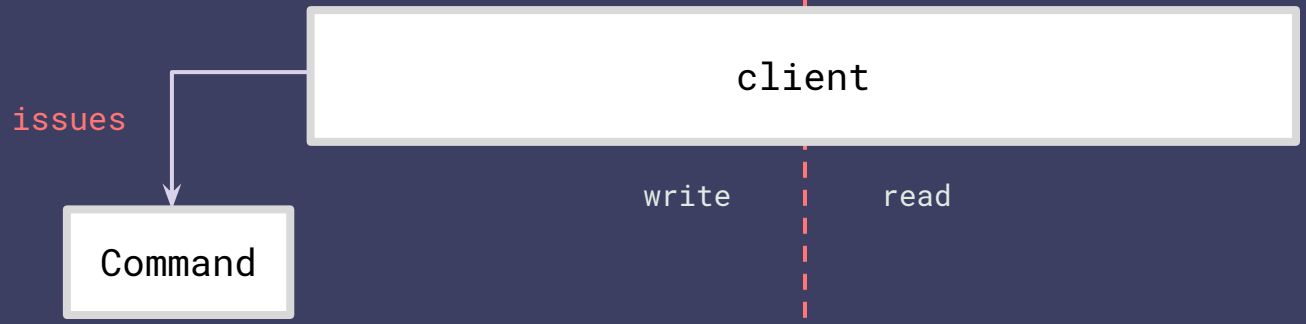
- data integrity
- optimize inserts/update
- write security

read

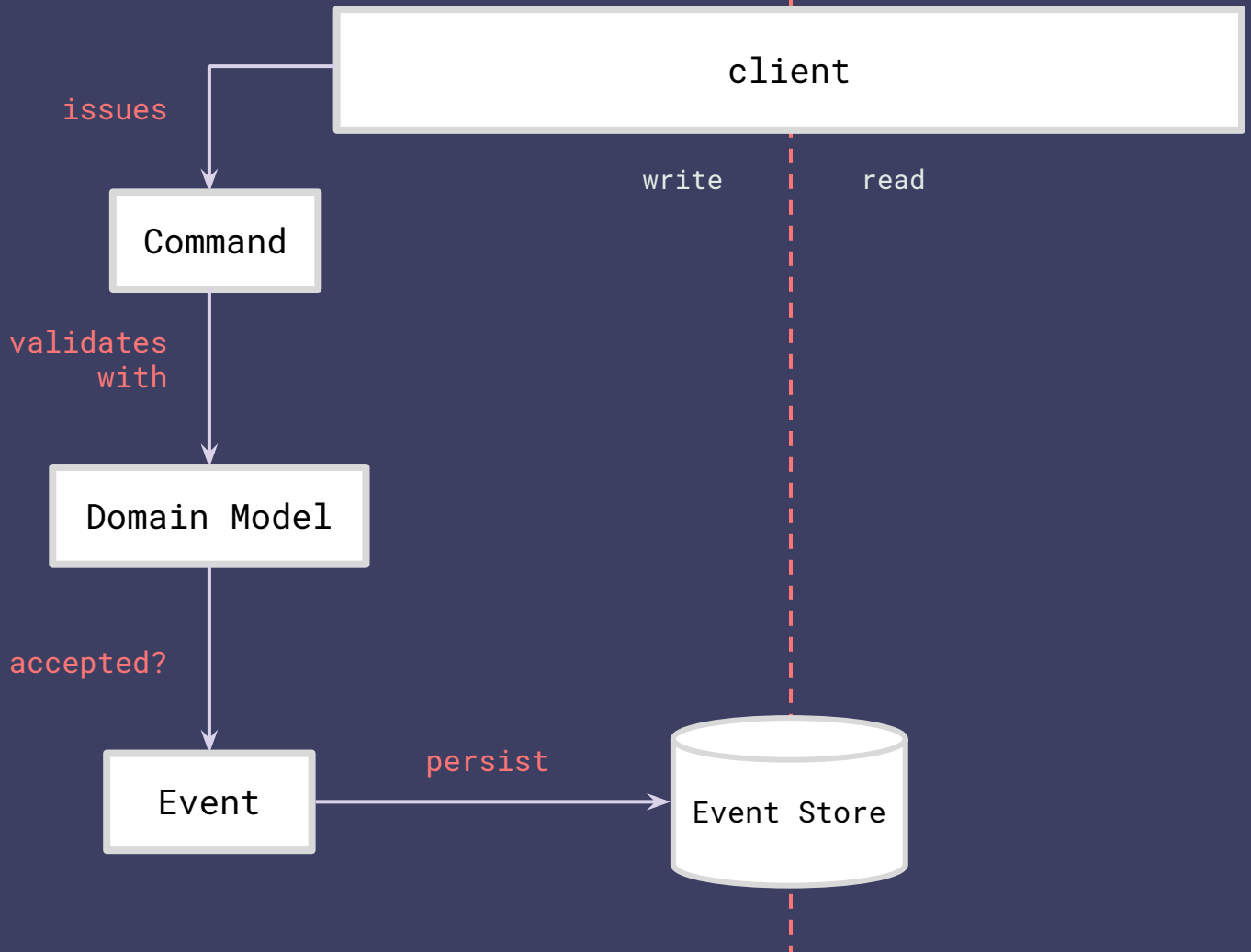
- efficient queries
- denormalized data
- read security

