EVENT SOURCING

in production systems

LARS VONK & BOB FORMA

zilverline

Let's take a look how we currently develop software

```ruby
class Invoices < Sinatra::Base
  post '/pay' do
    invoice = Invoice.find(params[:invoice_id])
    invoice.pay(pay_date: Date.today)
    invoice.save!
    redirect "/invoices/paid"
  end
end
```

```ruby
class Invoices < Sinatra::Base
  post '/pay' do
    invoice = Invoice.find(params[:invoice_id])
    invoice.pay(pay_date: Date.today)
    invoice.save!
    redirect "/invoices/paid"
  end
end
```

```erb
<table><tbody>
  <% Invoice.where(status: :paid).each do |invoice| %>
    <tr>
      <td><%= invoice.invoice_number %></td>
      <td><%= invoice.total_amount %></td>
      <td>...</td>
    </tr>
    <% end %>
</tbody></table>
```

One model to rule them all

```
class Invoices < Sinatra::Base
  post '/pay' do
    invoice = Invoice.find(params[:invoice_id])
    invoice.pay(pay_date: Date.today)
    invoice.save!
    redirect "/invoices/paid"
  end
end
```

```
<table><tbody>
  <% Invoice.where(status: :paid).each do |invoice| %>
    <tr>
      <td><%= invoice.invoice_number %></td>
      <td><%= invoice.total_amount %></td>
      <td>...</td>
    </tr>
    <% end %>
</tbody></table>
```

Mix of concerns

- Querying
- Transactions (commands)
- Reporting

	invoice_number character varying(255)	invoice_status character varying(255)	invoice_date_sent date
1	201209-001	sent	2012-09-07
2	201206-002	sent	2012-06-22
3	201209-021	paid	2012-09-07
4	201209-001	sent	2012-09-21

	invoice_number character varying(255)	invoice_status character varying(255)	invoice_date_sent date
1	201209-001	sent	2012-09-07
2	201206-002	sent	2012-06-22
3	201209-021	paid	2012-09-07
4	201209-001	sent	2012-09-21

How did this Invoice get to status :sent?

	invoice_number character varying(255)	invoice_status character varying(255)	invoice_date_sent date
1	201209-001	sent	2012-09-07
2	201206-002	sent	2012-06-22
3	201209-021	paid	2012-09-07
4	201209-001	sent	2012-09-21

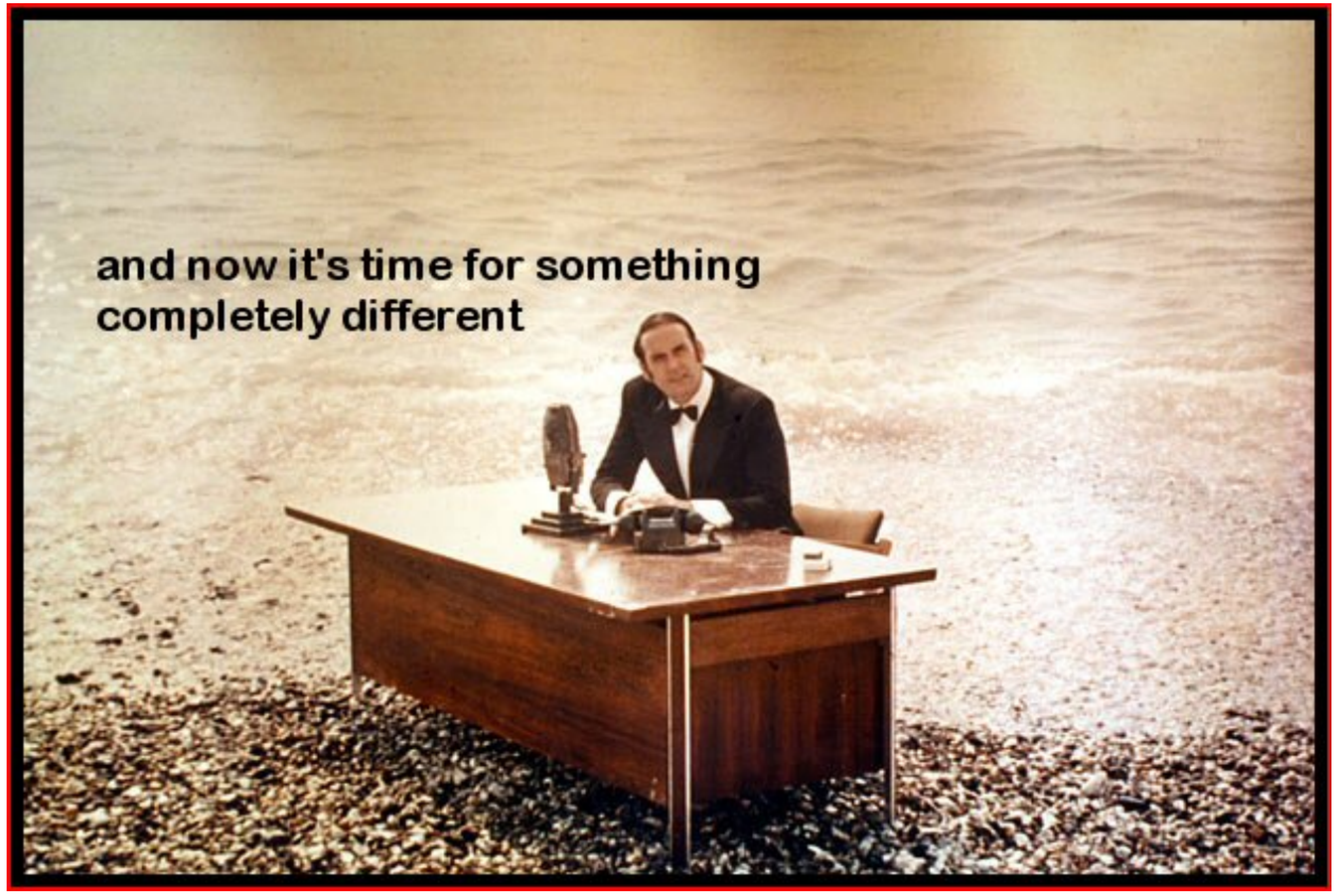
- How did this Invoice get to status :sent?

- How many items are deleted from the shopping cart *just* before an order is placed?

Information is lost

	invoice_number character varying(255)	invoice_status character varying(255)	invoice_date_sent date
1	201209-001	sent	2012-09-07
2	201206-002	sent	2012-06-22
3	201209-021	paid	2012-09-07
4	201209-001	sent	2012-09-21

- How did this Invoice get to status :sent?
- How many items are deleted from the shopping cart *just* before an order is placed?


and now it's time for something
completely different

Event Sourcing

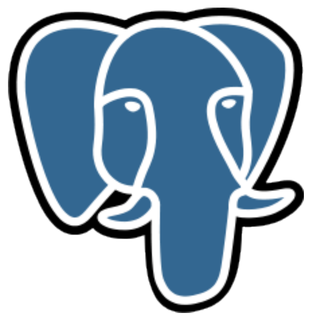
Capture all changes to an application state as a sequence of events

martinfowler.com/eeaDev/EventSourcing.html – Martin Fowler

Or maybe not so different...

