EVENT SOURCING

by: Silas Baronda

REVIEWING THE CLASSIC SYSTEM

CHARACTERISTICS

1. Read and write both go through the same layers
2. We use the same model for read and write access
3. We change data directly

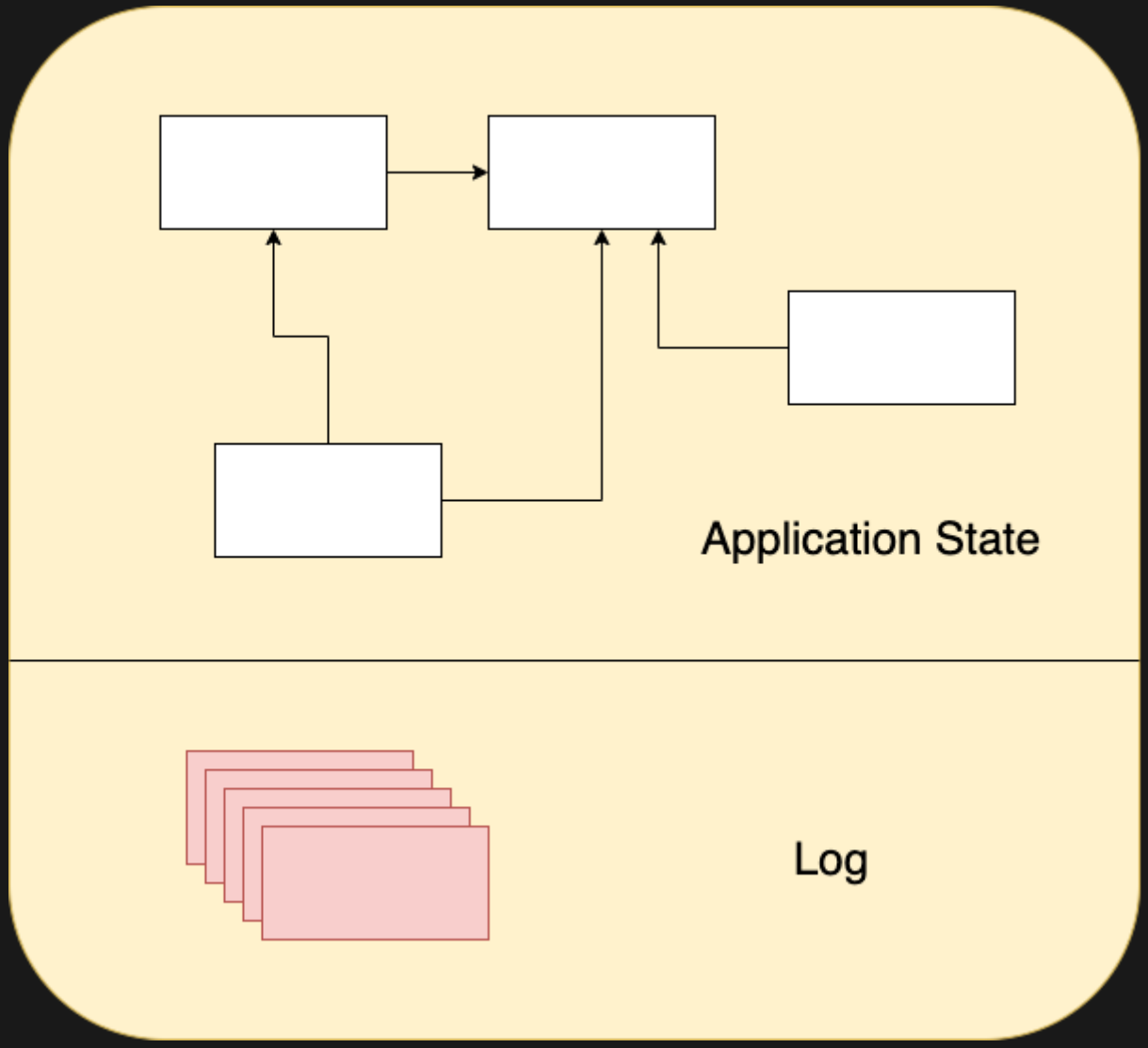
DRAWBACKS