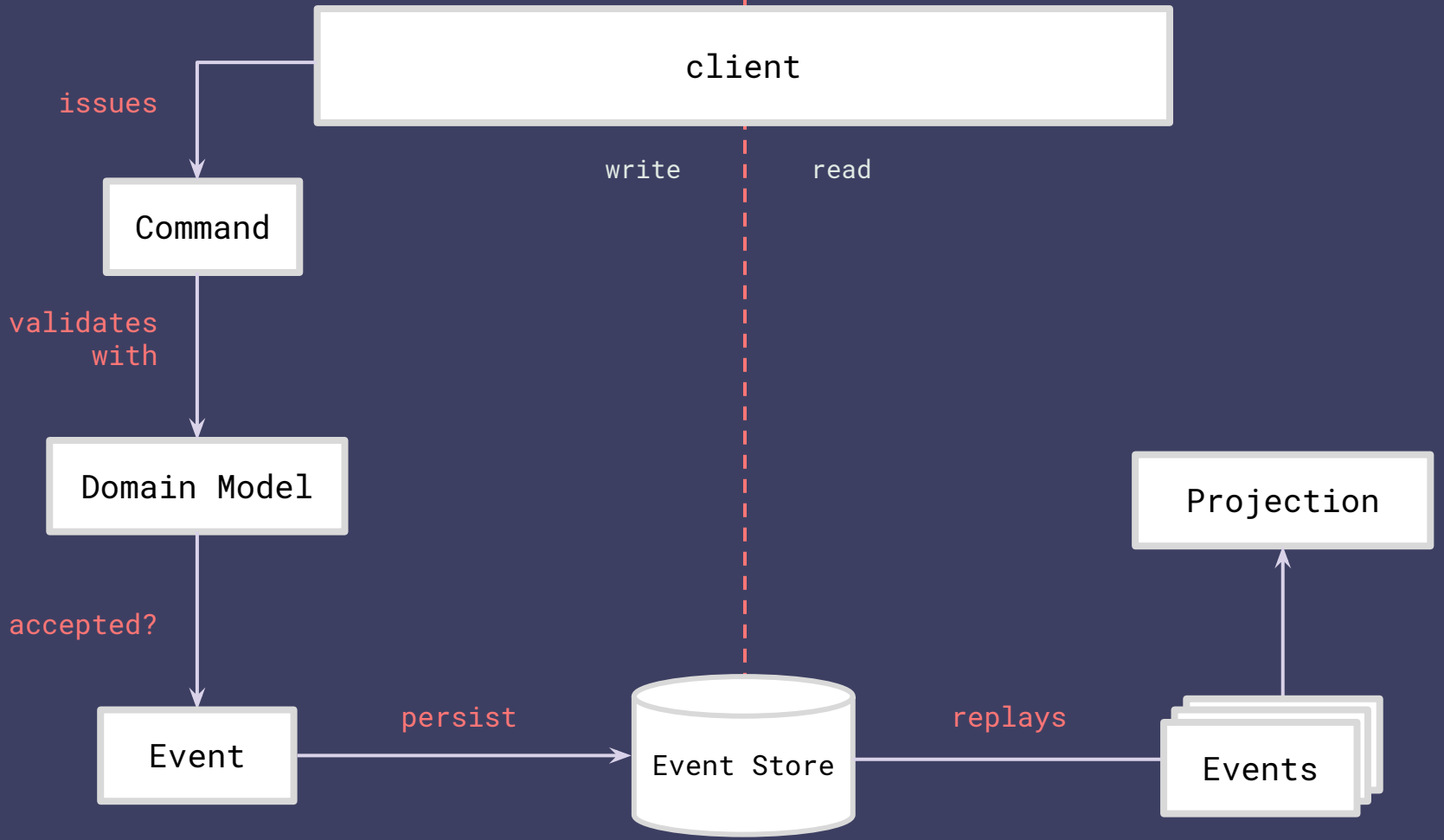

client
write
read
request
query
validates with
data shaping
Application Logic
Models
Services
Persistent storage