Rabobank

github
SOCIAL CODING



Trivial example

Current state

id	amount	status	recipient
ed3456f	121	sent	bob@zilverline.com

Trivial example

Current state

id	amount	status	recipient
ed3456f	121	sent	bob@zilverline.com

Event sourced

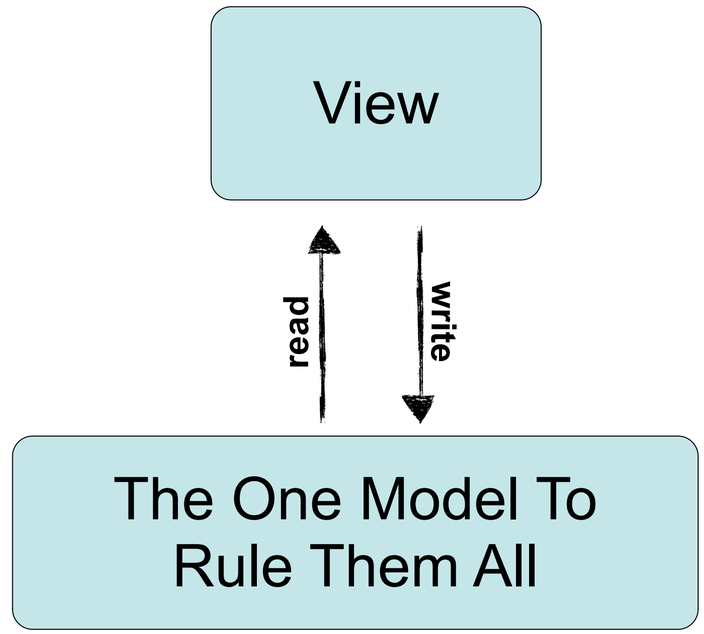
id	event_type	event_json
ed3456f	InvoiceCreatedEvent	{id:“ed3456f”}
ed3456f	LineItemAddedEvent	{id:“ed3456f”, line_item: {id: 1, amount: 100, vat: 21}}
ed3456f	InvoiceSentEvent	{id:“ed3456f”, to: “bob@zilverline.com”}

How many line items are deleted just before...

Event sourced

id	event_type	event_json
ed3456f	InvoiceCreatedEvent	{id:“ed3456f”}
ed3456f	LineItemAddedEvent	{id:“ed3456f”, line_item: {id: 1, amount: 100, vat: 21}}
ed3456f	LineItemRemovedEvent	{id:“ed3456f”, line_item_id: 1}
ed3456f	LineItemAddedEvent	{id:“ed3456f”, line_item: {id: 2, amount: 150, vat: 21}}
ed3456f	InvoiceSentEvent	{id:“ed3456f”, to: “bob@zilverline.com”}

So how do you query stuff?



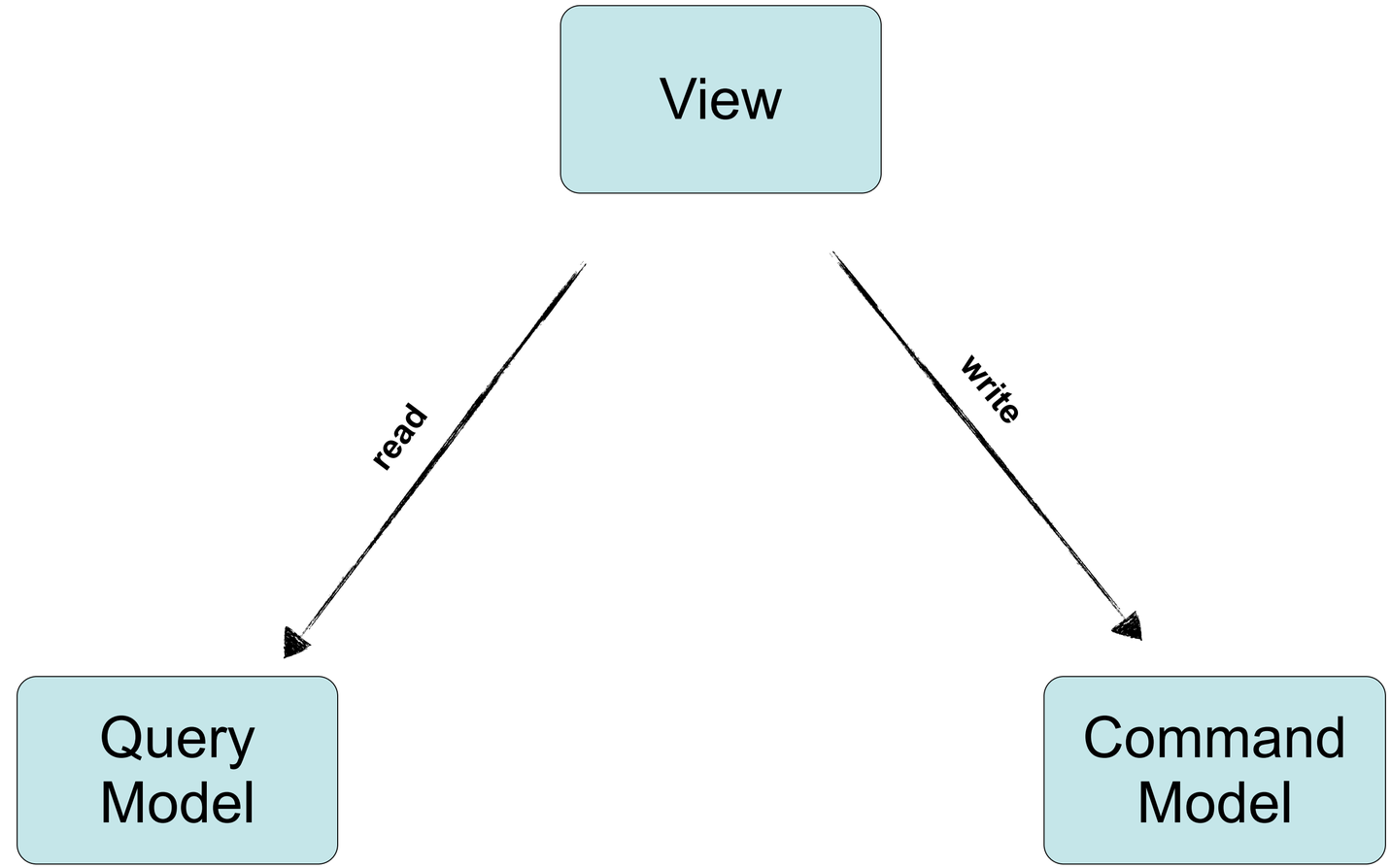
So how do you query stuff?

