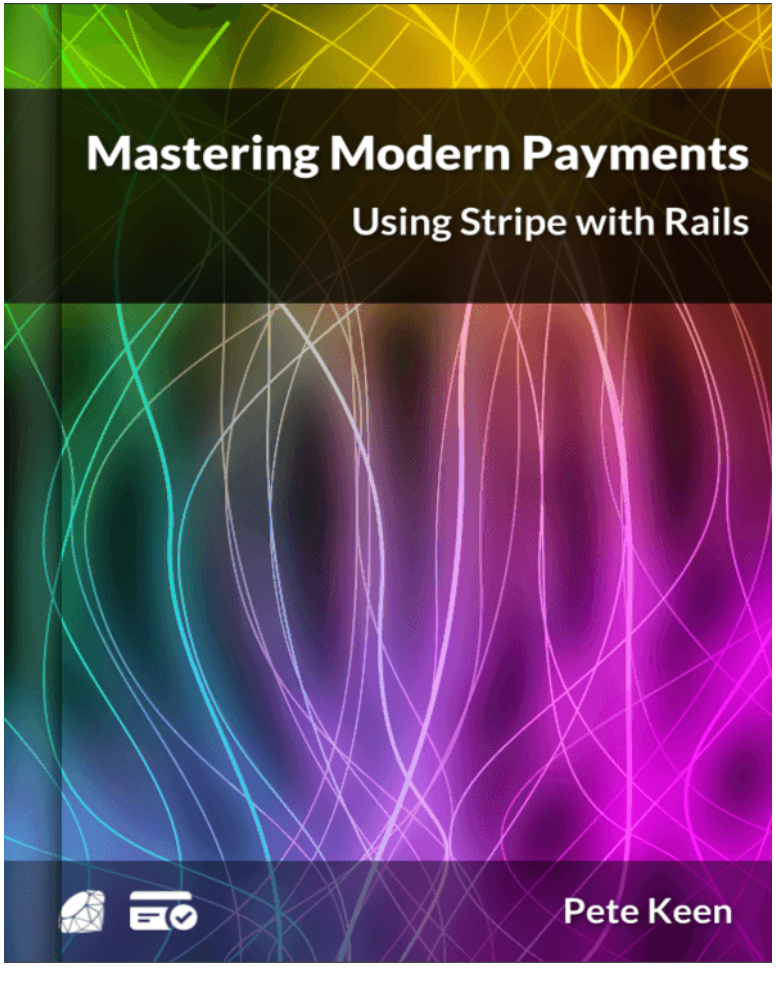

Mastering Modern Payments
Using Stripe with Rails
Pete Keen

Table of Contents

Chapter 1

Introduction

- Learn what makes Stripe great
- Read about this book
- Learn how the book is organized

Before Stripe came onto the scene, alteratives for accepting payments online were either expensive, annoying, complicated, or all three. PayPal had (and still has) a large portion of the business, but the API is very old and the payment flow is not conducive to a modern application. To accept credit cards, you either you had to use PayPal or integrate at least three separate services: a merchant account, a credit card gateway, and a recurring subscription service. You had the responsibility to make sure every piece was secure, and all of the software that integrated them had to be maintained.

Stripe makes all of this irrelevant. You can create an account on Stripe's website and be making real live charges the next day with simple, easy to use APIs and documentation. The power of this is huge: you can go from idea to accepting payments in very little time, often less than a day.

Why this guide?

If Stripe is so easy to use, why read a whole book about it? There's a few reasons. Stripe's documentation often does not go far enough. It assumes that Stripe will always be availble and responsive. It also only gives small or limited examples that don't directly apply to production applications.

This guide goes farther and deeper than Stripe's documentation. It builds up a complete production-level Rails application and covers every step along the way.

In this guide we're going to cover a basic Stripe integration and then expand upon it to cover things like background workers, subscriptions, marketplaces, audit trails, PCI compliance, and more. When you're done with the guide you should have a good grasp on how to build a complete, robust Stripe integration.

How does the guide work?

This guide builds an application lovingly named Sully after the big blue monster in the Pixar movie Monsters, Inc. Sully's job is to sell downloadable products. By the end of the book it will be a full marketplace where sellers can upload one-off and subscription content.

Each chapter has a GitHub discussion thread associated with it that you can get to by clicking on the Discuss button in the upper lefthand corner. Here, you can ask questions and give help to your fellow readers.

Who am I?

My name is Pete Keen. I've been working with the Stripe API for a little over three years now and have built many applications with it. In addition, I've worked with a wide variety of other payment systems at my full time and consulting jobs and learned quite a lot about how to handle payments in general.

Conventions

```
Code examples are marked out like this.
```

Shorter code snippets are marked `like this`.

Links are underlined and are all clickable from all of the electronic versions.

Versions

Versions of software used in the examples:

- Ruby 2.0.0-p0
- Rails 4.0.2
- Stripe Ruby API 1.15.0
- Devise 3.3.0
- Paper Trail 3.0

- PostgreSQL 9.2

See the Changelog for details about changes to the guide itself.

Chapter 2

Basic Integration

- Create a basic Rails application
- Do the simplest Stripe integration
- Learn how to deploy to Heroku

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.

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 need 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
$ cd sales
$ createuser -s sales
$ rake db:create
$ 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're using a Mac and don't have PostgreSQL installed, this is an excellent tutorial. If you want to use a different database, feel free to substitute. Any `ActiveRecord`-compatible database will do fine.