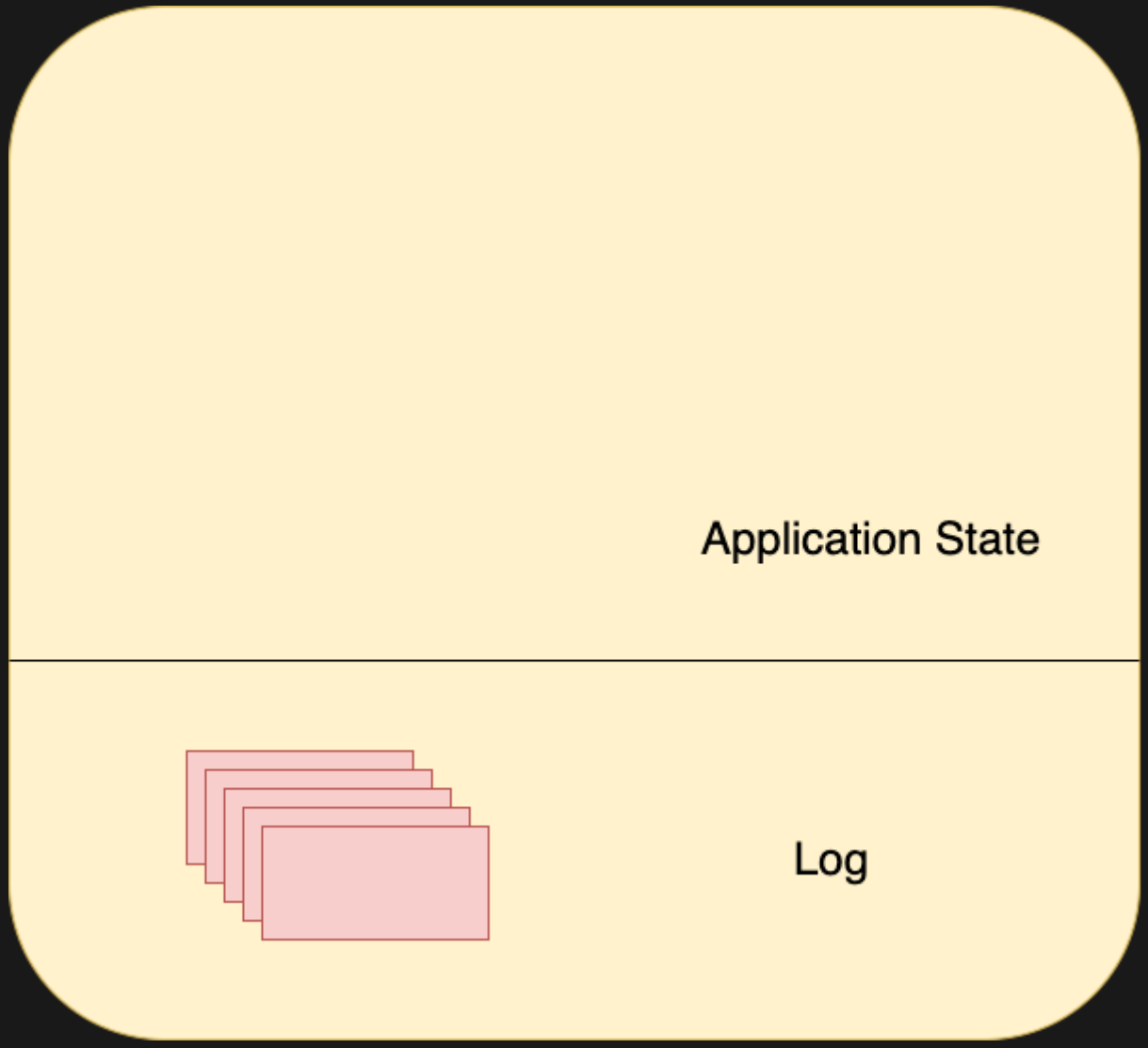
- CRUD systems perform update operations directly against a data store, which can slow down performance and responsiveness, and limit scalability, due to the processing overhead it requires.
- In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data.

