Mastering Modern Payments

Using Stripe with Rails

by Pete Keen

Sample Chapter

State and History

So far in our little example app we can buy and sell downloadable products using Stripe. We're not keeping much information in our own database, though. We can't easily see how much we've earned, we can't see how big Stripe's cut has been. Ideally our application's database would keep track of this. The mantra with financial transactions should always be "trust and verify". To that end we should be tracking sales through each stage of the process, from the point the customer clicks the buy button all the way through to a possible refund. We should know, at any given moment, what state a transaction is in and it's entire history.

State Machines

The first step of tracking is to turn each transaction into a state machine. A state machine is simply a formal definition of what states an object can be in and the transitions that can happen to get it between states. At any given moment an object can only be in a single state. For example, consider a subway turnstile. Normally it's locked. When you put a coin in or swipe your card, it unlocks. Then when you pass through it locks itself again. We could model that like this using a gem called AASM:

```
class Turnstile

  include AASM

  aasm do
    state :locked, initial: true
    state :unlocked

    event :pay do
      transitions from: :locked, to: :unlocked
    end

    event :use do
      transitions from: :unlocked, to: :locked
    end
  end
end
```

AASM is the successor to a previous gem named `acts_as_state_machine` which was hard-coded to ActiveRecord objects and had a few problems. AASM fixes those problems and lets you describe state machines inside any class, not just ActiveRecord. As you can see, it implements a simple DSL for states and events. AASM will create a few methods on instances of Turnstile, things like `pay!` and `use!` to trigger the corresponding events and `locked?` and `unlocked?` to ask about the state.

AASM can also be used with ActiveRecord, just like it's predecessor. Let's begin by adding some more fields to `Sale`:

```
$ rails g migration AddFieldsToSale \
    state:string \
    stripe_id:string \
    stripe_token:string \
    card_expiration:string \
    error:text \
    fee_amount:integer
$ rake db:migrate
```

Now, add `aasm` to your Gemfile and run `bundle install`:

```
gem 'aasm'
```

The Sale state machine will have four possible states:

- pending means we just created the record
- processing means we're in the middle of processing
- finished means we're done talking to Stripe and everything went well
- errored means that we're done talking to Stripe and there was an error

It'll also have a few different events for the transaction: `process`, `finish`, and `fail`. Let's describe this using `aasm`:

```
class Sale < ActiveRecord::Base
  attr_accessible \
    :email,
    :guid,
    :product_id,
    :state,
    :stripe_id,
    :stripe_token,
    :card_expiration,
    :error,
    :fee_amount

  before_save :populate_guid

  include AASM

  aasm column: 'state', skip_validation_on_save: true do
    state :pending, initial: true
    state :processing
    state :finished
    state :errored

    event :process, after: :charge_card do
      transitions from: :pending, to: :processing
    end
```

```
  event :finish do
    transitions from: :processing, to: :finished
  end

  event :fail do
    transitions from: :processing, to: :errored
  end
end

def charge_card
  begin
    save!
    charge = Stripe::Charge.create(
      amount: self.amount,
      currency: "usd",
      card: self.stripe_token,
      description: self.email,
    )
    self.update_attributes(
      stripe_id:       charge.id,
      card_expiration: Date.new(charge.card.exp_year, Charge.card.exp_month, 1),
      fee_amount:      charge.fee
    )
    self.finish!
  rescue Stripe::Error => e
    self.update_attributes(error: e.message)
```

```
      self.fail!
    end
  end

  def populate_guid
    if new_record?
      self.guid = SecureRandom.uuid()
    end
  end
end
```

Inside the `aasm` block, every state we described earlier gets a `state` declaration, and every event gets an `event` declaration. Notice that the `:pending` state is what the record will be created with. Also, notice that the transition from `:pending` to `:processing` has an `:after` callback declared. After AASM updates the `state` property and saves the record it will call the `charge_card` method. AASM will automatically create scopes, so for example you can find how many finished records there are with `Sale.finished.count`.