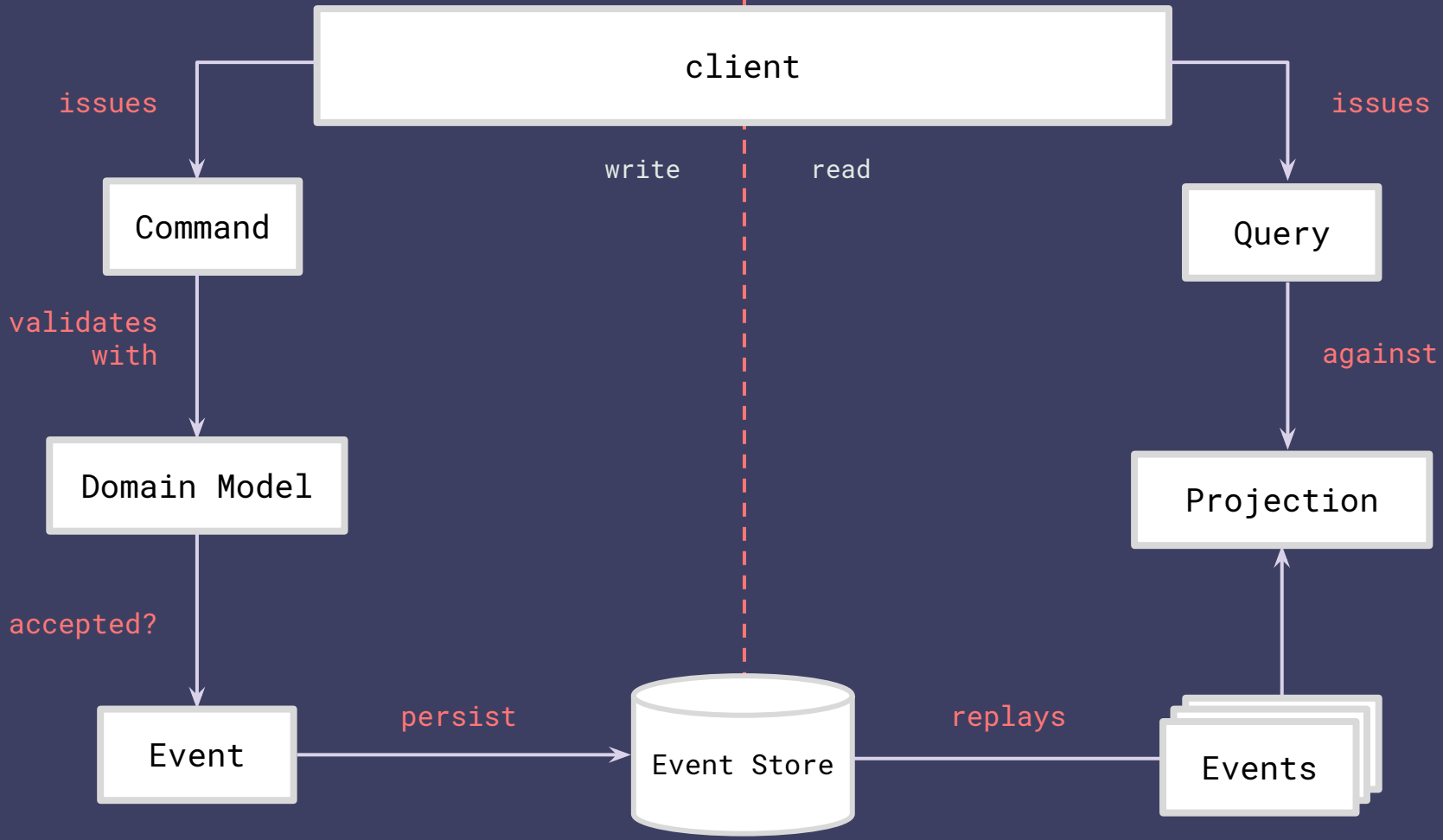

client
write
read


client
issues
Command
write
read


client
issues
write
read
Command
validates
with
Domain Model


client
issues
write
read
Command
validates
with
Domain Model
accepted?
Event
persist
Event Store


client
issues
write
read
Command
validates
with
Domain Model
Projection
accepted?
Event
persist
Event Store
replays
Events


client
issues
issues
write
read
Command
Query
validates with
against
Domain Model
Projection
accepted?
persist
replays
Event
Event Store
Events

Components

Command:

- communicates intent
- can be rejected

Components

Command:

- communicates intent
- can be rejected

Domain model:

- the ‘nouns’ of your system
- holds business logic

Components

Command:

- communicates intent
- can be rejected

Domain model:

- the ‘nouns’ of your system
- holds business logic

Event:

- all the information needed to change from one state to another
- append-only

Components

Command:
- communicates intent
- can be rejected

Domain model:
- the ‘nouns’ of your system
- holds business logic

Event:
- all the information needed to change from one state to another
- append-only

Projection:
- read-only
- current state

Show me the code

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    self.plan_tier = new_plan_tier
  end

  def disable(reason:)
    self.is_active = false

    if reason == DISABLED_REASONS[:manually_disabled]
      SendSorryToSeeYouGoEmailJob.new(self.id)
    end
  end
end
```

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    self.plan_tier = new_plan_tier
  end

  def disable(reason:)
    self.is_active = false

    if reason == DISABLED_REASONS[:manually_disabled]
      SendSorryToSeeYouGoEmailJob.new(self.id)
    end
  end
end
```

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    self.plan_tier = new_plan_tier
  end

  def disable(reason:)
    self.is_active = false

    if reason == DISABLED_REASONS[:manually_disabled]
      SendSorryToSeeYouGoEmailJob.new(self.id)
    end
  end
end
```

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    self.plan_tier = new_plan_tier
  end

  def disable(reason:)
    self.is_active = false

    if reason == DISABLED_REASONS[:manually_disabled]
      SendSorryToSeeYouGoEmailJob.new(self.id)
    end
  end
end
```

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

end
```

1. command

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

end
```

2. validations

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

end
```

3. some stuff

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    self.email = params[:email]
    self.is_active = true
  end

end
```

```ruby
# account.rb
class Account < ApplicationRecord

  def register(params = {})
    raise ArgumentError, 'Email is required' unless params[:email].present?
    account_registered(params)
  end

  ...

  private

  def account_registered(params = {})
    self.email = params[:email]
    self.is_active = true
  end
end
```

```ruby
# account.rb
class Account < ApplicationRecord

  def disable(reason:)
    account_disabled(reason)
  end

  ...

  private

  def account_disabled(reason)
    self.is_active = false

    if reason == DISABLED_REASONS[:manually_disabled]
      SendSorryToSeeYouGoEmailJob.new(self.id)
    end
  end
end
```

```ruby
class Account < ApplicationRecord

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    plan_changed(new_plan_tier)
  end

  ...

  private

  def plan_changed(new_plan_tier)
    self.plan_tier = new_plan_tier
  end
end
```

```ruby
class Account < ApplicationRecord

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    plan_changed(new_plan_tier)
  end

  ...

  private

  def plan_changed(new_plan_tier)
    self.plan_tier = new_plan_tier
  end
end
```