View

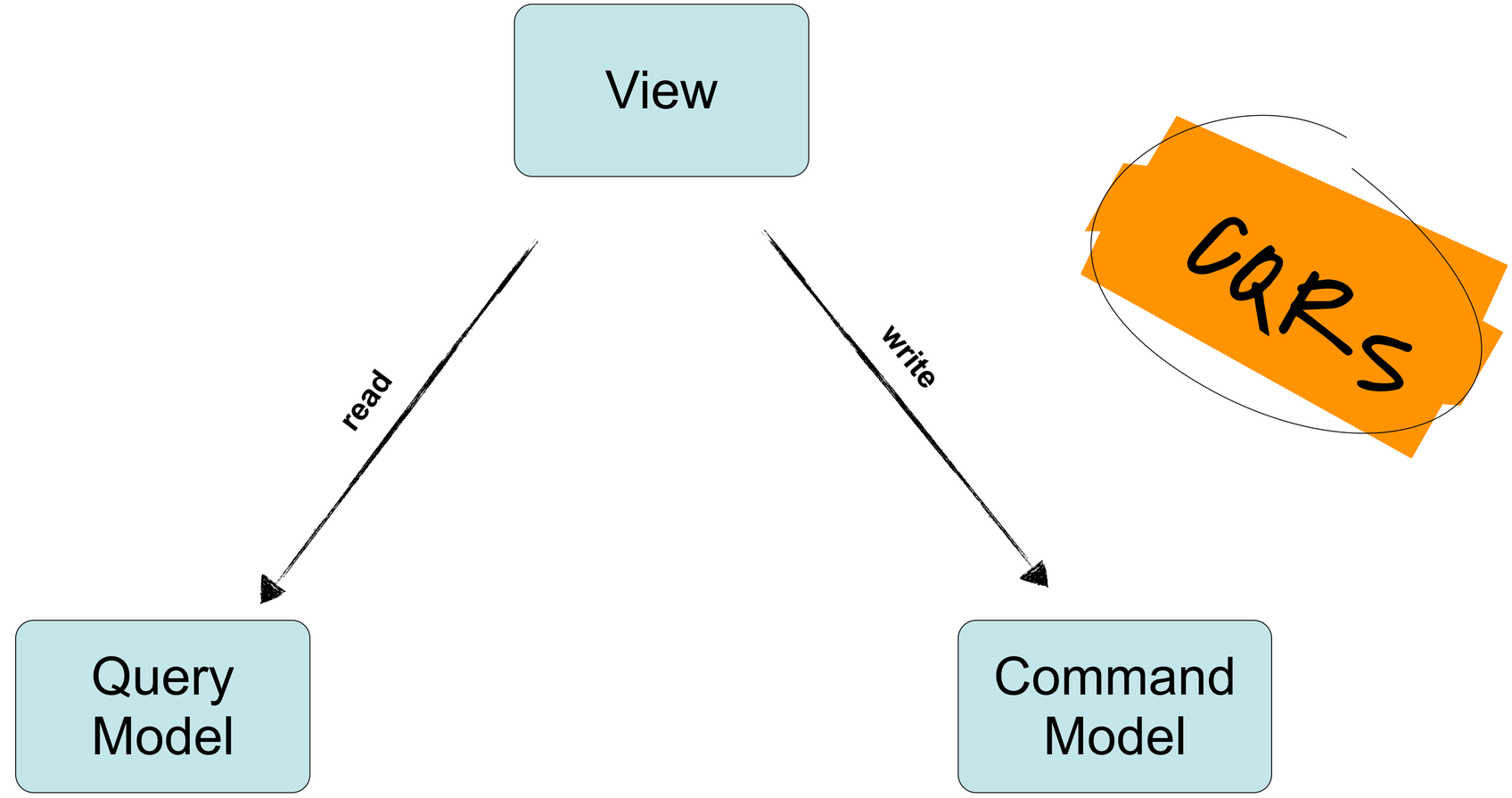
Query Model

Command Model

So how do you query stuff?



So how do you query stuff?



So how do you query stuff?

View

read

write

‘generates using events’

Query Model

Command Model

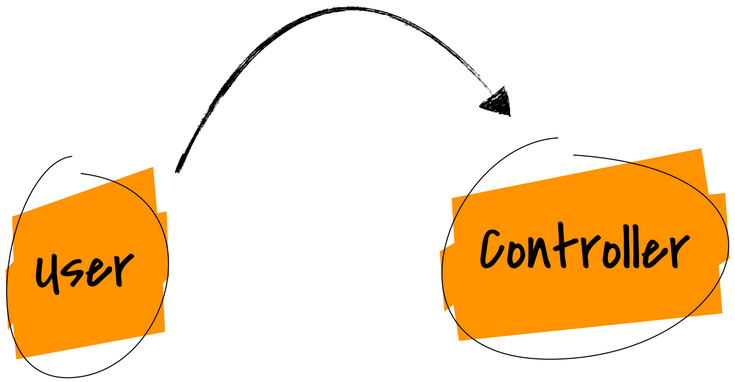
CQRS

Concrete example...

Concrete example...

Concrete example...

