Initial Application

In this chapter we're going to create a simple rails application so we have something to work with for later chapters. All of the rest of the examples in the guide will be based on this app. You can either follow along with the instructions or use the app in the `sales` directory in the example code.

A note on versions. The example app will be using Rails 3.2.13 and PostgreSQL 9.2.

Our app will sell downloadable products. Users will be able to create products and customers will be able to buy them, and we'll keep track of sales so we can do reporting later. Customers will be able to come back and download their purchases multiple times. We'll three models:

- **Product**, representing a product that we're going to be selling.
- **User**, for logging in and managing products
- **Sale**, to represent each individual customer purchase

Boilerplate

Let's create an initial application:

```
$ rails new sales --database postgresql --test-framework=rspec
$ cd sales
$ createuser -s sales
$ rake db:setup
$ rake db:migrate
$ rake test
```

I'm going to use PostgreSQL for the example app because that's what I know best, it's what Heroku provides for free, and it's what I suggest to everyone who

asks. If you want to use a different database, feel free to substitute. Any `ActiveRecord`-compatible database should be sufficient.

Authentication

Eventually we're going to want to be able to authenticate users and admins. The example is going to use a gem named Devise which handles everything user-related out of the box. Add it to your `Gemfile`:

```
gem 'devise', '~> 2.2.4'
```

then run bundler and set up Devise:

```
$ bundle install
$ rails generate devise:install
```

At this point you have to do some manual configuration. Add this to `config/environments/development.rb`:

```
config.action_mailer.default_url_options = { :host => 'localhost:300
```

This to `config/routes.rb`:

```
root :to => 'products#index'
```

and this in `app/views/layouts/application.html.erb` right after the `body` tag:

```
<p class="notice"><%= notice %></p>
<p class="alert"><%= alert %></p>
```

Also, you'll want to delete `public/index.html` because it gets in Devise's way.

Now, let's create a User model for devise to work with:

```
$ rails generate devise User
```

```
$ rake db:migrate
```

Open up `app/controllers/application_controller.rb` and add this line, which will secure everything by default:

```
before_filter :authenticate_user!
```

You'll need to a user so you can actually log in to the site. Fire up `rails console` and type:

```
User.create!(email: 'you@example.com', password: 'password', passwor
```

Models

Our sales site needs something to sell, so let's create a product model:

```
$ rails g scaffold Product name:string permalink:string description:
$ rake db:migrate
```

`name` and `description` will actually get displayed to the customer, `permalink` and `download_url` will be used later. Open up `app/models/product.rb` and change it too look like this:

```
class Product < ActiveRecord::Base
  attr_accessible :description, :name, :permalink, :price, :user_id,

  belongs_to :user
end
```

The sales site needs a way to track, you know, sales. Let's make a Sale model too.

```
$ rails g scaffold Sale email:string guid:string product_id:integer
$ rake db:migrate
```

Open up `app/models/sale.rb` and make it look like this:

```
class Sale < ActiveRecord::Base
  attr_accessible :email, :product_id

  belongs_to :product

  before_save :populate_guid

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

We're using a GUID here so that when we eventually allow the user to look at their transaction they won't see the `id`, which means they won't be able to guess the next ID in the sequence and potentially see someone else's transaction.

Deploying

Heroku is bar-none the fastest way to get a Rails app deployed into a production environment, so that's what we're going to use throughout the guide. If you already have a deployment system for your application by all means use that. First, download and install the Heroku Toolbelt for your platform. Make sure you `heroku login` to set your credentials.

We have to add one more config option. Because of how Heroku handles databases, our app does not have access to the database while compiling assets. Conveniently Rails has an option to turn off connecting to the database at that time. Set it in `config/application.rb`:

```
config.assets.initialize_on_precompile = false
```

Next, create an appliation and deploy the example code to it:

```
$ heroku create
$ git init .
$ git add .
$ git commit -m 'Initial commit'
$ git push heroku master
$ heroku run rake db:migrate
$ heroku run console # create a new user like we did before in the l
$ heroku restart web # restart the web dyno to pick up the database
$ heroku open
```

You should see a login prompt from Devise. Go ahead and login and create a few products. We'll get to buying and downloading in the next chapter.

Next

Now we have a very simple application which will be enough to get going. We have things to sell and a way to track sales, as well as authentication so not just anybody can come muck with our stuff. Next, we'll run through the simplest Stripe integration and actually sell some stuff.

The Simplest Stripe Integration

This chapter is going to be a whirlwind integration with Stripe. It's going to be simple and nothing you haven't seen before, but it'll give us something to build on for the next few sections. This is loosely based on Stripe's own Rails Checkout Guide.

Remember that this application is going to be selling digital downloads, so we're going to have two actions:

- **buy** where we create a Sale record and actually charge the customer,
- **pickup** where the customer can download their product.

Basic Setup

First, add the Stripe gem to your Gemfile:

```
gem 'stripe', git: 'https://github.com/stripe/stripe-ruby'
```

And then run `bundle install`.

We'll also need to set up the Stripe keys:

```
# in config/initializers/stripe.rb
Rails.configuration.stripe = {
  publishable_key: ENV['STRIPE_PUBLISHABLE_KEY'],
  secret_key:      ENV['STRIPE_SECRET_KEY'],
}

Stripe.api_key = Rails.configuration.stripe.secret_key
```

Note that we're getting the keys from the environment. This is for two reasons:

first, because it lets us easily have different keys for testing and for production; second, and more importantly, it means we don't have to hardcode any potentially dangerous security credentials. Putting the keys directly in your code means that anyone with access to your code base can make Stripe transactions with your account.

Controller

Next, let's create a new controller named `Transactions` where our Stripe-related logic will live:

```
# in app/controllers/transactions_controller.rb

class TransactionsController < ApplicationController
  skip_before_filter :authenticate_user!, only: [:new, :create]

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

  def show
    @sale = Sale.where(guid: params[:guid]).first
    raise ActionController::RoutingError.new("Not found") unless @sa
    @product = @sale.product
  end

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

    token = params[:stripeToken]

    begin
      charge = Stripe::Charge.create(
        amount:      product.price,
        currency:    "usd",
        card:        token,
        description: params[:email]
```

```
      )
      sale = Sale.create!(product_id: product.id, email: params[:ema
      redirect_to pickup_url(guid: sale.guid)
    rescue Stripe::CardError => e
      # The card has been declined or some other error has occured
      @error = e
      render :new
    end
  end

end
```

`#new` is just a placeholder for rendering the corresponding view. The real action happens in `#create` where we look up the product and actually charge the customer. In the last chapter we included a `permalink` attribute in `Product` and we use that here to look up the product, mainly because it'll let us generate nicer-looking URLs. If there's an error we display the `#new` action again. If there's not we redirect to a route named `pickup`.

Routes

The routes for transactions are pretty simple. Add this to `config/routes.rb`:

```
match '/buy/:permalink' => 'transactions#new',    via: :get,  as: :b
match '/buy/:permalink' => 'transactions#create', via: :post, as: :b
match '/pickup/:guid'   => 'transactions#show',   via: :get,  as: :p
```

Why not RESTful URLs?

RESTful URLs are great if you're building a reusable API, but for this example we're writing a pretty simple website and the customer-facing URLs should look good. If you want to use resources, feel free to adjust the examples.

Views

Time to set up the views. Put this in `app/views/transactions/new.html.erb`:

```
<h1><%= @product.name %></h1>

<%= @product.description.html_safe %>

<% if @error %>
<%= @error %>
<% end %>

<p>Price: <%= formatted_price(@product.price) %></p>

<%= form_tag buy_path(permalink: @product.permalink) do %>
  <script src="https://checkout.stripe.com/v2/checkout.js" class="st
          data-key="<%= Rails.configuration.stripe[:publishable_key
          data-description="<%= @product.name %>"
          data-amount="<%= @product.price %>"></script>
<% end %>
```

Drop the definition for `formatted_price` into `app/helpers/application_helper.rb`:

```
def formatted_price(amount)
  sprintf("$%0.2f", amount / 100.0)
end
```

This is a very simple example of a product purchase page with the product's name, description, and a Stripe button using `checkout.js`. Notice that we just drop the description in as html, so make sure that's locked down. We're rendering this for the `#create` action, too, so if there's an error we'll display it above the checkout button.

The view for `#pickup` is even simpler, since it basically just has to display the product's download link. In `app/views/transactions/pickup.html.erb`:

```
<h1>Download <%= @product.name %></h1>

<p>Thanks for buying "<%= @product.name %>". You can download your p
```

```
<p><%= link_to "Download", @product.download_url %></p>
```

Deploy

Add all the new files to git and commit, then run:

```
$ heroku config:add STRIPE_PUBLISHABLE_KEY=pk_your_test_publishable_
$ git push heroku master
```

You should be able to navigate to `https://your-app.herokuapp.com/buy/some_permalink` and click the buy button to buy and download a product.

Next

In this chapter we built (almost) the simplest Stripe integration possible. In the next chapter we're going to cover why and how to save more information about the transaction to our own database.

PCI Compliance

One of the biggest reasons to choose Stripe over other ways of processing credit cards is that they minimize your exposure to *PCI compliance*. PCI stands for "Payment Card Industry", if you were wondering.

Note: I'm not an expert in PCI compliance and this chapter shouldn't be interpreted as legal advice about it. Rather, this is background information and advice on how to implement Stripe's guidelines. If you have questions, please ask Stripe.

Back in the early 2000s the credit card industry got together and decided on a whole bunch of interrelated standards for how to secure a payment system such that personally identifiable information, especially credit card numbers, is unlikely to be leaked to the outside world. For example, before the era of PCI compliance it was common for your unencrypted credit card number to be tacked onto your user record in an application database. Also, it was typical for sites to use plain HTTP for payment forms instead of HTTPS.

Now, though, both of those practices along with a host of others would get your merchant account cancelled. Being PCI compliant means that you adhere to all of the practices that apply to the way you process credit cards. Stripe is certified Service Provider Level 1, which means they have to have store credit card information encrypted in separate machines, possibly in separate data centers, than all of the rest of their infrastructure. It also means that nobody internal to Stripe can access unencrypted credit card numbers. Their software makes charges based on your API calls by sending information to an exclusive set of providers, entirely hidden from employees.

Stripe and PCI

The real revolutionary part of how Stripe works is in how they reduce your compliance scope as a merchant. Before Stripe, a typical online merchant would have a normal HTML form on their website where customers would put in their credit card information. This form would post to the merchant's server, where they would take the credit card info and pass it along to their *gateway service*, which would then talk to all of the various banks and things and then eventually deposit the money into their *merchant account*. This means, among other things, that each merchant would have to become PCI certified, even if they weren't storing the credit card info anywhere in their system. Theoretically, an attacker could stick some code into a merchant's payment processing system and divert credit card numbers. Or, if the merchant's site wasn't using HTTPS they could perform a man-in-the-middle attack and capture credit card info that way.

Stripe, with `stripe.js`, makes all of this irrelevant. When you create a form using `stripe.js` or `checkout.js` loaded from Stripe's servers, none of the customer's credit card info is sent through your servers. The javascript that gets injected into your form instead sends that info to Stripe's servers over HTTPS, where they turn it into a single-use *token*. Your server can then use that token to refer to a customer's credit card without having seen it at all.

The only thing you as a merchant have to do to be PCI compliant in this situation is to make sure you're serving up your payment-related pages over HTTPS. As long as you're loading `stripe.js` from Stripe via HTTPS into a secure webpage which POSTs to a secure endpoint, and you make sure not to put `names` on any of the credit-card-related fields in the form (only fields with `names` get POSTed), you don't have to worry about PCI compliance at all.

Implementing HTTPS with Rails

Rails after v3.1 makes forcing visitors to HTTPS incredibly easy. In `config/environments/production.rb`:

```
config.force_ssl = true
```

This will redirect all non-https requests to your website to the secure endpoint automatically on production. For this example it's all we need to do because Heroku provides what's called a "wildcard ssl certificate" for all apps accessed at `herokuapp.com`. However, if you're using your own URL you'll need to get your own certificate (generally around $10 per year) and install it with Heroku, which will run $20 per month. These costs vary, of course, if you're using a different hosting provider. Most Amazon-based cloud providers will charge $20 because that's how much an Elastic Load Balancer costs.

Buying a Certificate

There are many different places where you can buy a certificate. I've had good luck buying them through my registrar [Namecheap.com][namecheap]. The steps are:

- Generate a private key
- Using your private key, generate a Certificate Signing Request
- Send the CSR to Namecheap
- Receive your shiny new certificate
- Remove the passphrase from your certificate so that the webserver can use it.

First make sure you have `openssl` installed on your machine. It comes installed by default on Mac OS X but on Linux you may have to install it from your package manager.

Generate a Private Key

```
$ openssl genrsa -out example.com.key 2048
```

This generates a 2048 bit RSA key and stores it in `example.com.key`.

Generate a Certificate Signing Request

```
$ openssl req -new -key example.com.key -out example.com.csr
```

OpenSSL will ask you a bunch of questions now. Fill them in like the prompts, but when you get to the `Common Name` question, use the exact name of web server. Note that this really does have to be an exact match, so if you want to secure, say, `www.example.com`. that's what you should use. Putting just `example.com` won't work. For a wildcard certificate you'd put `*.example.com`, which would let you secure `foo.example.com` and `bar.example.com`, but those cost quite a bit more than individual certificates.

Also, make sure to leave extra attributes including the challenge password blank.

Validate your new CSR

```
$ openssl req -noout -text -in example.com.csr
```

This will print out a bunch of information about your certificate. You can ignore almost all of it, but pay attention to the line `CN=example.com`. This should match what you put in for your server name in the `Common Name` field.

Buy the actual certificate

Head on over to Namecheap's SSL page. Here you're presented with a bunch of different options presented in what they feel is least-secure to most-secure list. I generally buy the cheapest option because they're all pretty much the same in the $10 range. If you want, you can get EV1 certification which will give you the green bar in Safari and Firefox. You'll have to do some more paperwork to get it, though. For now, let's just get the cheapest Comodo certificate.

Go through checkout and pay and you'll get sent to a page where you can pick your server type and paste your CSR. For Heroku you should choose the "Other" option in the server dropdown. Open your CSR up and paste the entire contents into the text box, then hit Next.

Namecheap will give you a list of email addresses to choose from. This is where it's going to send the verification email that contains a link you have to click to proceed through the process. If you don't already have one of these email aliases set up, you should do so now before picking one and clicking Next.

You'll now be prompted to enter your administrative contact info, which it helpfully copied from your domain registration if you registered through Namecheap. Fill this stuff out, then hit Next.

You'll get taken to a web page with a handy dandy flow chart, and within a few mintues you'll get an email. Click the link in the email, copy and paste the verification code, and hit the "Next" button. You'll get another email, this one with your new certificate attached.

Installing the certificate at Heroku

At this point, you'll need to attach the SSL certificate to your application. With Heroku, this is easy.

```
$ heroku addons:add ssl:endpoint
$ heroku certs:add www.example.com.crt bundle.pem example.com.key
```

To see if the certificate installed properly:

```
$ heroku certs
```

Now just configure `www.example.com` as a CNAME pointing at the `herokussl.com` endpoint printed from `heroku certs` and test it out:

```
$ curl -kvI https://www.example.com
```

This should print out a bunch of stuff about SSL and the headers from your application.

Custom Payment Forms

Until now we've been using Stripe's excellent `checkout.js` that provides a popup iframe to collect credit card information, post it to Stripe and turn it into a `stripeToken` and then finally post our form. There's something conspicuously absent from all of this, however. Remember how Sale has an email attribute? We're not populating that right now because `checkout.js` doesn't easily let us add our own fields. For that we'll need to create our own form. Stripe still makes this easy, though, with `stripe.js`. The first half of this chapter is adapted from Stripe's custom form tutorial.

Here's the form we'll be using:

```
<%= form_tag buy_path(permalink: @product.permalink), :class => 'for
  <span class="payment-errors"></span>
  <div class="control-group">
    <label class="control-label" for="email">Email</label>
    <div class="controls">
      <input type="email" name="email" id="email" placeholder="Email
    </div>
  </div>
  <div class="control-group">
    <label class="control-label" for="number">Card Number</label>
    <div class="controls">
      <input type="text" size="20" data-stripe="number" id="number"
    </div>
  </div>

  <div class="control-group">
    <label class="control-label" for="cvc">CVC</label>
    <div class="controls">
      <input type="text" size="3" data-stripe="cvc" id="cvc" placeho
    </label>
  </div>

  <div class="form-row">
```

```
    <label class="control-label">Expiration (MM/YYYY)</label>
    <div class="controls">
      <input type="text" size="2" data-stripe="exp-month" placeholde
      <span> / </span>
      <input type="text" size="4" data-stripe="exp-year" placeholder
    </div>
  </div>

  <div class="form-row">
    <div class="controls">
      <button type="submit" class="btn btn-primary">Pay</button>
    </div>
  </div>
<% end %>
```

There's a few interesting things going on here. First, notice the almost-excessive amount of markup. I'm using Twitter Bootstrap form markup for this, which gives nice looking styling by default.

Second, take a look at the inputs. Only one of them, `email`, actually has a `name` attribute. The rest have `data-stripe` attributes. Browsers will only send inputs that have a `name` to the server, the rest get dropped on the floor. In this case, the inputs with `data-stripe` attributes will get picked up by `stripe.js` automatically and fed to Stripe's servers to be turned into a token.

To do that we need to actually send the form to Stripe. First include `stripe.js` in the page. Stripe recommends you do this in the header for compatibility with older browsers, but we're just going to stick it in the body for now. Put this at the bottom of the page:

```
<script type="text/javascript" src="https://js.stripe.com/v2/"></scr
```

Next, Stripe needs our publishable key. Remember that we have that in the Rails config due to the initializer we set up before. To set it, call `Stripe.setPublishableKey()` like this:

```
<script type="text/javascript">
```

```
$(function({
  Stripe.setPublishableKey('<%= Rails.configuration.stripe[:publisha
});
</script>
```

To intercept the form submission process, tack on a `submit` handler using jQuery:

```
$('#payment-form').submit(function(event) {
  var form = $(this);
  form.find('button').prop('disabled', true);
  Stripe.createToken(form, stripeResponseHandler);
  return false;
});
```

When the customer clicks the "Pay" button we disable the button so they can't click it again, then call `Stripe.createToken`, passing in the form and a callback function. Stripe's javascript will submit all of the inputs with a `data-stripe` attribute to their server, create a token, and call the callback function with a status and response. The implmentation of `stripeResponseHandler` is pretty straightforward:

```
function stripeResponseHandler(status, response) {
  var form = $('#payment-form');
  if (response.error) {
    form.find('.payment-errors').text(response.error.message);
    form.find('button').prop('disabled', false);
  } else {
    var token = response.id;
    form.append($('<input type="hidden" name="stripeToken">').val(to
    form.get(0).submit();
  }
}
```

If the response has an error, display the error and re-enable the "Pay" button. Otherwise, append a hidden input to the form and resubmit using the DOM method instead of the jQuery method so we don't get stuck in an infinite loop.

Embedding the Form

Custom forms are all well and good, but wouldn't it be cool if we could embed it in another page just like Stripe's Checkout? Let's give it a shot. Create a file `public/example.html` and put this in it:

```
<html>
  <head>
    <link href="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/cs
  </head>
  <body>
    <h1>Example Iframe</h1>
    <button class="btn btn-primary" id="openBtn">Buy</button>
    <div id="paymentModal" class="modal hide fade" role="dialog">
      <div class="modal-body">
        <iframe src="" style="zoom:0.6" width="99.6%" height="550" f
      </div>
    </div>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/2.0.2/jquery
    <script src="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/j
    <script type="text/javascript">
      var frameSrc = "/buy/design-for-failure"; // You'll want to cu
      $("#openBtn").click(function() {
        $("#paymentModal").on("show", function() {
          $('iframe').attr('src', frameSrc);
        });
        $("#paymentModal").modal({show: true});
      });
    </script>
  </body>
</html>
```

This page loads jQuery and Twitter Bootstrap from public CDNs and then uses them to create a Bootstrap Modal containing an `iframe`. Initially this iframe's `src` attribute is set to nothing. This is to prevent the iframe from loading on page load which could cause a lot of unnecessary traffic on the server running the sales application. When the customer clicks the button we set up the `src` attribute of the iframe and then show the modal.

This is pretty cool but also problematic. The iframe just loads the normal `/buy` action which contains the whole product description. Second, and more importantly, after the customer buys the thing they expect to be able to click on the download link and save the product, but that won't happen because we haven't set the `X-Frame-Options` header to allow the iframe to do anything. Let's fix the first problem. Move the form into a new partial named `_form.html.erb` and then call it like this in `transactions/new.html.erb`:

```
<%= render :partial => 'form' %>
```

Then, create a new action named `iframe`:

```
# in config/routes.rb

match '/iframe/:permalink' => 'transactions#iframe', via: :get, as:

# in app/controllers/transactions_controller.rb

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

In `app/views/transactions/iframe.html.rb`:

```
<h1><%= @product.name %></h1>

<p>Price: <%= formatted_price(@product.price) %></p>

<%= render :partial => 'form' %>
```

Now, change `frameSrc` to point at `/iframe/design-for-failure` and reload the page.

We can fix the other problem, with the `X-Frame-Options` header, simply by changing the language to say "Make sure to right-click and select Save As" indead of just telling the customer to click the link. In a later chapter I'll talk

about emailing and we'll be changing this some more.

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 or a timer expires, 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
      transitions from: :locked, to: :unlocked
    end
```

```
    event :use
      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_last4:string \
    card_expiration:string \
    card_type: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_last4,
    :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_last4:      charge.card.last4
        card_expiration: Date.new(charge.card.exp_year, Charge.card.
        card_type:       charge.card.type,
        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 @prod

  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 'Back', sales_path %>
```

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. There are a few different schools of thought on how to implement this. 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.

Handling Webhooks

Stripe will send your application events that they call webhooks as things happen to payments that you initiate and your users' subscriptions. The full list of event types can be found in Stripe's API documentation, but here's a brief list:

- when a charge succeeds or fails
- when a subscription is due to be renewed
- when something about a customer changes
- when a customer disputes a charge

Some of these are more important than others. For example, if you're selling one-off products you probably don't care about the events about charge successes and failures because you're initiating the charge and will know immediately how it went. Those events are more useful for subscription sites where Stripe is handling the periodic billing for you. On the other hand, you always want to know about charge disputes. Too many of those and Stripe may drop your account.

Webhook handling is going to be unique to every application. For the example app we're just going to handle disputes for now. We'll add more when we get to the chapter about subscriptions.

Validating Events

Stripe unfortunately does not sign their events. If they did we could verify that they sent them cryptographically, but because they don't the best thing to do is to take the ID from the POSTed event data and ask Stripe about it directly. Stripe also recommends that we store events and reject IDs that we've seen already to protect against replay attacks. To knock both of these requirements out at the same time, lets make a new model called Event:

```
$ rails g model Event \
    stripe_id:string \
    type:string
```

We only need to store the `stripe_id` because we'll be looking up the event using the API every time. Storing the type could be useful later on for reporting purposes.

The model should look like this:

```
class Event < ActiveRecord::Base
  validates_uniqueness_of :stripe_id

  def stripe_event
    Stripe::Event.retrieve(stripe_id)
  end
end
```

Controller

We'll need a new controller to handle callbacks:

```
# in app/controllers/events.rb

class EventsController < ApplicationController
  skip_before_filter :authenticate_user!
  before_filter :parse_and_validate_event

  def create
    render :nothing => true, :status => 200
  end

  private
  def parse_and_validate_event
    event = JSON.parse(request.body.read)
    @event = Event.new(id: event['id'], type: event['type'])
    unless event.save
      render :nothing => true, :status => 400
```

```
      return
    end
    @stripe_event = @event.stripe_event
  end
end
```

From the top, we skip Devise's `authenticate_user!` before filter because Stripe is obviously not going to have a user for our application. Then, we make our own `before_filter` that actually parses out the event and does the work of preventing replay attacks. If the event doesn't validate for some reason we return 400 and move on. If, on the other hand, it saves correctly we ask Stripe for a fresh copy of the event and then deal with it. All `#create` has to do is return 200 to tell Stripe that we successfully dealt with the event.

But we haven't actually done anything yet. # Processing Payments with Background Workers

Processing payments correctly is hard. This is one of the biggest lessons I've learned while writing my various SaaS projects. Stripe does everything they can to make it easy, with quick start guides and great documentation. One thing they really don't cover in the docs is what to do if your connection with their API fails for some reason. Processing payments inside a web request is asking for trouble, and the solution is to run them using a background job.

The Problem

Let's take Stripe's example code:

```
Stripe.api_key = ENV['STRIPE_API_KEY']

# Get the credit card details submitted by the form
token = params[:stripeToken]

# Create the charge on Stripe's servers - this will charge the user'
begin
  charge = Stripe::Charge.create(
```

```
    :amount => 1000, # amount in cents, again
    :currency => "usd",
    :card => token,
    :description => "payinguser@example.com"
  )
rescue Stripe::CardError => e
  # The card has been declined
end
```

Pretty straight-forward. Using the `stripeToken` that `stripe.js` inserted into your form, create a charge object. If this fails due to a `CardError`, you can safely assume that the customer's card got declined. Behind the scenes, `Stripe::Charge` makes an `https` call to Stripe's API. Typically, this completes almost immediately.

But what if it doesn't? The internet between your server and Stripe's could be slow or down. DNS resolution could be failing. There's a million reasons why this code could take awhile. Browsers typically have around a one minute timeout and application servers like Unicorn usually will kill the request after 30 seconds. That's a long time to keep the user waiting just to end up at an error page.

The Solution

The solution is to put the call to `Stripe::Charge.create` in a background job. By separating the work that can fail or take a long time from the web request we insulate the user from timeouts and errors while giving our app the ability to retry (if possible) or tell us something failed (if not).

There's a bunch of different background worker systems available for Rails and Ruby in general, scaling all the way from simple in-process threaded workers with no persistence to external workers persisting jobs to the database or Redis, then even further to message busses like AMQP, which are overkill for what we need to do.

In-Process

One of the best in-process workers that I've come across is called Sucker Punch. Under the hood it uses the actor model to safely use concurrent threads for work processing, but you don't really have to worry about that. It's pretty trivial to use, just include the `SuckerPunch::Worker` module into your worker class, declare a queue using that class, and chuck jobs into it.

```
# in app/workers/banana_worker.rb
class BananaWorker
  include SuckerPunch::Worker

  def perform(event)
    puts "I am a banana!"
  end
end

# in config/initializers/queues.rb
SuckerPunch.config do
  queue name: :banana_queue, worker: BananaWorker, workers: 10
end

# somewhere in a controller
SuckerPunch::Queue[:banana_queue].async.perform("hi")
```

The drawback to Sucker Punch, of course, is that if the web process falls over then your jobs evaporate. This will happen, no two ways about it. Errors and deploys will both kill the web process and erase your jobs.

Database Persistence

The classic, tried-and-true background worker is called Delayed Job. It's been around since 2008 and is battle tested and production ready. At my day job we use it to process hundreds of thousands of events every day and it's basically fire and forget. It's also easier to use than Sucker Punch. Assuming a class like this:

```
class Banana
```

```
  def initialize(size)
    @size = size
  end

  def split
    puts "I am a banana split, #{@size} size!"
  end
end
```

To queue the `#split` method in a background job, all you have to do is:

```
Banana.new('medium').delay.split
```

That is, put a call to `delay` before the call to `split`. Delayed Job will serialize the object, put it in the database, and then when a worker is ready to process the job it'll do the reverse and finally run the `split` method.

To work pending jobs, just run

```
$ bundle exec rake jobs:work
```

Delayed Job does have some drawbacks. First, because it stores jobs in the same database as everything else it has to content with everything else. For example, your database serve almost certainly has a limit on the number of connections it can handle, and every worker will require two of them, one for Delayed Job itself and another for any ActiveRecord objects. Second, it can get tricky to backup because you really don't need to be backing up the jobs table. That said, it's relatively simple and straight forward and has the distinct advantage of not making you run any new external services.

Redis

Redis bills itself as a "networked data structure server". It's a database server that provides rich data types like lists, queues, sets, and hashes, all while being extremely fast because everything is in-memory all the time. The best Redis-based background worker, in my opinion, is Sidekiq written by Mike

Perham. It uses the same actor-based concurrency library under the hood as Sucker Punch, but because it stores jobs in Redis it can also provide things like a beautiful management console and fine-grained control over jobs. The setup is essentially identical to Sucker Punch:

```
# in app/workers/banana_worker.rb
class BananaWorker
  include Sidekiq::Worker

  def perform(event)
    puts "I am a banana!"
  end
end

# somewhere in a controller
BananaWorker.perform_async("hi")
```

To work jobs, fire up Sidekiq:

```
$ bundle exec sidekiq
```

For this example we're going to use Sidekiq. If you'd like to use one of the other job systems described above, or if you already have your own for other things, it should be trivial to change.

First, let's create a job class:

```
class StripeCharger
  include Sidekiq::Worker

  def perform(event)
    ActiveRecord::Base.connection_pool.with_connection do
      token =  event[:token]
      txn = Transaction.find(event[:transaction_id])

      begin
        charge = Stripe::Charge.create(
          amount: txn.amount,
```

```
          currency: "usd",
          card: token,
          description: txn.email
        )
        txn.state = 'complete'
        txn.stripe_id = charge.id
        txn.save!
      rescue Stripe::Error => e
        txn.state = 'failed'
        txn.error = e.json_body
        txn.save!
      end
    end
  end
end
```

Again, pretty straightforward. Sidekiq will create an instance of your job class and call `#perform` on it with a hash of values that you pass in to the queue, which we'll get to in a second. We look up a `Transaction` record, initiate the charge, and capture any errors that happen along the way.

Now, in the TransactionsController, replace the transaction processing code with a call to `perform_async`, like so:

```
class TransactionsController < ApplicationController

  def create
    txn = Transaction.new(
      amount: 1000,
      email: params[:email],
      state: 'pending'
    )
    if txn.save
      StripCharger.perform_async(
        transaction_id: txn.id,
        token: params[:stripeToken]
      )
      render json: txn.to_json
    else
      render json: {error: txn.error_messages}, status: 422
```

```
    end
  end

  def show
    txn = Transaction.find(params[:id])
    raise ActionController::RoutingError.new('not found')
      unless txn

    render json: txn.to_json
  end
end
```

The `create` method creates a new `Transaction` record, setting it's state to `pending`. It then queues the transaction to be processed by `StripeCharger`. The `show` method simply looks up the transaction and spits back some JSON. On your customer-facing page you'd do something like this:

```
function doPoll(id){
    $.get('/transactions/' + id, function(data) {
        if (data.state === "complete") {
          window.location = '/thankyou';
        } elsif (data.state === "failed") {
          handleFailure(data);
        } else {
          setTimeout(function(){ doPoll(id); }, 500);
        }
    });
}
```

Your page will poll `/transactions/<id>` until the transaction ends in either success or failure. You'd probably want to show a spinner or something to the user while this is happening.

With this setup, you've insulated yourself from problems in your connection to Stripe, your connection to your customer, and everything in between.