Notice that the stuff about charging the card moved into the model. This adheres to the Fat Model Skinny Controller principle, where all of the logic lives in the model and the controller just drives it. Here's how `TransactionsController#create` method looks now:

```
def create
  @product = Product.where(permalink: params[:permalink]).first
  raise ActionController::RoutingError.new("Not found") unless @product

  token = params[:stripeToken]
  sale = Sale.create(
    product_id:   @product.id,
    amount:       @product.price,
    email:        params[:email],
    stripe_token: token
  )
  sale.process!
  if sale.finished?
    redirect_to pickup_url(guid: sale.guid)
  else
    flash[:alert] = sale.error
    render :new
  end
end
```

Not that much different, really. We create the Sale object, and then instead of doing the Stripe processing in the controller we call the `process!` method that `aasm` creates. If the sale is finished we'll redirect to the pickup url. If isn't finished we assume it's errored, so we render out the `new` view with the error.

It would be nice to see all of this information we're saving now. Let's change the `Sales#show` template to dump out all of the fields:

```
<p id="notice"><%= notice %></p>
<table>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
<% @sale.attributes.sort.each do |key, value| %>
<tr>
<td><%= key %></td>
<td><%= value %></td>
</tr>>
<% end %>
</table>

<%= link_to 'Stripe', "https://manage.stripe.com/payments/#{@sale.stripe_id}" %>
<%= link_to 'Back', sales_path %>
```

Notice that we're deep-linking directly into Stripe's management interface. That will give you one-click access to everything that Stripe knows about this transaction, as well as a button to refund the payment.

Audit Trail

Another thing that will be very useful is an audit trail that tells us every change to a record. Every time AASM updates the `state` field, every change that happens during the charging process, every change to the object at all. Why would you even want this? It could potentially be quite a bit of work for not much payoff. Let me tell you a story. The company I work for deals with payments from about eight different payment providers. Each one of them is custom, one-off code that shares very little with the rest of the system. One day we started getting complaints about payments going missing, so started digging. However, not only were we not keeping history in the database, we weren't even comprehensively logging things. It took two software developers almost a week straight to finally figure out how payments were going missing on our end, by cross checking what little information we did have with what the payment providers had. We ended up giving away a bunch of gifts to keep everyone happy, not to mention the opportunity cost of having developers tracking things down. With a proper audit trail on our end we would have instantly been able to see when and where things were getting lost.

There are a few different schools of thought on how to implement audit trails. The classical way would be to use database triggers to write copies of the database rows into an audit table. This has the advantage of working whether you use the ActiveRecord interface or straight SQL queries, but it's really hard to implement properly. The easiest way to implement audit trails that I've found is to use a gem named Paper Trail. Paper Trail monitors changes on a record using ActiveRecord's lifecycle events and will serialize the state of the object before the change and stuff it into a `versions` table. It has convenient methods for navigating versions, which we'll use to display the history of the record in an admin interface later.

First, add the gem to your Gemfile:

```
gem 'paper_trail', '~> 2'
```

Install the gem, which will generate a migration for you, and run the migration:

```
$ rails generate paper_trail:install --with-changes
$ rake db:migrate
```

And now add `has_paper_trail` to the Sale model:

```
class Sale < ActiveRecord::Base
  has_paper_trail

  # ... rest of Sale from before
end
```

`has_paper_trail` takes a bunch of options for things like specifiying which lifecycle events to monitor, which fields to include and which to ignore, etc. which are all described in it's documentation. The defaults should usually be fine.

Here's some simple code for the `SalesController#show` action to display the history of the sale. In `app/views/sales/show.html.erb`:

```
<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Event</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<%= @sale.versions.each do |version| %>
<tr>
<td><%= version.created_at %></td>
<td><%= version.event %></td>
<td>
<% version.changeset.sort.each do |key, value| %>
<b><%= key %></b>: <%= value[0] %> to <%= value[1] %><br>
<% end %>
</td>
</tr>
<% end %>
</tbody>
</table>
```

Each change will have a timestamp, the event, and a block of changes, one row for each column that changed in that update. For a typical completed sale there will be three rows,

"pending", "processing", and "completed" with all of the information from Stripe. By examining the audit trail for a clean transaction you can do things like get rough performance numbers for your interactions with Stripe, and if you ever have broken transactions you can see when things went wrong and more importantly how things went wrong which will better help you fix them.