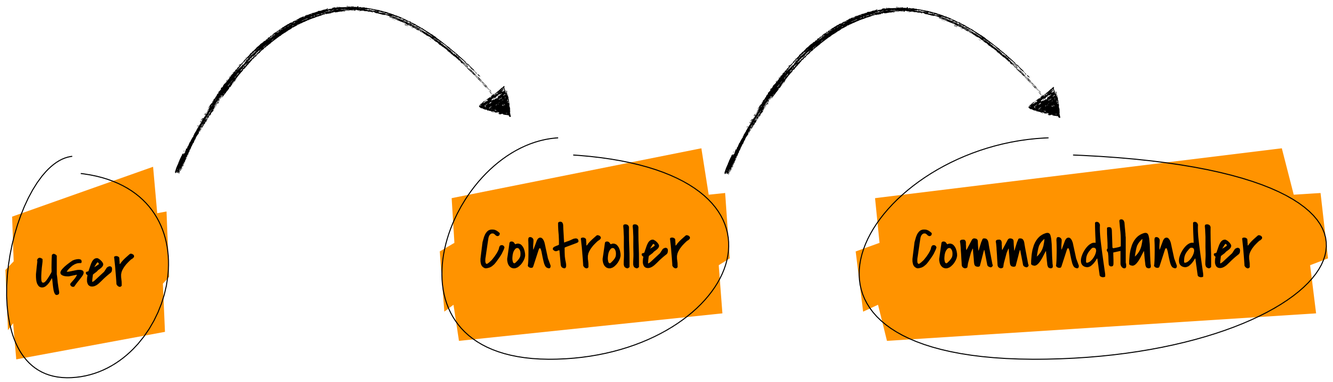
POST /invoices/:id/send



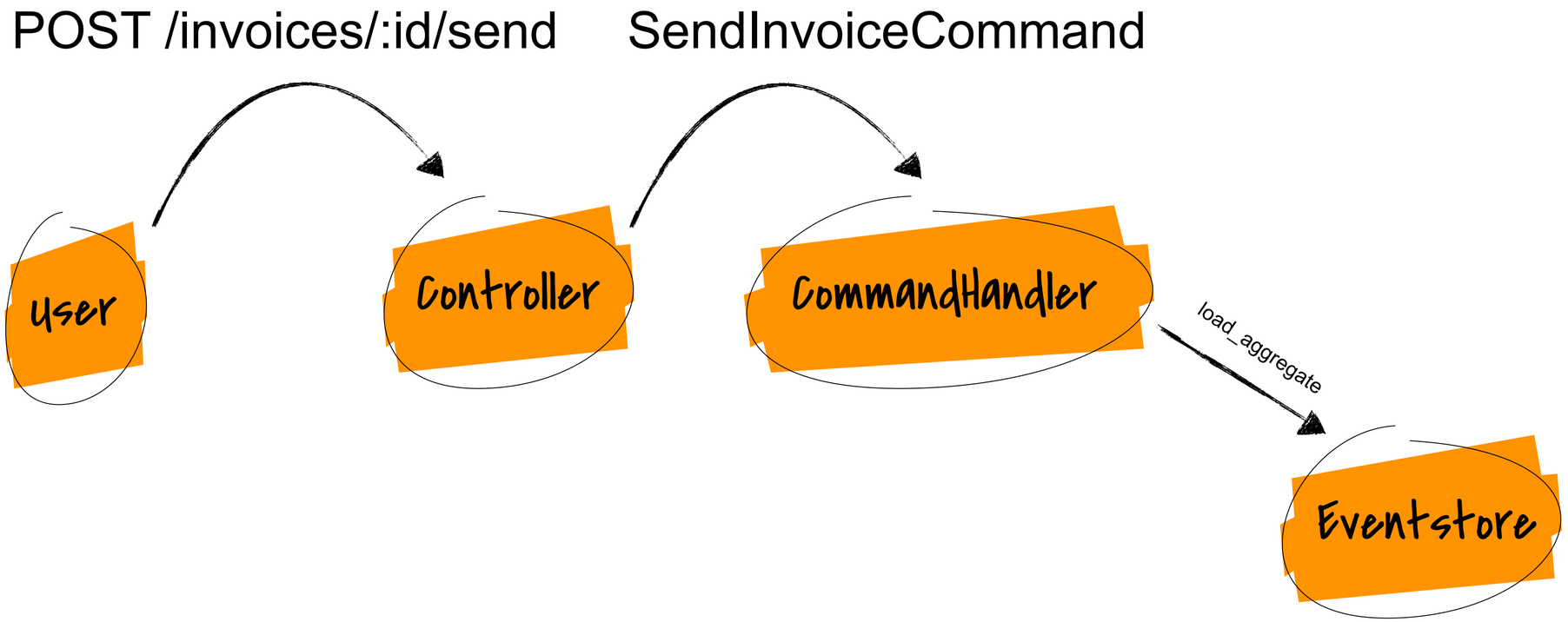
Concrete example...

POST /invoices/:id/send

SendInvoiceCommand



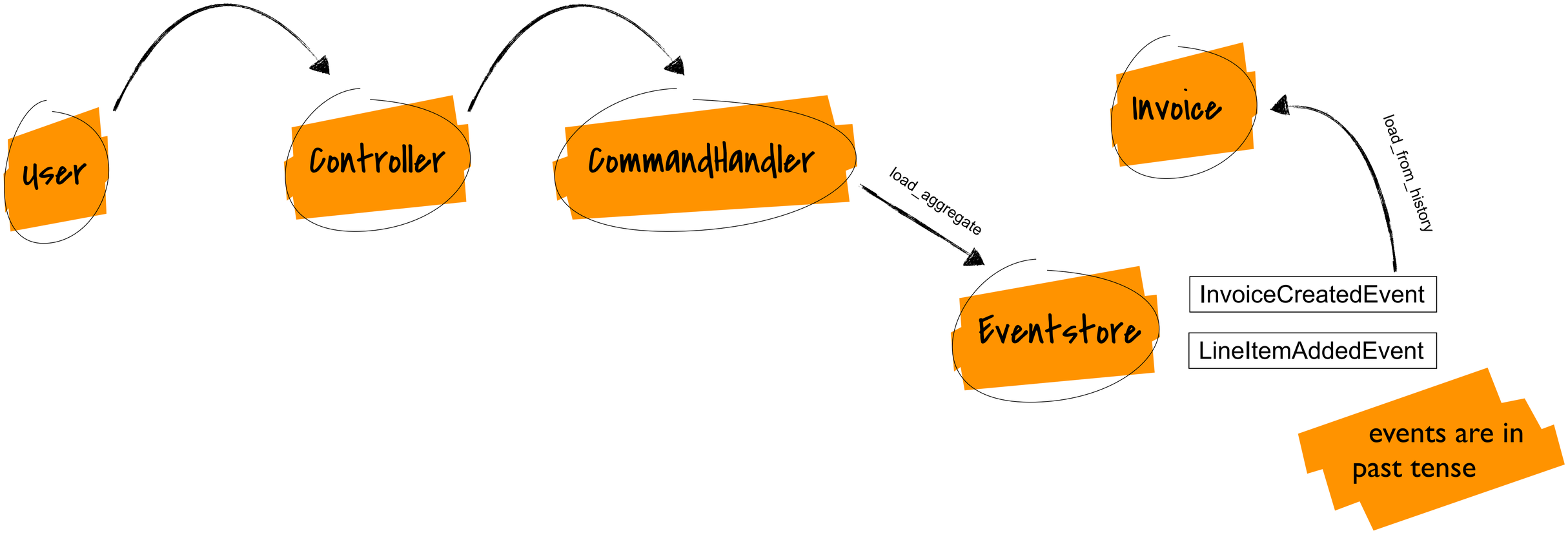
Concrete example...



Concrete example...