Authentication

We're going to want to be able to authenticate users who can add and manage products and view sales. 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', '~> 3.3.0'
```

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:3000'
}
```

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

```
<% flash.each do |type, msg| %>
<%= content_tag :p, msg, class: type %>
<% end %>
```

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_action :authenticate_user!
```

You'll need to create 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',  # has to be at least 8 characters
  password_confirmation: 'password'
)
```

Models

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

```
$ rails g scaffold Product \
    name:string \
    permalink:string \
    description:text \
    price:integer \
    user:references
$ rake db:migrate
```

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

```
class Product < ActiveRecord::Base
  has_attached_file :file

  belongs_to :user
end
```

Note the `has_attached_file`. We're using Paperclip to attach the downloadable files to the product record. Let's add it to `Gemfile`:

```
gem 'paperclip', '~> 4.2.0'
```

And `bundle install` again to get Paperclip installed.

Now we need to generate the migration so paperclip has a place to keep the file metadata:

```
$ rails generate paperclip product file
```

We should add an upload button to the Product edit form as well. In `app/views/products/_form.html.erb` inside the `form_for` below the other fields:

```
<div class="field">
  <%= f.label :file %><br />
  <%= f.file_field :file %>
</div>
```

We also need to populate the `user` reference. In `ProductsController#create`:

```
def create
  @product = Product.new(product_params)
  @product.user = current_user

  respond_to do |format|
    if @product.save
      format.html {
        redirect_to @product,
          notice: 'Product was successfully created.'
      }
      format.json {
        render json: @product,
          status: :created,
          location: @product
      }
    else
      format.html { render 'new' }
      format.json {
        render json: @product.errors,
          status: :unprocessable_entity
      }
    end
  end
end
```

We should also define `product_params` in `ProductsController` toward the bottom. Note that we include the `file` attribute for Paperclip:

```
private
def product_params
  params.require(:product).permit(:description, :name, :permalink, :price, :file)
end
```

We don't allow the `user_id` because we're setting it explicitly to `current_user`.

Our app needs a way to track product sales. Let's make a Sale model too.

```
$ rails g scaffold Sale \
    email:string \
    guid:string \
    product:references \
    stripe_id:string
$ rake db:migrate
```

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

```
class Sale < ActiveRecord::Base
  belongs_to :product

  before_save :populate_guid
  validates_uniqueness_of :guid

  private

  def populate_guid
    if new_record?
      while !valid? || self.guid.nil?
        self.guid = SecureRandom.random_number(1_000_000_000).to_s(36)
      end
    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. This isn't an official UUID since those tend to be long and awkward. Instead, we pick a number between 0 and one billion, turn it into a string by encoding it with base 36 (lowercase letters a-z and numbers 0-9). Then we test to make sure the record is valid. The loop will continue until there's a unique GUID value because of the `validates_uniqueness_of` on `:guid`.

We should also add the relationship to `Product`:

```
class Product < ActiveRecord::Base
  has_many :sales

  validates_numericality_of :price,
    greater_than: 49,
    message: "must be at least 50 cents"

  has_attached_file :file

  validates_attachment_content_type :file, :content_type => [
    "image/jpg",
    "image/jpeg",
    "image/png",
    "image/gif",
    "application/pdf",
    "application/zip"
  ]

end
```

Stripe does not allow charges less than $0.50, so we add a validation to make sure a product doesn't end up like that.

We're also setting up the paperclip integration on `Product`. The first line, `has_attached_file :file`, tells Paperclip to add the appropriate access methods. The second section, `validates_attachment_content_type`, ensures that only files of the specified content types get uploaded. This list includes several image types as well as PDFs and Zip files. If you're going to be selling something else, make sure to add the appropriate MIME types here.

At this point, you should be able to fire up `rails server` and create a product or two.

Deploying

Heroku is 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'll need to add one more thing, since Rails' asset pipeline doesn't play well with Heroku. Add this to `Gemfile` and run `bundle install` one more time:

```
gem 'rails_12factor', group: :production
```

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

```
$ git init
$ git add .
$ git commit -m 'Initial commit'
$ heroku create
$ git push heroku master
$ heroku run rake db:migrate
$ heroku run console # create a user
$ heroku restart web
$ heroku open
```

We'll need to set a few more config options to make our site usable on Heroku. First, we need to set up an outgoing email server and configure `ActionMailer` to use it. Let's add the Mandrill addon:

```
$ heroku addons:add mandrill:starter
```

Now configure it in `config/environments/production.rb`:

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  address:        'smtp.mandrillapp.com',
  port:           587,
  user_name:      ENV['MANDRILL_USERNAME'],
  password:       ENV['MANDRILL_APIKEY'],
  domain:         'heroku.com',
  authentication: :plain
}
config.action_mailer.default_url_options = {
  :host => 'your-app.herokuapp.com'
}
```

Mandrill is made by the same folks that make MailChimp. It's reliable, powerful, and cost effective. They give you 12,000 emails per month for free to get started. I use it for all of my applications. Note also that we configure the `default_url_options` here again for ActionMailer. This is what Devise uses to generate links inside emails, so it's pretty important to get it right.