Event

“all the information needed to change from one state to another ”

```ruby
class Account < ApplicationRecord

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    plan_changed()
  end

  ...

  private

  def plan_changed()
    self.plan_tier = new_plan_tier
  end
end
```

```ruby
class Account < ApplicationRecord

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && new_plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    plan_changed(
      Events::Account::PlanChanged.new(
        object_reference_id: self.uuid,
        object_type: self.class.to_s,
        payload: {
          old_plan: self.plan_tier,
          new_plan: new_plan_tier
        }
      )
    )
  end

  private

  def plan_changed(event)
    self.plan_tier = event.payload[:new_plan]
  end
end
```

```ruby
class Account < ApplicationRecord

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && new_plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    plan_changed(
      Events::Account::PlanChanged.new(
        object_reference_id: self.uuid,
        object_type: self.class.to_s,
        payload: {
          old_plan: self.plan_tier,
          new_plan: new_plan_tier
        }
      )
    )
  end

  private

  def plan_changed(event)
    self.plan_tier = event.payload[:new_plan]
  end
end
```

```ruby
class Account < ApplicationRecord
  include Domain::BaseObject

  def change_plan(new_plan_tier:)
    if self.plan_tier == 'free' && new_plan_tier == 'advanced'
      raise ArgumentError, 'illegal plan transition'
    end

    apply(
      Events::Account::PlanChanged.new(
        object_reference_id: self.uuid,
        object_type: self.class.to_s,
        payload: {
          old_plan: self.plan_tier,
          new_plan: new_plan_tier
        }
      )
    )
  end

  private

  def plan_changed(event)
    self.plan_tier = event.payload[:new_plan]
  end
end
```

```ruby
# domain/base_object.rb
def apply(event, is_new_event: true)
  handle(event)
  publish(event, is_new_event)
end

private

def handle(event)
  handler = event.handler
  if self.respond_to? handler
    public_send(handler, event)
  end
end

def publish(event, is_new_event)
  EventStore.save(event) if is_new_event
end
```

```ruby
# domain/base_object.rb
def apply(event, is_new_event: true)
  handle(event)
  publish(event, is_new_event)
end

private

def handle(event)
  handler = event.handler
  if self.respond_to? handler
    public_send(handler, event)
  end
end

def publish(event, is_new_event)
  EventStore.save(event) if is_new_event
end
```

e.g. :account_registered
:plan_changed
:account_disabled

```ruby
# domain/base_object.rb
def apply(event, is_new_event: true)
  handle(event)
  publish(event, is_new_event)
end

private

def handle(event)
  handler = event.handler
  if self.respond_to? handler
    public_send(handler, event)
  end
end

def publish(event, is_new_event)
  EventStore.save(event) if is_new_event
end
```

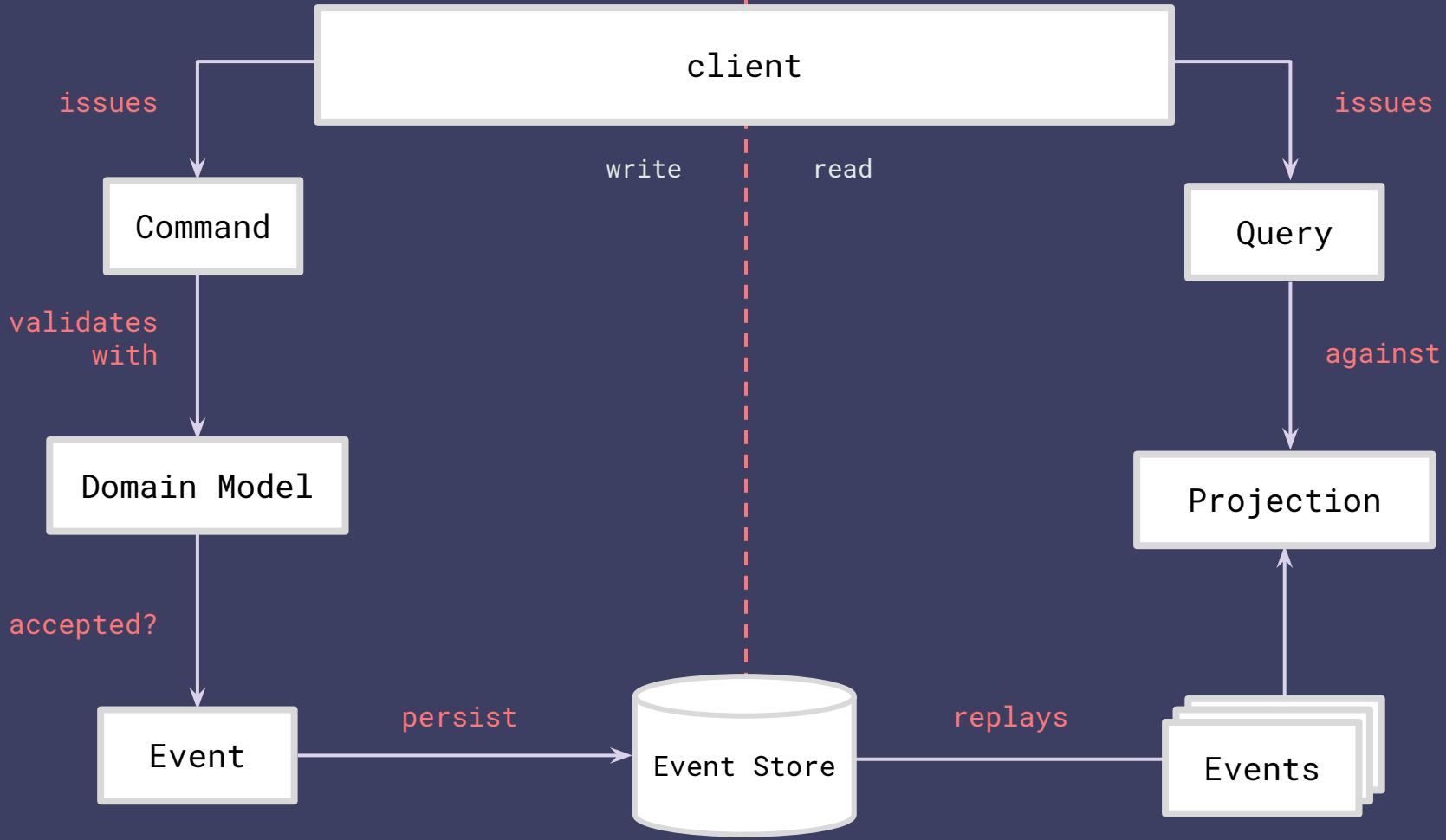
Not production ready!