POST /invoices/:id/send

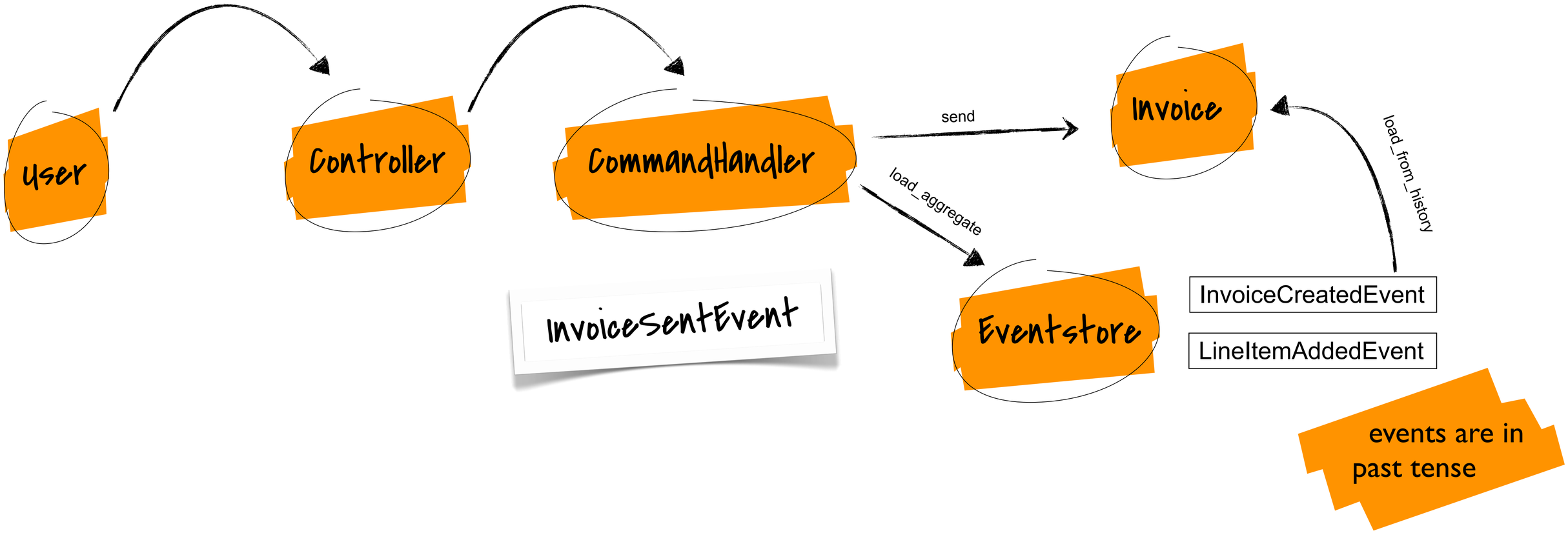
SendInvoiceCommand



Concrete example...

POST /invoices/:id/send

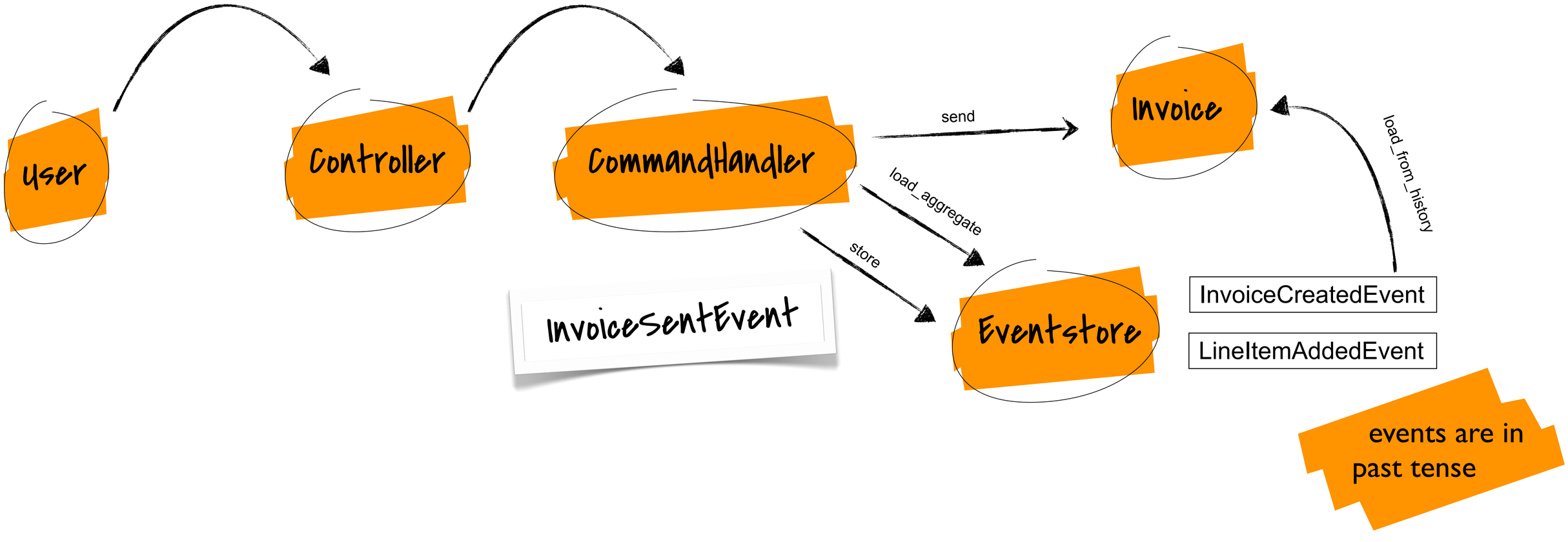
SendInvoiceCommand



Concrete example...

POST /invoices/:id/send

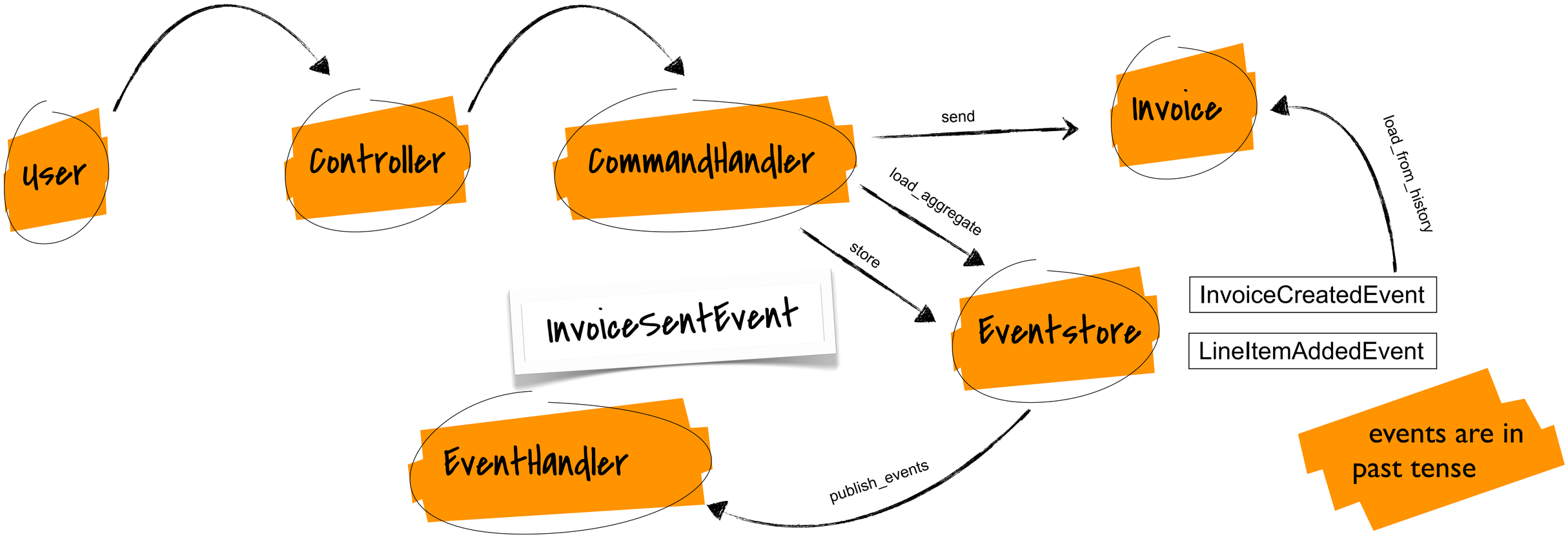
SendInvoiceCommand



Concrete example...