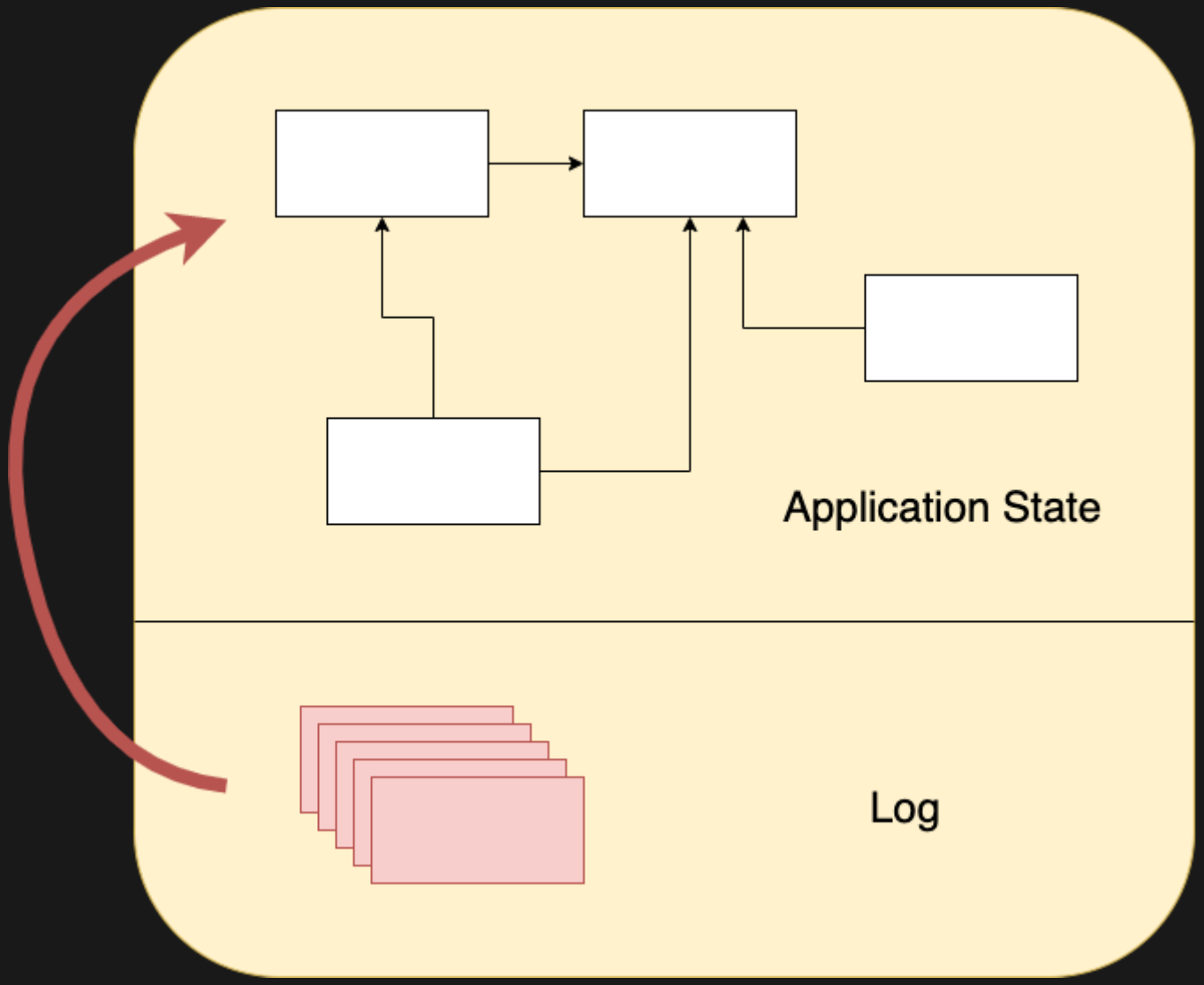
- Unless there's an additional auditing mechanism that records the details of each operation in a separate log, history is lost.