We also need to set up Paperclip to save uploaded files to S3 instead of the local file system. On Heroku your processes live inside what they call a dyno which is just a lightweight Linux virtual machine with your application code inside. Each dyno has an ephemeral filesystem which gets erased at least once every 24 hours, thus the need to push uploads somewhere else. Paperclip makes this pretty painless. You'll need to add another gem to your Gemfile:

```
gem 'aws-sdk'
```

and then configure Paperclip to use it in `config/environments/production.rb`:

```
config.paperclip_defaults = {
  storage: :s3,
  s3_credentials: {
    bucket: ENV['AWS_BUCKET'],
    access_key_id: ENV['AWS_ACCESS_KEY_ID'],
    secret_access_key: ENV['AWS_SECRET_ACCESS_KEY']
  }
}
```

Sign up on Amazon's site to get AWS credentials if you don't already have them.

Just set those config variables with Heroku, `bundle install`, and then commit and push up to Heroku.

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 section.

The Simplest Stripe Integration

Now we're going to do a whirlwind Stripe integration, 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 three actions:

- `buy` - where we create a Sale record and actually charge the customer
- `pickup` - where the customer can download their product
- `download` which will actually send the file to the customer

In addition, we're going to leverage Stripe's excellent management interface which will show us all of our sales as they come in.

Basic Setup

First, add the Stripe gem to your Gemfile:

```
gem 'stripe', '~> 1.15.0'
```

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.

You can get your publishable and secret key from your Stripe account settings in the Dashboard.

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_action :authenticate_user!,
    only: [:new, :create]

  def new
    @product = Product.find_by!(
      permalink: params[:permalink]
    )
  end

  def pickup
    @sale = Sale.find_by!(guid: params[:guid])
    @product = @sale.product
  end

  def create
    product = Product.find_by!(
      permalink: params[:permalink]
    )

    token = params[:stripeToken]

    begin
      charge = Stripe::Charge.create(
        amount:      product.price,
        currency:    "usd",
```

```
      card:        token,
      description: params[:email]
    )
    @sale = product.sales.create!(
      email:      params[:email],
      stripe_id:  charge.id
    )
    redirect_to pickup_url(guid: @sale.guid)
  rescue Stripe::CardError => e
    # The card has been declined or
    # some other error has occurred
    @error = e
    render :new
  end
end

def download
  @sale = Sale.find_by!(guid: params[:guid])

  resp = HTTParty.get(@sale.product.file.url)

  filename = @sale.product.file.url
  send_data resp.body,
    :filename => File.basename(filename),
    :content_type => resp.headers['Content-Type']
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. Note that we hardcode `usd` as the currency. If you have a Stripe account in a different country you'll want to provide your country's currency code here.

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`. Inside the view for `#pickup` we include link to `/download` which sends the data to the user from S3.

We get the data from S3 using a gem named `HTTParty`. Let's add it to the Gemfile:

```
gem 'httparty'
```

Routes

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

```
get  '/buy/:permalink', to: 'transactions#new',      as: :show_buy
post '/buy/:permalink', to: 'transactions#create',   as: :buy
get  '/pickup/:guid',   to: 'transactions#pickup',   as: :pickup
get  '/download/:guid', to: 'transactions#download', as: :download
```

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="stripe-button"
    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`. Checkout puts a simple button on your page that pops up a small overlay onto your page where the user puts in their credit card information. Stripe automatically processes the card information into a single use token while handling errors for you. When all of that is done `checkout.js` will submit the surrounding form to your server, taking care to strip out sensitive information. It's a convenient way to collect card information if you don't want to go to the trouble of making your own custom form, which we'll talk about in a later chapter.

Notice that we just drop the description in as html which makes it a risk for cross-site-scripting attacks. Make sure you trust the users you allow to create products. We're rendering the `new` view 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 purchase by clicking the link below.
</p>

<p><%= link_to "Download", download_url(guid: @sale.guid) %></p>
```

Testing

Testing is vitally important to any modern web application, doubly so for applications involving payments. Tests are one of the best ways to make sure your app works the way you think it does.

Manually testing your application is a good first step. Stripe provides test mode keys that you can find in your account settings. By using the test mode keys you can run transactions through Stripe with testing credit card numbers and hit not only the happy case, but also a variety of failure cases. Stripe provides a variety of credit card numbers that trigger different failure modes. Here's a small selection:

- `4242 4242 4242 4242` : always succeeds
- `4000 0000 0000 0010` : address failures
- `4000 0000 0000 0101` : cvs check failure
- `4000 0000 0000 0002` : card will always be declined

There are a bunch more failure modes you can check but those are the big ones. Make sure to manually run your test through at least these failure cases. You'll catch bugs you wouldn't think to test for and you'll actually be interacting with Stripe's API, which you won't be in your automated tests.

Automated Tests

Manual testing is all well and good but you should also write repeatable unit and functional tests that you can run as part of your deploy process. This can get a little tricky, though, because you don't really want to be hitting Stripe's API servers with your test requests. They'll be slower and you'll pollute your testing environment with junk data.

Instead, let's use mocks and factories. In `Gemfile`:

```
group :development do
  gem 'stripe_mock'
  gem 'database_cleaner'
end
```

StripeMock provides mocks for the entire Stripe API so your tests don't have to actually hit Stripe's servers. Database Cleaner cleans out the database between test runs.

Let's set all of this up. In `test/test_helper.rb`:

```
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../../config/environment', __FILE__)
require 'rails/test_help'
require 'database_cleaner'

class ActiveSupport::TestCase

  setup do
    DatabaseCleaner.start
    StripeMock.start
  end

  teardown do
    DatabaseCleaner.clean
    StripeMock.stop
  end
end
```

Note that Mocha must be required as the very last thing in `test_helper`.

Let's write a test for `TransactionsController`. In `test/functional/transactions_controller_test.rb`:

```
class TransactionsControllerTest < ActionController::TestCase
  test "should post create" do
    product = Product.create(
      permalink: 'test_product',
      price:     100
    )

    post :create, email: email, stripeToken: token

    assert_not_nil assigns(:sale)
    assert_not_nil assigns(:sale).stripe_id
    assert_equal product.id, assigns(:sale).product_id
    assert_equal email, assigns(:sale).email
  end
end
```

This is a straight forward controller test. First, we create a `Product` instance, then we `POST` at the `:create` action, which will create an instance of `Sale`, setting the appropriate attributes. The important part here is that `stripe_id` is populated, which means `Stripe::Mock` is doing it's job by mocking out all of the Stripe API calls.

Deploy

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

```
$ heroku config:add \
    STRIPE_PUBLISHABLE_KEY=pk_test_publishable_key \
    STRIPE_SECRET_KEY=sk_test_secret_key
$ 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 take a detour and talk about PCI and what you have to do to be compliant and secure while using Stripe and Rails.

Chapter 3

Security and PCI Compliance

- Learn about PCI compliance
- Generate and install an SSL certificate
- Set up Rails security tools

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

In 2004 some of the big credit card processing companies, including Mastercard, Visa, and Discover, all started putting together security standards that their merchants would have to agree to abide by if they wanted to charge cards. This included things like what information from a card you can and can't store and what types of security your systems would have to have.

By 2006 the whole industry had joined up and published version 1 of the Payment Card Industry Data Security Standards (PCI-DSS). Every credit card processor put language in their

merchant agreement that bound each merchant to abiding by PCI-DSS. Security breaches from things that PCI-DSS prevents lead to audits and possibly dropping your account and getting put on an industry-wide blacklist.

Developer's Guide

One of the best resources that I've found that talks about all of these requirements is Ken Cochrane's Developers Guide to PCI Compliant Web Applications. He goes into quite a bit of depth on the various rules, regulations, and mitigation strategies that are out there. Most of the advice is platform-agnostic but some is Django-specific. All of it is great.

Stripe and PCI

The really revolutionary part of how Stripe works is in how they reduce your compliance scope as a merchant. Let's walk through a transaction on a page that doesn't use Stripe.

1. Enter your card information into a normal HTML.
2. This form POSTs to the merchant's server
3. The merchant's software passes the credit card information to their gateway service
4. The gateway service talks to all of the various banks involved.
5. When a transaction is approved by the bank, the gateway eventually deposits the money into the merchant account.

There's quite a few attack surfaces here. By far the most common was for an attacker to exploit a bug in the merchant's application and get into their database. Once they're in the database, an attacker would have free-range to copy credit card information.

Another attack vector would be to listen in on an unencrypted wifi network and wait for someone to put their card information into a non-encrypted web page, which was also common before PCI-DSS came onto the scene.

Stripe makes both of these attacks, along with many more, irrelevant with their tokenization process. When you create a form using `stripe.js` or Stripe Checkout loaded from Stripe's servers none of your customer's credit card info is sent through your servers. Instead, Stripe's JavaScript sends that info to their servers over HTTPS where they turn it into a single-use token. This token is injected into your form and sent to your server which can use it to refer to a customer's credit card without ever having seen it.

The only thing you as a merchant have to do to be PCI compliant according to Stripe is to make sure you're serving up your payment-related pages over HTTPS and ensure they use `stripe.js` or Stripe Checkout. We've already talked about Checkout and we'll cover `stripe.js` in the chapter on Custom Forms. Let's talk about setting up HTTPS.

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 HTTPS automatically on production and sets the `Strict-Transport-Security` header which tells the customer's browser to always use HTTPS. It also ensures that all cookies get the `secure` flag.

For this example `force_ssl` is all we need to do because Heroku provides what's called a "wildcard ssl certificate" for all apps accessed at `herokuapp.com`. The disadvantage of using Heroku's free certificate is that you're constrained to using `yourapp.herokuapp.com`.

If you want to use your own URL you'll need to get your own certificate (generally around $10 per year) and install it with Heroku which will cost $20 per month.

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. 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

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. You can also create what's called a wildcard certificate which will secure all of the subdomains for a given parent domain by setting `*.example.com` for the `Common Name`. This is much more expensive than single domain certificates, of course.

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 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.

Another option is to go through Extended Verification (EV), which involves quite a bit more paperwork, time, and money. The benefit is that your customers will see a green bar in Firefox and Safari with your company name instead of the URL. This increases customer confidence that you are who you say you are.

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 minutes 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

Now that you have your bright shiny new certificate you'll need to attach it 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 by `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. If things didn't work properly it'll give you errors and hints on how to fix them.

Rails Security Tools

There are a few other tools you can use help ensure that your Rails application is secure, above and beyond requiring HTTPS. Adhering to Rails best practices and making sure you don't accidentally introduce known attack vectors into your code are two of the lowest-effort things you can do to make your app secure.