POST /invoices/:id/send

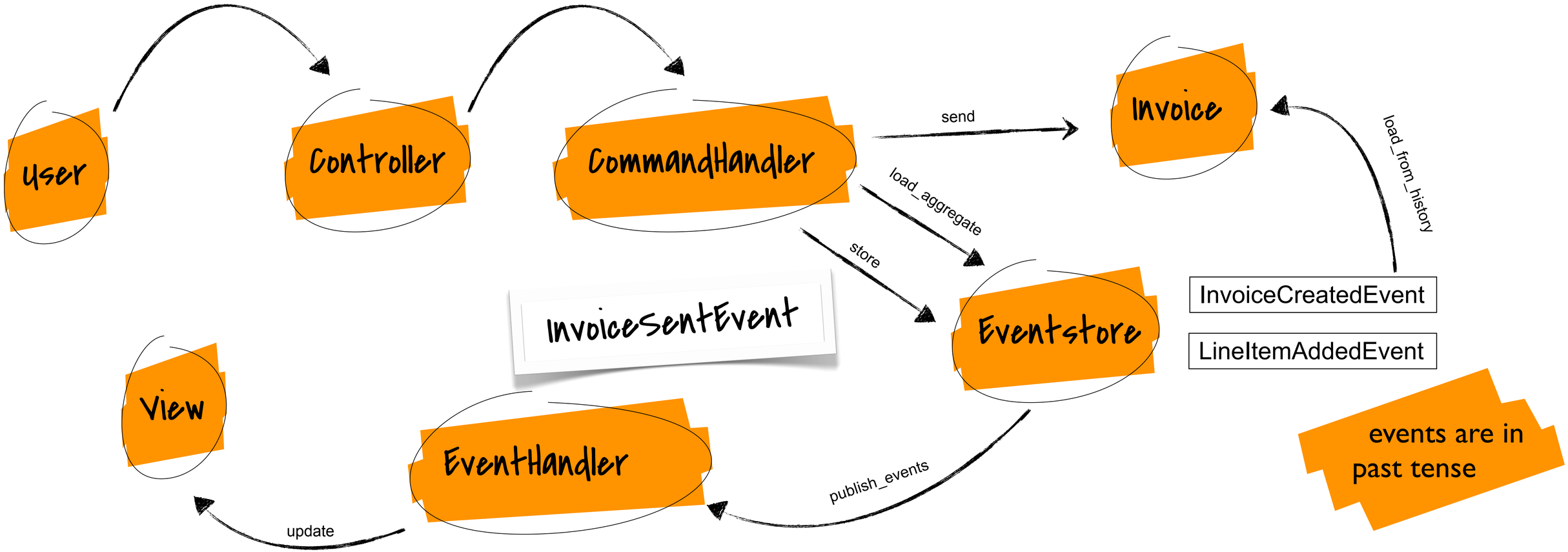
SendInvoiceCommand



Concrete example...

POST /invoices/:id/send

SendInvoiceCommand



Why Event Sourcing

Part 1: For the business

Why Event Sourcing

Part 1: For the business

To capture user intent

Why Event Sourcing

Part 1: For the business

- To capture user intent
- Historic information

Why Event Sourcing

Part 1: For the business

- To capture user intent
- Historic information
- Storage decoupling

Why Event Sourcing

Part 1: For the business

- To capture user intent
- Historic information
- Storage decoupling
- Defer decisions

Why Event Sourcing

Part 2: For us nerds

Why Event Sourcing

Part 2: For us nerds

Tell, don’t ask

Why Event Sourcing

Part 2: For us nerds

- Tell, don’t ask
- Isolated domain tests

Why Event Sourcing

Part 2: For us nerds

- Tell, don't ask
- Isolated domain tests

```
All examples were filtered out; ignoring {:focus=>true}
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................

Finished in 1.63 seconds
326 examples, 0 failures
```

Why Event Sourcing

Part 2: For us nerds

- Tell, don’t ask
- Isolated domain tests
- Debug information

```
All examples were filtered out; ignoring {:focus=>true}
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................

Finished in 1.63 seconds
326 examples, 0 failures
```

Why Event Sourcing

Part 2: For us nerds

- Tell, don’t ask
- Isolated domain tests
- Debug information
- Conflict resolution

```
All examples were filtered out; ignoring {:focus=>true}
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................

Finished in 1.63 seconds
326 examples, 0 failures
```

It is not a Silver bullet

It is not a Silver bullet

- Not for CRUD

It is not a Silver bullet

- Not for CRUD
- Learning curve

It is not a Silver bullet

- Not for CRUD
- Learning curve
- More work to do simple stuff

Case study Freemle.com

- Financial administration system
- Integrates with various systems
- Web-based
- Event source approved