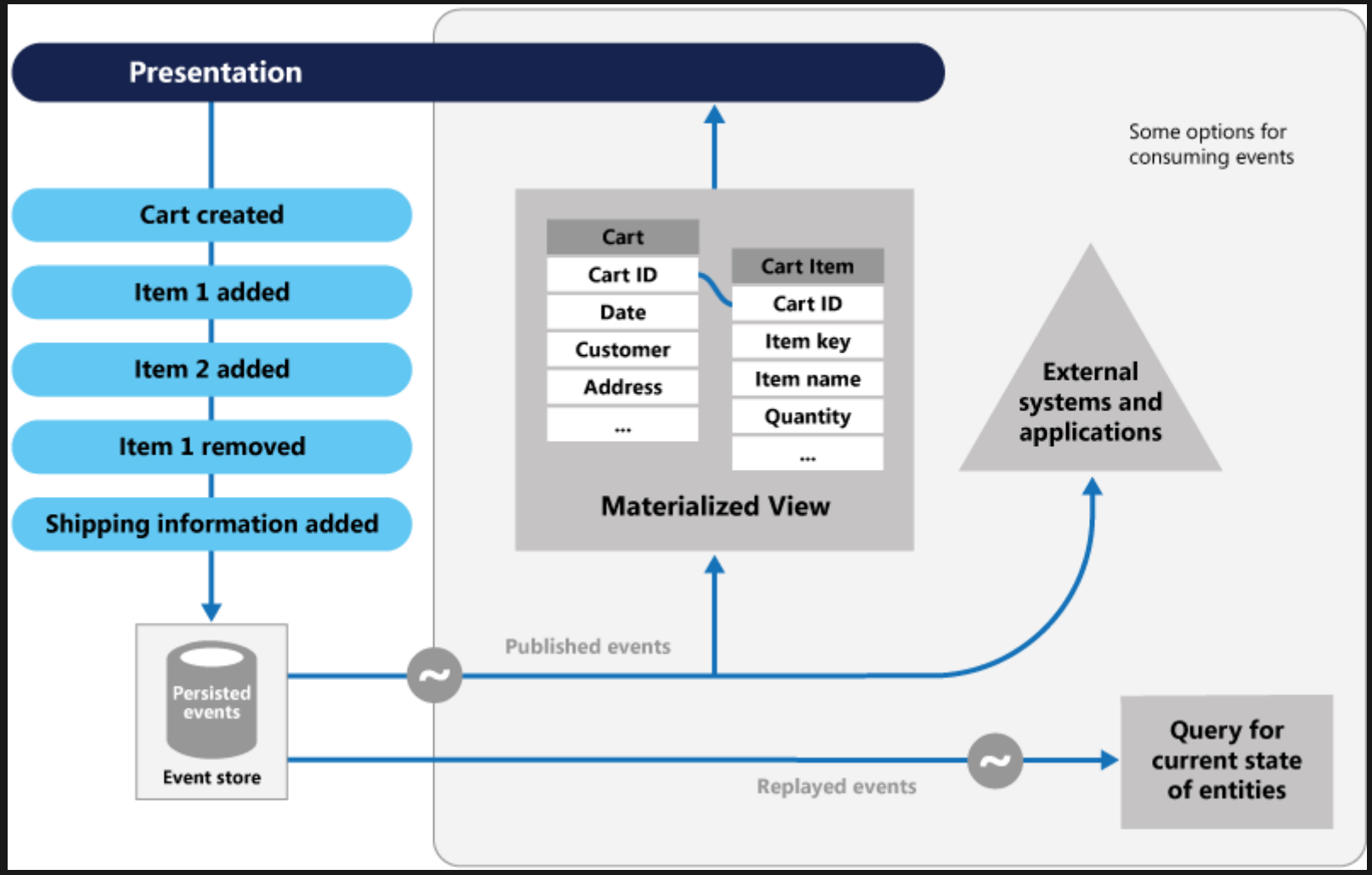
WHAT IS EVENT SOURCING

[...] handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they're persisted. Each event represents a set of changes to the data (such as `AddedItemToOrder`*).*