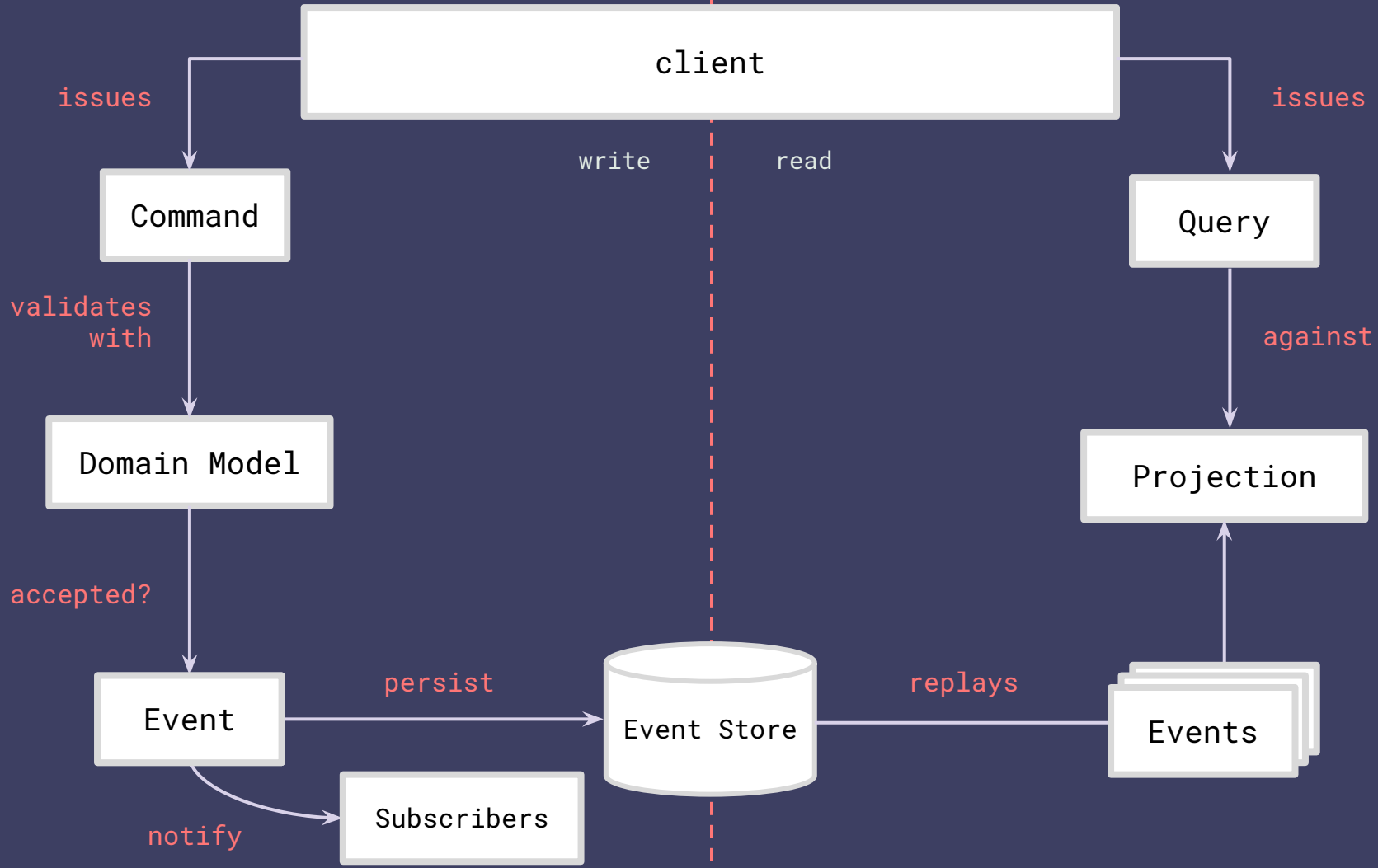
```ruby
# events/event_store.rb
class EventStore
  @@event_streams={}

  def self.save(event)
    current_stream = @@event_streams[event.object_reference_id] || []
    current_stream << event
    @@event_streams[event.object_reference_id] = current_stream
  end

  def self.load(klass, uuid = nil)
    object = klass.new
    object.uuid = uuid
    events = @@event_streams[uuid]
    object.rebuild(events) if events.present?
    object
  end
end
```

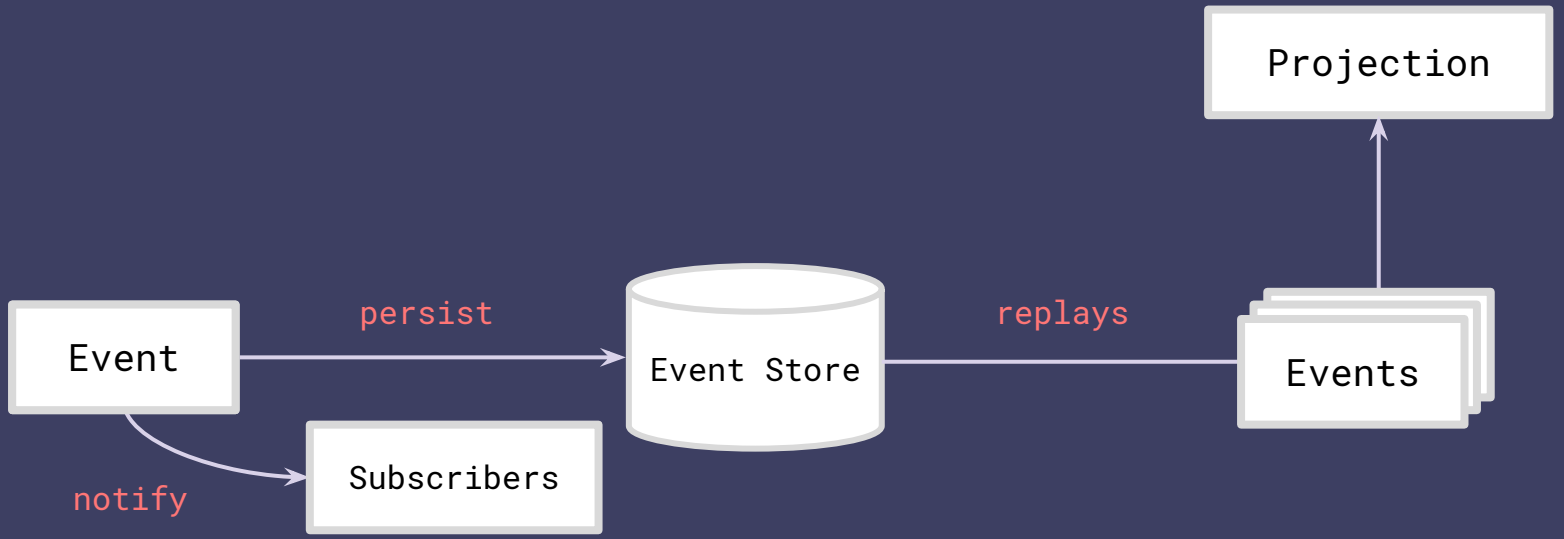
```ruby
# events/event_store.rb
class EventStore
  @@event_streams={}

  def self.save(event)
    current_stream = @@event_streams[event.object_reference_id] || []
    current_stream << event
    @@event_streams[event.object_reference_id] = current_stream
  end

  def self.load(klass, uuid = nil)
    object = klass.new
    object.uuid = uuid
    events = @@event_streams[uuid]
    object.rebuild(events) if events.present?
    object
  end
end
```