Rails Best Practices

Rails Best Practices is a website where people can submit and upvote various practices that help to keep your app safe and secure, and help structure your code in a maintainable way. Conveniently, Rails Best Practices also publishes a gem that automatically checks your app against more than fourty of the most common best practices. To install it, add it to the `Gemfile`:

```
group :development do
  gem 'rails_best_practices'
end
```

You should add a rake task to simplfy running it. In `lib/tasks/security.rake`:

```
task :rails_best_practices do
  path = File.expand_path("../../../", __FILE__)
  sh "rails_best_practices #{path}"
end
```

The Rails scaffolding tends to produce code that doesn't adhere to these practices. Most notably it uses instance variables inside partials and it generates verbose render statements inside forms. The fix is pretty easy. Change this:

```
<%= form_for @object do |f| %>
  <%= f.text_input :attribute %>
<% end %>
```

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

To this:

```
<%= form_for object do |f| %>
  <%= f.text_input :attribute %>
<% end %>
```

```
<%= render 'form', object: @object %>
```

In more recent versions of Rails `render` is a lot smarter than it used to be. It knows based on context what we mean by the second argument. We also don't have to specify the `locals:` key anymore, it's just implied that the second argument is the locals hash when rendering a partial.

Brakeman

Brakeman is a security static analysis scanner for Rails applications. It goes through your code looking for known security vulnerabilities and suggests fixes. The default Rails application, in fact, ships with one of these vulnerabilities: Rails generates a "secret key base" that it uses to encrypt session information and sign cookies so users can't modify it. By default, it sticks this token into `config/initializers/secret_token.rb` as plain text. This is a vulnerability because if, for example, you release your application as open source anyone can find the token and decrypt your sessions and sign their own cookies and generally cause havok. One way to mitigate this vulnerability is to put the secret key base in an environment variable. In `config/initializers/secret_token.rb`:

```
Sales::Application.config.secret_key_base = ENV['SECRET_KEY_BASE']
```

```
$ heroku config:add SECRET_KEY_BASE=some-long-random-string
```

Installing and running `brakeman` is similar to `rails_best_practices`. Just install it and invoke it from the root of your project to start a scan. In `Gemfile`:

```
gem 'brakeman'
```

You can create a rake task to run `brakeman` too. In `lib/tasks/security.rake`:

```
task :brakeman do
  sh "brakeman -q -z"
end
```

The `-q` option tells `brakeman` to be suppress informational and `-z` tells it to treat warnings as errors.

Running Security Scanners on Deploy

I have found it helpful to create task named `check` which runs tests, Brakeman, and Rails Best Practices all at the same time:

```
task :check do
  Rake::Task['test'].invoke # could also be spec if you're using rspec
  Rake::Task['brakeman'].invoke
  Rake::Task['rails_best_practices'].invoke
end
```

In my projects I usually take this one step even farther and create a task named `deploy` which

runs the `check` task before deploying the project. For the application that sells this book I have this task:

```
task :deploy do
  Rake::Task['check'].invoke
  sh "git push origin heroku"
end
```

This checks the code using the test suite and the two scanners and then pushes it to heroku. You should have a task like this and always use it to deploy. That way you know you always have correct code running on the server.

Code Climate

A service named Code Climate wraps both both Rails Best Practices and Brakeman up into an automated service that hooks into your GitHub account. Every time you push, Code Climate will pick up the change and analyze it for known bugs, security issues, and code cleanliness problems, and then email you a report. I highly recommend it if you are working with more than just yourself and if you're using GitHub hosting. It's well worth the price.

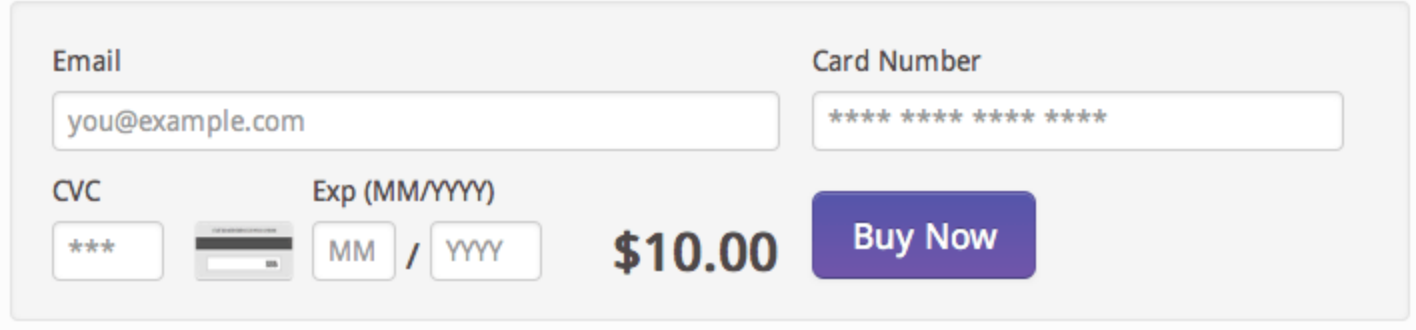
Chapter 4

Custom Payment Forms

- Build a custom payment form using stripe.js
- Learn how to embed forms in iframes

In the simple Stripe introduction we used 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. Sometimes Checkout doesn't really do what you need, though. For example, let's say you're building a subscription site and you need to include a package selection dropdown or a password field. Checkout doesn't let you add your own fields, which means you're left building your own form. In this example we're going to build a form that just duplicates what Checkout gives us, but the principles hold for any kind of form you build.