Application State
Log


Application State
Log


Application State
Log


Presentation
Some options for consuming events
Cart created
Item 1 added
Item 2 added
Item 1 removed
Shipping information added
Cart
Cart ID
Date
Customer
Address
...
Cart Item
Cart ID
Item key
Item name
Quantity
...
Materialized View
External systems and applications
Persisted events
Event store
Published events
Replayed events
Query for current state of entities

Microsoft

THE BENEFITS

- Ephemeral Data-Structures and the ability to take the existing events and view them in a new way
- Easier communication with domain experts - these events are part of the Ubiquitous language used
- Matches what we are currently modeling
- Reports becomes dead simple-you have a time machine
- Audit log / Historic state
- Event system-being reactive to events
- Replay events on a staging / development machine-debuggablity

Drawbacks

- Eventual Consistency-event gets created and other systems won't hear about it immediately
- Event upgrading
- New line of thinking-from CRUD to events

You don't need to implement every single Event Sourcing pattern to have an "Event Sourced" system

fagnerbrack

Basic event sourcing is quite simple to implement. All the bells and whistles people sell alongside event sourcing are hard - whether you do event sourcing or not.

codebje

Event sourcing is REALLY hard to figure out how to do “right.” A lot of getting it right is modeling knowledge/experience, understanding your domain.

linkmotif

THREE DIFFERENT ARCHITECTURE PATTERNS:

1. event sourcing (build your models based on immutable facts)
2. event driven (side affects triggered by messages, often delivered by queues)
3. workflow

1. EVENT SOURCING / EVENTS

The names of events are part of the Ubiquitous Language, part of DDD

Good names

- `ItemAdded`
- `CartCheckedOut`
- `CustomerCreated`

Bad Names

- `CreateCustomer`
- `StartCart`
- `AddItem`

Event Example

```
 1 class Events::Subscription::Activated < \
 2   Events::Subscription::BaseEvent
 3   data_attributes :stripe_key
 4
 5   def apply(subscription)
 6     subscription.stripe_key = stripe_key
 7     subscription.status = "active"
 8     subscription.activated_at = self.created_at
 9     subscription
10   end
11 end
```

Event Example

```
class Events::Subscription::Activated < \
  Events::Subscription::BaseEvent
  data_attributes :stripe_key

  def apply(subscription)
    subscription.stripe_key = stripe_key
    subscription.status = "active"
    subscription.activated_at = self.created_at
    subscription
  end
end
```

Event Example

```ruby
class Events::Subscription::Activated < \
  Events::Subscription::BaseEvent
  data_attributes :stripe_key

  def apply(subscription)
    subscription.stripe_key = stripe_key
    subscription.status = "active"
    subscription.activated_at = self.created_at
    subscription
  end
end
```

Event Example

```
class Events::Subscription::Activated < \
  Events::Subscription::BaseEvent
  data_attributes :stripe_key

  def apply(subscription)
    subscription.stripe_key = stripe_key
    subscription.status = "active"
    subscription.activated_at = self.created_at
    subscription
  end
end
```

What these events might look like in when stored

id	subscription_id	type	data (JSON)	metadata (JSON)	created_at
38920	12	Created	{ reward_id: 334, payment_source_key: 224, ...	{ user_id: 123 }	2018-04-06
38921	12	Activated	{ stripe_key: “sub_66123”	{ notification_id: 33456 }	2018-04-06

Aggregates

- Current state of the world
- Event Data + `apply` (Calculator) => Aggregate
- The "thing" that most other domains will interact with
- Materialized view, projection, snapshot, cached data....

```ruby
class Subscription < ApplicationRecord
  belongs_to :user
  belongs_to :reward
  has_many :events
end
```

```ruby
subscription = Subscription.find(12)
Events::Subscription::Activated.create!(
  subscription: subscription,
  stripe_key: "sub_66123",
  metadata: { notification_id: 33456 }
)
subscription.activated? # => true
```

```ruby
class Subscription < ApplicationRecord
  belongs_to :user
  belongs_to :reward
  has_many :events
end
```

```ruby
subscription = Subscription.find(12)
Events::Subscription::Activated.create!(
  subscription: subscription,
  stripe_key: "sub_66123",
  metadata: { notification_id: 33456 }
)
subscription.activated? # => true
```

```ruby
class Subscription < ApplicationRecord
  belongs_to :user
  belongs_to :reward
  has_many :events
end
```

```ruby
subscription = Subscription.find(12)
Events::Subscription::Activated.create!(
  subscription: subscription,
  stripe_key: "sub_66123",
  metadata: { notification_id: 33456 }
)
subscription.activated? # => true
```