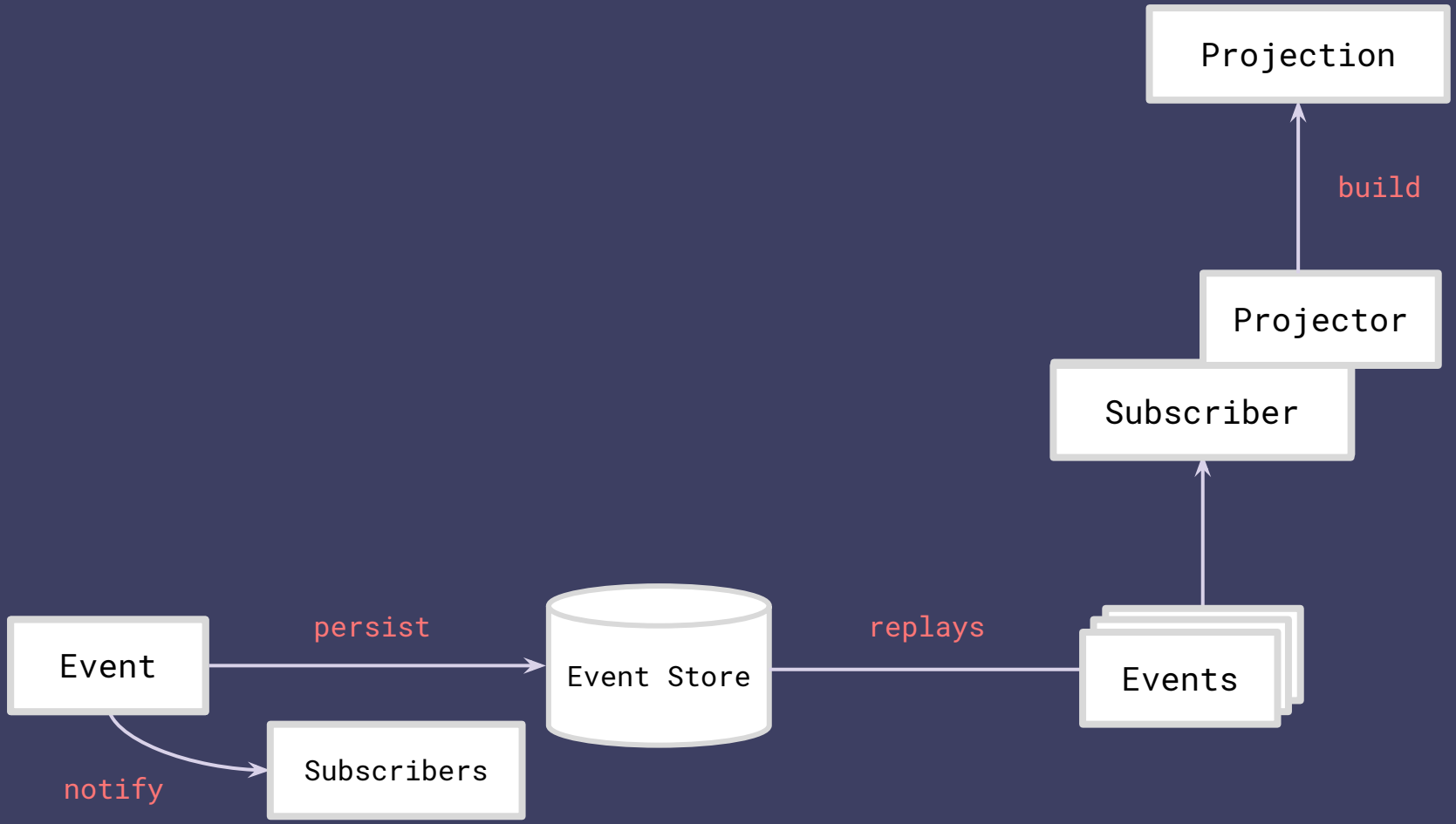
```ruby
# events/event_store.rb
class EventStore
  @@event_streams={}

  def self.save(event)
    current_stream = @@event_streams[event.object_reference_id] || []
    current_stream << event
    @@event_streams[event.object_reference_id] = current_stream
  end

  def self.load(klass, uuid = nil)
    object = klass.new
    object.uuid = uuid
    events = @@event_streams[uuid]
    object.rebuild(events) if events.present?
    object
  end
end
```