Here's the form we'll be using:



And here's the code:

```
<!-- css links should go in the header of the document, this is just here for completeness -->
<link href="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css" rel=
"stylesheet">
<!-- end header content -->

<div class="well" style="margin-left: 0px; position: relative; min-width: 650px; min-height: 180px;
max-height: 180px">
  <%= form_tag buy_path(permalink: permalink), :class => '', :id => 'payment-form' do %>
    <div class="row">
    </div>
    <div class="row">
      <div style="position: absolute; left: 20px">
        <label class="control-label" for="email">Email</label>
        <input type="email" name="email" id="email" placeholder="you@example.com" style="width: 25em"/>
```

```
      </div>
      <div style="position: absolute; left: 400px">
        <label class="control-label" for="number">Card Number</label>
        <input type="text" size="20" data-stripe="number" id="number" placeholder="**** **** **** ****"
pattern="[\d ]*" style="width: 18em"/>
      </div>
    </div>
    <div class="row" style="margin-top: 65px">
      <div style="position: absolute; left: 20px">
        <label class="control-label" for="cvc">CVC</label>
        <input type="text" style="width: 3em" size="3" data-stripe="cvc" id="cvc" placeholder="***"
pattern="\d*"/>
        <img id="card-image" src="/img/credit.png" style="height: 30px; padding-left: 10px; margin-top:
-10px">
      </div>
      <div style="position: absolute; left: 150px">
        <label class="control-label">Exp (MM/YYYY)</label>
        <input style="width: 2em" type="text" size="2" id="exp-month" data-stripe="exp-month"
placeholder="MM" pattern="\d*"/>
        <span> / </span>
        <input style="width: 3em" type="text" size="4" id="exp-year" data-stripe="exp-year" placeholder=
"YYYY" pattern="\d*"/>
      </div>
    </div>
    <div class="row" style="margin-top: 70px">
      <div class="price" style="position: absolute; left: 20px;"><%= price %></div>
```

```
      <div style="position: absolute; left: 400px">
        <button type="submit" class="btn btn-primary btn-large">Buy Now</button>
        <img style="display: none;" src="/img/well_spinner.gif" id="spinner">
      </div>
    </div>
  <% end %>
</div>
```

(You can find the file `credit.png` in a free download from Shopify.)

There's a few interesting things going on here. First, notice the almost-excessive amount of markup. We're using Twitter Bootstrap form markup for this, which gives nice looking styling for the form elements but requires a bunch of layout markup.

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/"></script>
```

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

```
<script type="text/javascript">
$(function(){
  Stripe.setPublishableKey('<%= Rails.configuration.stripe[:publishable_key] %>');
});
</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 it so they can't click 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 implementation 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(token));
    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/css/bootstrap-combined.min.css" rel=
"stylesheet">
  </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" frameborder="0"></iframe>
      </div>
    </div>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/2.0.2/jquery.min.js"></script>
    <script src="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/js/bootstrap.min.js"></script>
    <script type="text/javascript">
      var frameSrc = "/buy/design-for-failure"; // You'll want to customize this.
      $("#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.

The iframe just loads the normal `/buy` action which contains the whole product description. 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 Rails adds a header named `X-Frame-Options` that disallows any part of the site from being framed. 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 'form', permalink: @product.permalink, sale: @sale, price: formatted_price(@product.price) %>
```

Then, create a new action named `iframe`, first in `config/routes.rb`:

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

In `app/controllers/transactions_controller.rb`:

```
before_filter :strip_iframe_protection

def iframe
  @product = Product.find_by!(permalink: params[:permalink])
  @sale = Sale.new(product_id: @product)
end

private
def strip_iframe_protection
  response.headers.delete('X-Frame-Options')
end
```

The `strip_iframe_protection` removes the `X-Frame-Options` header. Without that, all modern browsers will refuse to display the form within an iframe. We remove it from all actions in `TransactionsController` so that the redirect after `POST /buy` works properly.

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

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

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

<%= render 'form', permalink: @product.permalink, sale: @sale, price: formatted_price(@product.price) %>
```

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

Next

We have an application where you can upload products and sell them to customers who can then download them. In this chapter we created a custom payment form. In the next chapter we're going to talk about keeping an audit trail using state machines and a gem named Papertrail.

Chapter 5

State and History

- Learn how to structure processes as state machines
- Add an automatic audit trail to ActiveRecord models

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" and 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 can model this turnstile in Ruby using a gem named 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
```

As you can see AASM implements a simple DSL for states and events. It 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 by adding a column to hold the current state. Let's begin by adding some more fields to `Sale`:

```
$ rails g migration AddFieldsToSale \
    state:string \
    stripe_id:string \
    stripe_token:string \
    card_expiration:date \
    error:text \
    fee_amount:integer \
    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
  include AASM

  aasm column: 'state' 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,
      )
      balance = Stripe::BalanceTransaction.retrieve(charge.balance_transaction)
      self.update(
        stripe_id:       charge.id,
        card_expiration: Date.new(charge.card.exp_year, charge.card.exp_month, 1),
        fee_amount:      balance.fee
      )
      self.finish!
    rescue Stripe::StripeError => e
      self.update_attributes(error: e.message)
      self.fail!
    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 initially. 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`.