What these aggregates might look like when stored

id	user_id	reward_id	status	stripe_key	created_at	activated_at	deactivated_at
12	123	234	active	sub_66123	2018-02-01	2018-02-01	NULL
13	345	234	unpaid	sub_66134	2018-02-02	2018-02-02	2018-04-06

2. EVENT DRIVEN

REACTORS AND DISPATCHERS

We react to events with `reactors` via the `dispatcher`

```
 1 class Dispatcher
 2   # ...
 3   on Events::Subscription::Activated,
 4     async: Reactor::Notifications::SubscriptionConfirmation
 5   on Events::Subscription::Activated,
 6     async: Reactor::Notifications::NewSubscriberNotificatio
 7   on Events::Subscription::Activated,
 8     trigger: ....
 9   # ...
10 end
```

We react to events with `reactors` via the `dispatcher`

```
 1 class Dispatcher
 2   # ...
 3   on Events::Subscription::Activated,
 4     async: Reactor::Notifications::SubscriptionConfirmation
 5   on Events::Subscription::Activated,
 6     async: Reactor::Notifications::NewSubscriberNotificatio
 7   on Events::Subscription::Activated,
 8     trigger: ....
 9   # ...
10 end
```

We react to events with `reactors` via the `dispatcher`

```
 1 class Dispatcher
 2   # ...
 3   on Events::Subscription::Activated,
 4     async: Reactor::Notifications::SubscriptionConfirmation
 5   on Events::Subscription::Activated,
 6     async: Reactor::Notifications::NewSubscriberNotificatio
 7   on Events::Subscription::Activated,
 8     trigger: ....
 9   # ...
10 end
```

Reactor - can be triggered async or synchronous

```
class Reactors::Notifications::SubscriptionConfirmation
  def self.call(event)
    SubscriberMailer.confirm_subscription(
      subscription_id: event.subscription_id
    ).deliver
 end
end
class Reactors::Notifications::NewSubscriberNotification
  def self.call(event)
    CreatorMailer.queue_new_subscriber(
      subscription_id: event.subscription_id
    ).deliver
  end
end
```

3. WORKFLOW - COMMANDS

Not part of Event Sourcing, but is a nice additional pattern.

COMMANDS

Responsble for:

- Validating attributes
- Validating that the action can be performed given the current state of the application
- Building and persisting the event

```ruby
class Commands::Subscription::Activate
  include Command
  attributes :subscription, :stripe_key, :metadata
  validate stripe_key, presence: true

  def build_event
    Events::Subscription::Activated.new(
      subscription: subscription,
      stripe_key: stripe_key,
      metadata: metadata
    )
  end
  def noop?
    subscription.activated?
  end
```

```ruby
class Commands::Subscription::Activate
  include Command
  attributes :subscription, :stripe_key, :metadata
  validate stripe_key, presence: true

  def build_event
    Events::Subscription::Activated.new(
      subscription: subscription,
      stripe_key: stripe_key,
      metadata: metadata
    )
  end
  def noop?
    subscription.activated?
  end
```

```ruby
class Commands::Subscription::Activate
  include Command
  attributes :subscription, :stripe_key, :metadata
  validate stripe_key, presence: true

  def build_event
    Events::Subscription::Activated.new(
      subscription: subscription,
      stripe_key: stripe_key,
      metadata: metadata
    )
  end
  def noop?
    subscription.activated?
  end
```

```ruby
  include Command
  attributes :subscription, :stripe_key, :metadata
  validate stripe_key, presence: true

  def build_event
    Events::Subscription::Activated.new(
      subscription: subscription,
      stripe_key: stripe_key,
      metadata: metadata
    )
  end
  def noop?
    subscription.activated?
  end
end
```

```ruby
  include Command
  attributes :subscription, :stripe_key, :metadata
  validate stripe_key, presence: true

  def build_event
    Events::Subscription::Activated.new(
      subscription: subscription,
      stripe_key: stripe_key,
      metadata: metadata
    )
  end
  def noop?
    subscription.activated?
  end
end
```

```ruby
Commands::Subscription::Activate.call(
  subscription: subscription,
  stripe_key: "sub_66123",
  metadata: { notification_id: 33456 }
)
# => <#Events::Subscription::Activated ...>
```