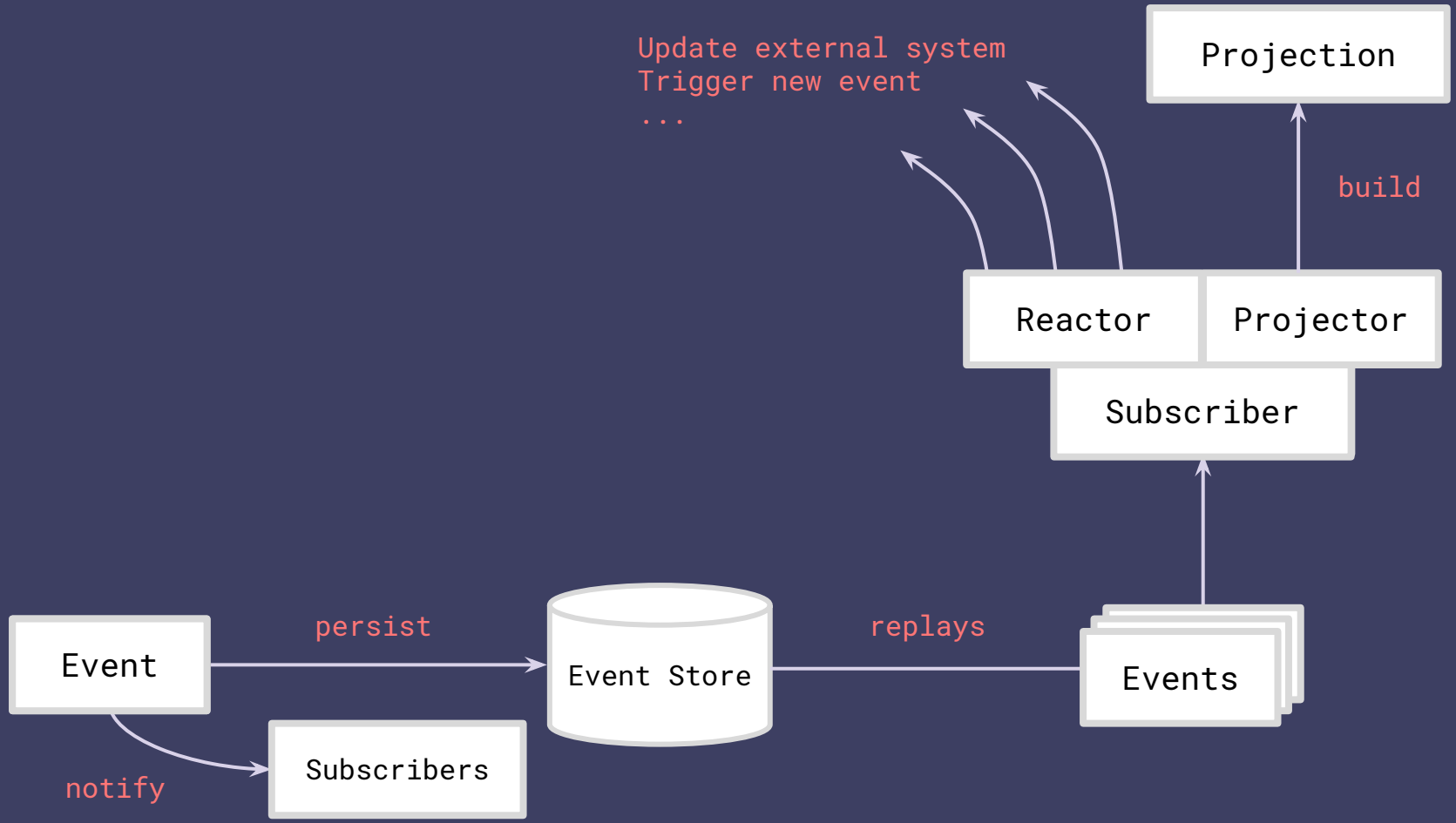
```ruby
# domain/base_object.rb
def apply(event, is_new_event: true)
  handle(event)
  publish(event, is_new_event)
end

def rebuild(events)
  events.each do |event|
    apply(event, is_new_event: false)
  end
end

private

def handle(event)
  handler = event.handler
  if self.respond_to? handler
    public_send(handler, event)
  end
end

def publish(event, is_new_event)
  EventStore.save(event) if is_new_event
end
```


client
issues
issues
write
read
Command
Query
validates with
against
Domain Model
Projection
accepted?
Event
persist
Event Store
replays
Events


client
issues
issues
write
read
Command
Query
validates with
against
Domain Model
Projection
accepted?
persist
replays
Event
Event Store
Events
Subscribers
notify


Projection
Event
persist
Event Store
replays
Events
Subscribers
notify


Projection
build
Projector
Subscriber
Event
persist
Event Store
replays
Events
Subscribers
notify


Update external system
Trigger new event
...
Projection
build
Reactor
Projector
Subscriber
Event
persist
Event Store
replays
Events
Subscribers
notify

```ruby
# domain/base_object.rb
def apply(event, is_new_event: true)
  handle(event)
  publish(event, is_new_event)
end

...
private

def publish(event, is_new_event)
  EventStore.save(event) if is_new_event
  notify_subscribers(event, is_new_event)
end

def notify_subscribers(event, is_new_event)
  SubscriberManager.notify_subscribers(event, is_new_event)
end
```

```ruby
# subscribers/subscriber_manager.rb
class SubscriberManager
  ALL_SUBSCRIBERS = [
    Reactors::DisabledAccountEmailSender,
    Projectors::AllAccounts,
    Projectors::ActiveAccounts,
  ]

  def self.notify_subscribers(event, is_new_event)
    ALL_SUBSCRIBERS.each do |subscriber|
      subscriber.new(event, is_new_event).process
    end
  end
end
```

```ruby
# subscribers/subscriber_manager.rb
class SubscriberManager
  ALL_SUBSCRIBERS = [
    Reactors::DisabledAccountEmailSender,
    Projectors::AllAccounts,
    Projectors::ActiveAccounts,
  ]

  def self.notify_subscribers(event, is_new_event)
    ALL_SUBSCRIBERS.each do |subscriber|
      subscriber.new(event, is_new_event).process
    end
  end
end
```

```ruby
# subscribers/base_subscriber.rb
module Subscribers
  class BaseSubscriber
    attr_reader :event, :is_new_event
    def initialize(event, is_new_event)
      @event = event
      @is_new_event = is_new_event
    end

    def valid?
      raise NotImplementedError, 'derived classes must implement #valid?'
    end
  end
end
```

```ruby
# reactors/disabled_account_email_sender.rb
module Reactors
  class DisabledAccountEmailSender < Subscribers::BaseSubscriber
    def process
      return unless valid?
      SendSorryToSeeYouGoEmailJob.perform_later(event.object_reference_id)
    end

    def valid?
      event.is_a?(Events::Account::Disabled) &&
      is_new_event &&
      event.payload[:reason] == Account::DISABLED_REASONS[:manually_disabled]
    end
  end
end
```