THE INTERNALS

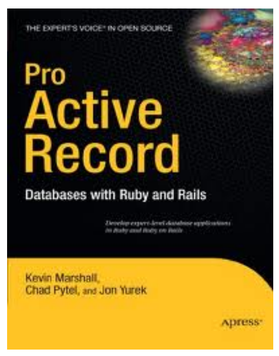
Stack

Ruby
A Programmer's Best Friend

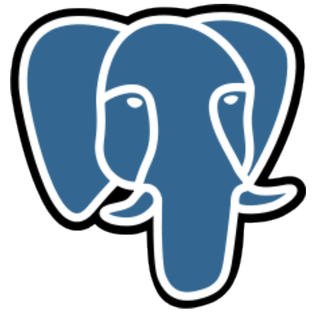
APACHE HTTP SERVER

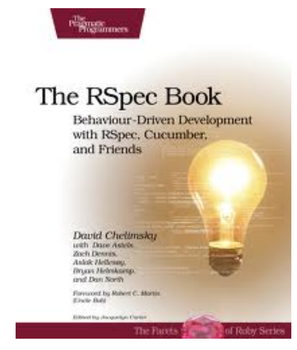
Sinatra

passenger
Rails deployment that just works

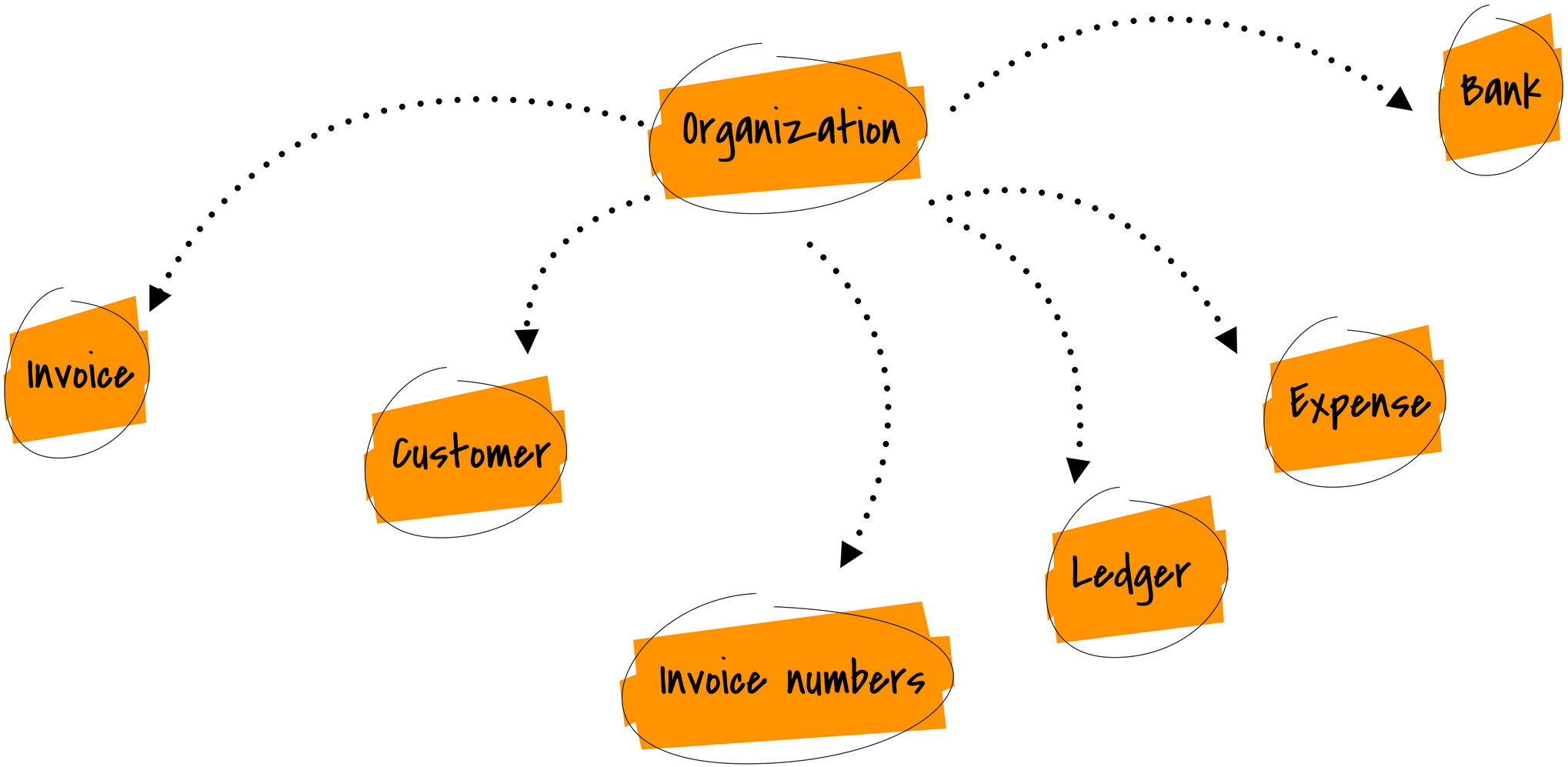


Scala



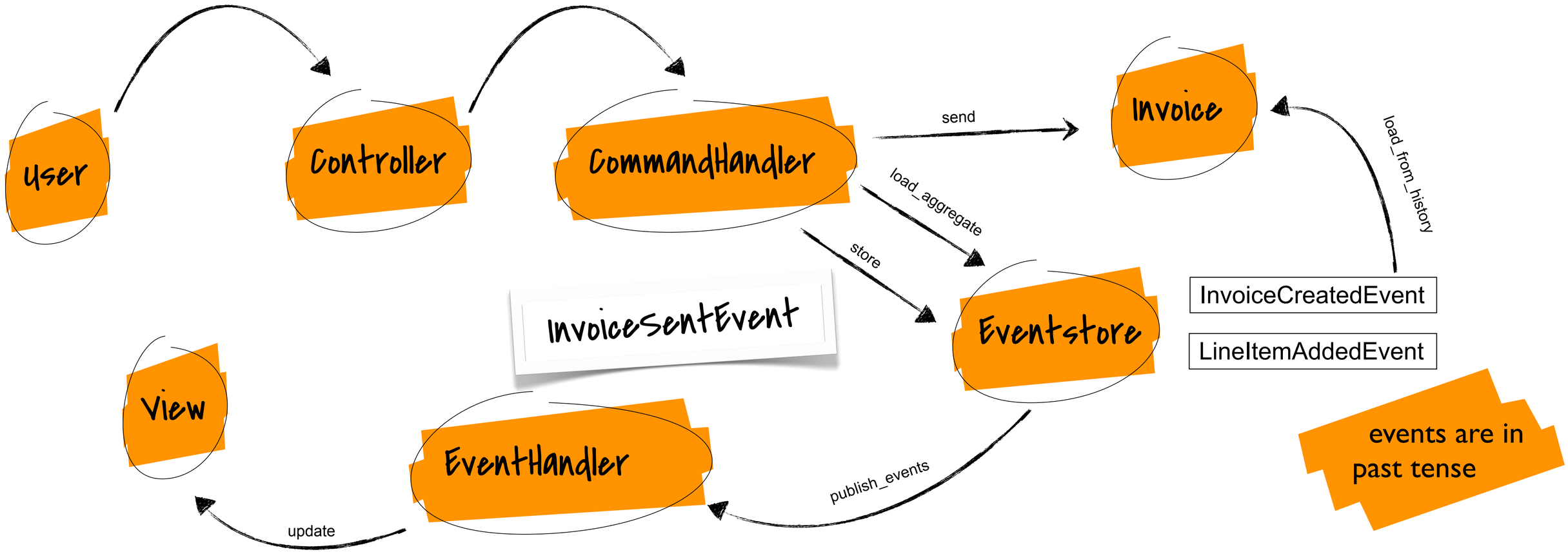


Domain


Organization
Bank
Invoice
Customer
Invoice numbers
Ledger
Expense

Remember this picture?

POST /invoices/:id/send SendInvoiceCommand



User wants to mark an invoice as paid

```
class Invoices < Sinatra::Base

  post '/pay' do
    execute MarkInvoicePaidCommand, on: invoice, with: {pay_date: Date.today}
    redirect "/invoices/id/#{aggregate_id}"
  end

end
```

Request is translated to specific command

```
class MarkInvoicePaidCommand < UpdateCommand
  include InvoiceCommand

  attrs pay_date: Date
  validates_presence_of :pay_date
  validate :valid_date
end
```

Propagates to correct Domain object

```
class InvoiceCommandHandler < BaseCommandHandler

  on MarkInvoicePaidCommand do |command|
    do_with_aggregate(command, Invoice) do |invoice|
      invoice.mark_paid(command.pay_date)
    end
  end

end
```

Retrieves all events for Aggregate

```
class EventStore

  def load_events(aggregate_id)
    to_events EventRecord.where(aggregate_id: aggregate_id).order(:sequence_number).all
  end

end
```