Back to events and how they apply

```
1  class BaseEvent
2    before_save :apply_and_persist
3    ....
4    private def apply_and_persist
5      # Lock! (all good, we're in the ActiveRecord callback c
6      aggregate.lock! if aggregate.persisted?
7
8      # Apply!
9      self.aggregate = apply(aggregate)
10
11     # Persist!
12     aggregate.save!
13     self.aggregate_id = aggregate.id if aggregate_id.nil?
14   end
15 end
```

Back to events and how they apply

```
1  class BaseEvent
2    before_save :apply_and_persist
3    ....
4    private def apply_and_persist
5      # Lock! (all good, we're in the ActiveRecord callback c
6      aggregate.lock! if aggregate.persisted?
7
8      # Apply!
9      self.aggregate = apply(aggregate)
10
11     # Persist!
12     aggregate.save!
13     self.aggregate_id = aggregate.id if aggregate_id.nil?
14   end
15 end
```

Back to events and how they apply

```
1  class BaseEvent
2    before_save :apply_and_persist
3    ....
4    private def apply_and_persist
5      # Lock! (all good, we're in the ActiveRecord callback c
6      aggregate.lock! if aggregate.persisted?
7
8      # Apply!
9      self.aggregate = apply(aggregate)
10
11     # Persist!
12     aggregate.save!
13     self.aggregate_id = aggregate.id if aggregate_id.nil?
14   end
15 end
```

Back to events and how they apply

```
1  class BaseEvent
2    before_save :apply_and_persist
3    ....
4    private def apply_and_persist
5      # Lock! (all good, we're in the ActiveRecord callback c
6      aggregate.lock! if aggregate.persisted?
7
8      # Apply!
9      self.aggregate = apply(aggregate)
10
11     # Persist!
12     aggregate.save!
13     self.aggregate_id = aggregate.id if aggregate_id.nil?
14   end
15 end
```

Back to events and how they apply

```
1  class BaseEvent
2    before_save :apply_and_persist
3    ....
4    private def apply_and_persist
5      # Lock! (all good, we're in the ActiveRecord callback c
6      aggregate.lock! if aggregate.persisted?
7
8      # Apply!
9      self.aggregate = apply(aggregate)
10
11     # Persist!
12     aggregate.save!
13     self.aggregate_id = aggregate.id if aggregate_id.nil?
14   end
15 end
```

Back to events and how they apply

```
 1 class BaseEvent
 2   before_save :apply_and_persist
 3   ....
 4   private def apply_and_persist
 5     # Lock! (all good, we're in the ActiveRecord callback c
 6     aggregate.lock! if aggregate.persisted?
 7
 8     # Apply!
 9     self.aggregate = apply(aggregate)
10
11     # Persist!
12     aggregate.save!
13     self.aggregate_id = aggregate.id if aggregate_id.nil?
14   end
15 end
```

Back to events and how they apply

```
1  class BaseEvent
2    before_save :apply_and_persist
3    ....
4    private def apply_and_persist
5      # Lock! (all good, we're in the ActiveRecord callback c
6      aggregate.lock! if aggregate.persisted?
7
8      # Apply!
9      self.aggregate = apply(aggregate)
10
11     # Persist!
12     aggregate.save!
13     self.aggregate_id = aggregate.id if aggregate_id.nil?
14   end
15 end
```

DEMO

QUESTIONS

RESOURCES / WANT TO LEARN MORE

- These slides/demo application
- A short talk from Martin Fowler that explains the concepts really well
- Kickstarter blog post
- Kickstarter example application
- Blog post that explains Kickerstarter blog post
- Accompanying repo from the above blog post
- Event Sourcing explained with diagrams
- Gradually migrating from CRUD to Event Sourcing