All active users

```ruby
# projectors/active_accounts.rb
module Projectors
  class ActiveAccounts
    def process
      case event
      when Events::Account::Registered
        account = Account.find_or_create_by!(uuid: event.object_reference_id)
        account.update(is_active: true, email: event.payload[:email])

      when Events::Account::PlanChanged
        account = Account.find_by(uuid: event.object_reference_id)
        account.update(plan_tier: event.payload[:new_plan]) if account

      when Events::Account::Disabled
        account = Account.find_by(uuid: event.object_reference_id)
        account.destroy
      end
    end
  end
end
```


I IMMEDIATELY REGRET THIS DECISION
memegenerator.net

write:

```
class Domain::AccountObject
```

read:

```
class Account < ApplicationRecord
```

write:

```
class Domain::AccountObject
```

read:

```
class Account < ApplicationRecord
class ActiveAccount < ApplicationRecord
class PlanTierCount < ApplicationRecord
class DisabledAccountReasonByMonth < ApplicationRecord
```


client
issues
Command
validates
with
Domain Model

```ruby
module Domain
  class AccountObject
    def register(params = {})
      apply(
       ...
      )
    end

    def change_plan(new_plan_tier:)
      apply(
        ...
      )
    end

    def disable(reason:)
      apply(
        ...
      )
    end
  end
end
```

```ruby
module Commands
  module Account
    module Register
      class Command < BaseCommand
        attr_reader :args
        def initialize(args = {})
          @args = args
          validate!
        end

        def validate!
          raise ArgumentError, 'email is missing' if args[:email].nil?
        end
      end
    end
  end
end
```

```ruby
class CommandHandler
  def handle(command)
    account = EventStore.load(Domain::AccountObject)
    account.apply(
      Events::Account::Registered.new(
        object_reference_id: account.uuid,
        payload: {
          is_active: command.args[:is_active],
          email: command.args[:email]
        }
      )
    )
  end
end
```

What's easy?

- Modelling business logic
- Communicating with interdisciplinary team
- Scaling reads/writes without coupling
- Traceability & audits

What's hard?

- Naming ALL THE THINGS
- More work for implementing simple logic
- Schema changes
- Eventual consistency
- Side effects on replaying events
- Creating lots of data

Monoliths

A simple arrangement of complex things

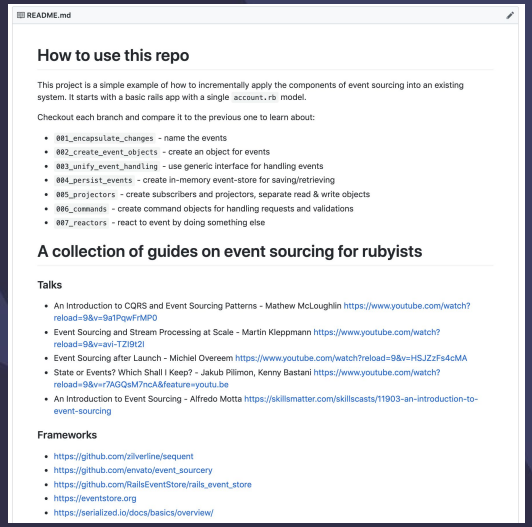
Event sourcing

A complex arrangement of simple things

— Sebastian von Conrad

On the shoulders of giants 🙏

github.com/jennaleeb/
event_sourcing_for_everyone



README.md

How to use this repo

This project is a simple example of how to incrementally apply the components of event sourcing into an existing system. It starts with a basic rails app with a single `account.rb` model.

Checkout each branch and compare it to the previous one to learn about:

- `001_encapsulate_changes` - name the events
- `002_create_event_objects` - create an object for events
- `003_unify_event_handling` - use generic interface for handling events
- `004_persist_events` - create in-memory event-store for saving/retrieving
- `005_projectors` - create subscribers and projectors, separate read & write objects
- `006_commands` - create command objects for handling requests and validations
- `007_reactors` - react to event by doing something else

A collection of guides on event sourcing for rubyists

Talks

- An Introduction to CQRS and Event Sourcing Patterns - Mathew McLoughlin https://www.youtube.com/watch?reload=9&v=9a1PqwFrMP0
- Event Sourcing and Stream Processing at Scale - Martin Kleppmann https://www.youtube.com/watch?reload=9&v=avi-TZI9t2I
- Event Sourcing after Launch - Michiel Overeem https://www.youtube.com/watch?reload=9&v=HSJZzFs4cMA
- State or Events? Which Shall I Keep? - Jakub Pilimon, Kenny Bastani https://www.youtube.com/watch?reload=9&v=r7AGQsM7ncA&feature=youtu.be
- An Introduction to Event Sourcing - Alfredo Motta https://skillsmatter.com/skillscasts/11903-an-introduction-to-event-sourcing

Frameworks

- https://github.com/zilverline/sequent
- https://github.com/envato/event_sourcery
- https://github.com/RailsEventStore/rails_event_store
- https://eventstore.org
- https://serialized.io/docs/basics/overview/