What does an event look like?

```ruby
class Event
  include ActiveModel::Serializers::JSON

  attrs aggregate_id: String,
        sequence_number: Integer,
        created_at: DateTime,
        organization_id: String
end
```

```json
{"aggregate_id":"273b29de-8b62-41f8-968f-9ef2f4f87649",
 "created_at":"2012-09-14T16:37:00+02:00",
 "organization_id":"88cca57a-c177-4b89-a941-3f4c1fae0f93",
 "pay_date":"2012-09-14",
 "sequence_number":8}
```

```ruby
class InvoiceCreatedEvent < Event
  attrs creditor: Creditor, recipient: Recipient, line_items: array(LineItem)
end

class InvoiceSentEvent < Event
  attrs send_date: Date, mail_data: MailData,
        customer_id: String, invoice_date: Date,
        invoice_due_date: Date
end
```

Rebuild yourself from history

```
class AggregateRoot

  attr_reader :id, :uncommitted_events

  def self.load_from_history(events)
    aggregate_root = allocate()
    aggregate_root.load_from_history(events)
    aggregate_root
  end

end
```

Events are applied

```ruby
class Invoice < AggregateRoot

  private

  def on_invoice_sent_event(event)
    @invoice_sent = true
    @due_amount = @invoice_total + @invoice_vat_total
    @due_date = event.invoice_due_date
    @invoice_date = event.invoice_date
  end

end
```

Propagates to correct Domain object

```
class InvoiceCommandHandler < BaseCommandHandler

  on MarkInvoicePaidCommand do |command|
    do_with_aggregate(command, Invoice) do |invoice|
      invoice.mark_paid(command.pay_date)
    end
  end

end
```

Tell the invoice it has been paid

```
class Invoice < AggregateRoot

  def mark_paid(pay_date)
    raise("pay_date is mandatory. aggregate_id: #@id") unless pay_date
    raise("can only mark an invoice as paid when it is sent. aggregate_id: #@id") unless @invoice_sent
    raise("cannot pay an invoice that is already paid. aggregate_id: #@id") if @marked_paid
    apply InvoiceMarkedPaidEvent, pay_date: pay_date
  end

end
```

Apply events on yourself

```
class AggregateRoot

  def apply(event, params={})
    event = build_event(event, params)
    apply_message_to_self(event)
    @uncommitted_events << event
    @sequence_number += 1
  end

end
```

Update your state

```
class Invoice < AggregateRoot

  private

  def on_invoice_marked_paid_event(event)
    @marked_paid = true
  end

end
```

Commit the events in the EventStore

```
class EventStore

  def commit_events(command, events)
    store_events command, events
    publish_events events
  end

end
```

Publish the events to registered Handlers

```
class EventStore

  private

  def publish_events(events)
    events.each do |event|
      @event_handlers.each do |handler|
        handler.handle_message event
      end
    end
  end

end
```

Update view model

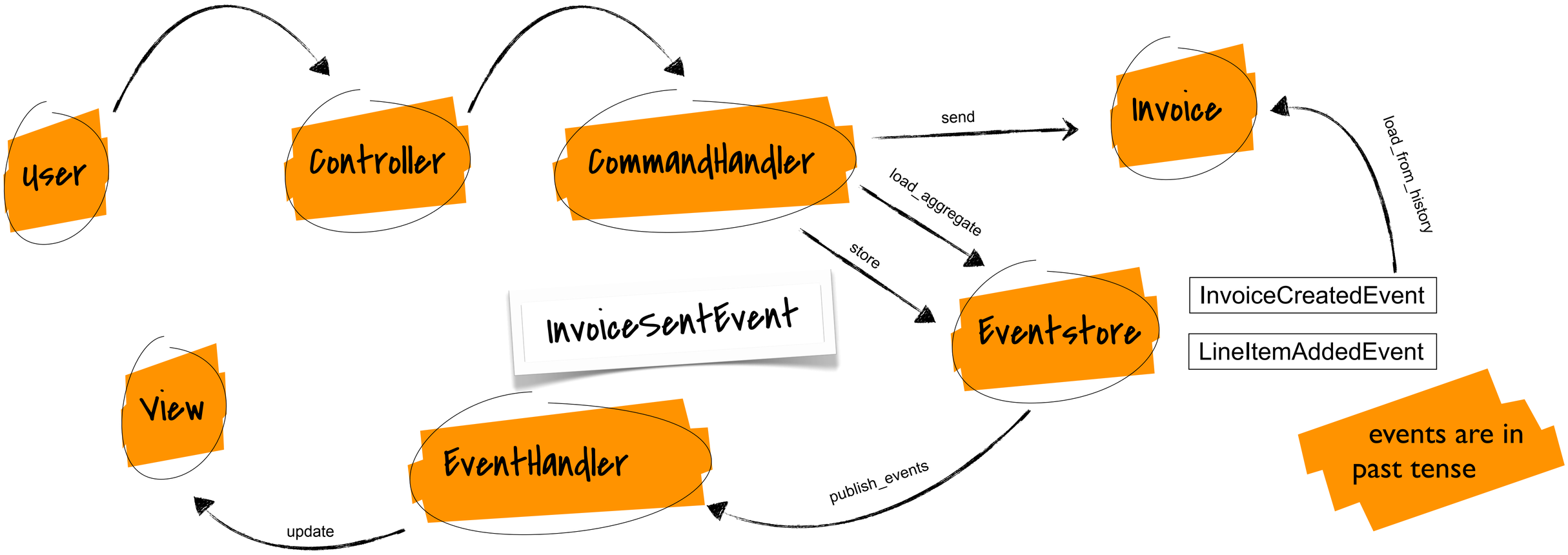
```
class InvoiceEventHandler

  on InvoiceMarkedPaidEvent do |event|
    update_record(InvoiceRecord, event) do |invoice|
      invoice.invoice_status = :paid
      invoice.invoice_pay_date = event.pay_date
    end
  end

end
```

Render view

```
<table><tbody>
  <% InvoiceRecord.where(status: :paid).each do |invoice_record| %>
    <tr>
      <td><%= invoice_record.invoice_number %></td>
      <td><%= invoice_record.total_amount %></td>
      <td>...</td>
    </tr>
    <% end %>
</tbody></table>
```

Summary

POST /invoices/:id/send SendInvoiceCommand



EXAMPLE HISTORIC INTEL

Introducing the timeline

What was easy?

What was easy?

- Querying (separate model)

What was easy?

- Querying (separate model)
- Implementing business logic

What was easy?

- Querying (separate model)
- Implementing business logic
- Measuring feature usage

What was easy?

- Querying (separate model)
- Implementing business logic
- Measuring feature usage
- Read performance (optimized view model)

What was easy?

- Querying (separate model)
- Implementing business logic
- Measuring feature usage
- Read performance (optimized view model)
- No BDUF for the data model

What was hard?

What was hard?

Mental model change (thinking in events)

What was hard?

- Mental model change (thinking in events)
- Event upcasting

What was hard?

- Mental model change (thinking in events)
- Event upcasting
- Defining events

Open challenges

Open challenges

Replay all events performance

Open challenges

- ~~Replay all events performance~~

Open challenges

- ~~Replay all events performance~~
- Asynchronous view model update

Open challenges

- ~~Replay all events performance~~
- Asynchronous view model update
- Zero downtime deployments

Open challenges

- ~~Replay all events performance~~
- Asynchronous view model update
- Zero downtime deployments
- Scaling

Questions?