We moved the stuff about charging the card into the model which adheres to the Fat Model Skinny Controller principle, where all of the logic lives in the model and the controller just drives it. `TransactionsController#create` is quite a bit simpler now:

```
def create
  @product = Product.find_by!(
    permalink: params[:permalink]
  )

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

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 it 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

A company I used to 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. Another, easier way, is to hook into your ORM to keep track of things. The easiest way to do this 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', '~> 3.0'
```

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 its 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>
```

And here's what it looks like:

History

Timestamp	Event	Changes
2013-07-06 21:17:12 UTC	create	**amount**: to 900 **coupon_id**: to 6 **created_at**: to 2013-07-06 21:17:12 UTC **email**: to pete@bugsplat.info **guid**: to 519f40e9-435d-4d68-b9b6-3796e92aa0f2 **id**: to 50 **product_id**: to 130 **state**: to pending **stripe_token**: to tok_29LkF8JjuRDzAM **updated_at**: to 2013-07-06 21:17:12 UTC
2013-07-06 21:17:12 UTC	update	**state**: pending to processing **updated_at**: 2013-07-06 21:17:12 UTC to 2013-07-06 21:17:12 UTC
2013-07-06 21:17:13 UTC	update	**card_expiration**: to 2014-04-01 **card_last4**: to 4242 **card_type**: to Visa **fee_amount**: to 56 **stripe_id**: to ch_29LkKF2ZiRJL9u **updated_at**: 2013-07-06 21:17:12 UTC to 2013-07-06 21:17:13 UTC

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 we'll see three rows: record creation, the state change from "pending" to "processing" when the background worker picks the job up, and another row when the background worker updates the Stripe information. The current state table will show the record as "finished". 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.

Next

In the next chapter we're going to talk about how to handle Stripe's events system, which will call a webhook in your app whenever interesting things happen with your charges, customers, or subscriptions.

Chapter 6

Handling Webhooks

- Handling webhooks with Stripe Event
- Testing your event handling
- Learn how to send PDF receipts

Stripe tracks every event that happens to the payments, invoices, subscriptions, plans, and recipients that belong to your account. Every time something happens they create an Event object and save it to their database. If you'd like you can iterate over all of these events using the API, but a much more efficient way to deal with them is to register a webhook endpoint with Stripe. Whenever they create a new event, Stripe will POST the information to all of your registered webhooks. Depending on how you respond they may retry later as well. 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.

We're going to use the StripeEvent gem to listen for webhooks. It provides an easy to use interface for handling events from Stripe in any way you choose.

Setup

The first thing to do is to add `stripe_event` to your `Gemfile`:

```
gem 'stripe_event'
```

Then, run `bundle install`.

`StripeEvent` acts as a Rails engine, which means you get everything it offers just by mounting it in your routes. Add this to `config/routes.rb`:

```
mount StripeEvent::Engine => '/stripe-events'
```

In Stripe's management interface you should add a webhook with the address `https://your-app.example.com/stripe-events`.

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, let's make a new model called StripeWebhook:

```
$ rails g model StripeWebhook \
    stripe_id:string
```

The model should look like this:

```
class StripeWebhook < ActiveRecord::Base
  validates_uniqueness_of :stripe_id
end
```

Notice that we've set up a simple uniqueness validator on `stripe_id`.

When a webhook event comes in `StripeEvent` will ignore everything except the ID that comes from Stripe using what it calls an "event retriever". To actually deduplicate events let's set up a custom event retriever in `config/initializers/stripe_event.rb`:

```
StripeEvent.event_retriever = lambda do |params|
  return nil if StripeWebhook.exists?(stripe_id: params[:id])
  StripeWebhook.create!(stripe_id: params[:id])
  StripeEvent.retrieve(params[:id])
end
```

Returning `nil` from your event retriever tells `StripeEvent` to ignore this particular event. You could use this to do other things. For example, if you are using Stripe Connect and you want to ignore events from certain users you would put that logic here.

Handling Events

The first thing we should do is handle a dispute which fires when a customer initiates a chargeback. In response to a dispute we send an email to ourselves with all of the details which should be enough to deal with them, since they should be fairly rare. In `config/initializers/stripe_event.rb`:

```
StripeEvent.configure do |events|
  events.subscribe 'charge.dispute.created' do |event|
    StripeMailer.admin_dispute_created(event.data.object).deliver
  end
end
```

In `app/mailers/stripe_mailer.rb`:

```
class StripeMailer < ActionMailer::Base
  default from: 'you@example.com'

  def admin_dispute_created(charge)
    @charge = charge
    @sale = Sale.find_by(stripe_id: @charge.id)
    if @sale
      mail(to: 'you@example.com', subject: "Dispute created on charge #{@sale.guid} (#{charge.id})").
deliver
    end
  end
end
```

And in `app/views/stripe_mailer/admin_dispute_created.html.erb`:

```
<html>
  <body>
    <p>Dispute opened on <%= link_to "charge #{@sale.guid}", sale_url(@sale) %></p>
  </body>
</html>
```

Disputes are sad. We should also handle a happy event, like someone buying something. Let's do `charge.succeeded`:

```ruby
StripeEvent.configure do |events|
  # ...

  events.subscribe 'charge.succeeded' do |event|
    charge = event.data.object
    StripeMailer.receipt(charge).deliver
    StripeMailer.admin_charge_succeeded(charge).deliver
  end
end
```

```ruby
class StripeMailer < ActionMailer::Base
  # ...

  def admin_charge_succeeded(charge)
    @charge = charge
    mail(to: 'you@example.com', subject: 'Woo! Charge Succeeded!')
  end

  def receipt(charge)
    @charge = charge
    @sale = Sale.find_by!(stripe_id: @charge.id)
    mail(to: @sale.email, subject: "Thanks for purchasing #{@sale.product.name}")
  end
end
```

In `app/views/admin_charge_succeeded.html.erb`:

```
<html>
  <body>
    <p>Charge succeeded! Amount: <%= @sale.amount %> </p>
    <p><%= link_to @sale.guid, sale_url(@sale) %></p>
  </body>
</html>
```

In response to a charge succeeding we send a receipt to the customer and an alert to ourselves so we can get that sweet dopamine hit when the email alert sound dings. We'll show the body of the receipt email below.

Many of the events that Stripe sends are for dealing with subscriptions. For example, Stripe will let you know when they're about to initiate a periodic charge and give you the opportunity to add extra things to the invoice, like monthly add-ons or overage billing. We'll talk more about this in the chapter on Subscriptions.

Testing Events

Stripe helpfully provides for test-mode webhooks. Assuming you have a publicly accessible staging version of your application, you can set up webhooks to fire when you make test mode transactions. Testing webhooks automatically is pretty simple with `StripeMock`. Let's create a new test in `test/integration/webhooks_test.rb`:

```
class WebhooksTest < ActionDispatch::IntegrationTest
  test 'charge created' do
    event = StripeMock.mock_webhook_event('charge.succeeded', id: 'abc123')

    product = Product.create(price: 100, name: 'foo')
    sale = Sale.create(stripe_id: 'abc123', amount: 100, email: 'foo@bar.com', product: product)

    post '/stripe-events', id: event.id
    assert_equal "200", response.code

    assert_equal 2, StripeMailer.deliveries.length

    assert_equal 'abc123', StripeWebhook.last.stripe_id
  end
end
```

Effective Emailing

Customers expect to be emailed when things happen with their account, and especially when you're charging them money. It's critical that you send them a few basic transactional emails and Stripe's events make it really easy.

Events to care about

For a simple app that just sells downloadable things, there aren't that many events that you really need to care about. Your relationship with the customer, as far as their credit card is concerned, is a one time thing. Be sure to send them a receipt when the transaction goes through. Note that Stripe has built in receipts but if you want to modify the content, layout, or attachments, you'll need to do it yourself. Disputes are about the only thing that can cause you pain and we've already dealt with them above.

Subscription businesses, on the other hand, get a rich variety of events from Stripe. For example, in the chapter on Subscriptions we're going to talk about how to use the Invoice events to handle Utility-style billing. One helpful hint: if you use Stripe's subscription trial periods you should ignore the first charge event, since it will be for zero dollars.

How to generate PDF Receipts

Customers, especially business customers, appreciate getting a PDF receipt along with the email. You make their lives measurably easier by including a file that they can just attach to their expense report, rather than having to go through a convoluted dance to convert your email into something they can use.

There is a paid product named PrinceXML that makes excellent PDFs but it is very expensive and not very usable on cloud platforms like Heroku. DocRaptor is a paid service that has licensed PrinceXML and provides a nice API. There's also a nice gem named Prawn that lets you generate PDFs without going through an HTML intermediary. However, the easiest and cheapest way to generate PDFs that I know of is to use an open-source service that I created

named Docverter. All you have to do is generate some HTML and pass it to Docverter's API which then returns a PDF:

In `Gemfile`:

```
gem 'docverter'
```

In `app/mailers/receipt_mailer.rb`:

```
class ReceiptMailer < ActionMailer::Base
  def receipt(sale)
    @sale = sale
    html = render_to_string('receipt_mailer/receipt.html')

    pdf = Docverter::Conversion.run do |c|
      c.from = 'html'
      c.to = 'pdf'
      c.content = html
    end

    attachment['receipt.pdf'] = pdf
    mail(to: sale.email_address, subject: 'Receipt for your purchase')
  end
end
```

In `app/views/receipt_mailer/receipt.html.erb`:

```
<html>
  <body>
    <h1>Receipt</h1>
    <p>
      You purchased <%= @sale.product.name %> for <%= formatted_price(@sale.amount) %> on <%= @sale.
created_at %>.
    </p>
    <p>
      <%= link_to "Click here", pickup_url(guid: @sale.guid) %> to download your purchase.
    </p>
    <p>
      Thank you for your purchase!
    </p>
    <p>
      -- Pete
    </p>
  </body>
</html>
```

This will send a PDF copy of the customer's receipt along with the email which they should be able to drop directly into their expense reporting sytem.

Chapter 7

Background Workers

- Learn why processing payments in the background is necessary
- Survey the landscape of background processing systems
- Make charges in the background

Stripe does everything in their power to make sure the payment process goes smoothly for you and your customers, but sometimes things out of everyone's control can go wrong. This chapter is about making sure that your payment system keeps going in the face of things like connection failures and large bursts of traffic to your application.

The Problem

Let's take Stripe's example code:

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

token = params[:stripeToken]

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
```

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.

Sometimes it takes a long time, for one of a million different reasons For example, the connection between your server and Stripe's could be slow or down. DNS resolution could be failing. Stripe's backend could be returning errors or just not returning at all.

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). The customer will submit the form using AJAX and poll while the background job contacts Stripe and handles the payment details.

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. 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
```

Then, in a controller somewhere:

```
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 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 contend with everything else. For example, your database server 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.

Another PostgreSQL-specific database backed worker system is Queue Classic, which leverages some PostgreSQL-specific features to deliver jobs to workers very efficiently. Specifically it uses `listen` and `notify`, the built-in publish/subscribe system, to tell workers when there are jobs to be done so they don't have to poll. It also uses row-level locking to reduce database load and ensure only one worker is working on a job at any given time.

Redis Persistence

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:

```
class BananaWorker
  include Sidekiq::Worker

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

Then in a controller:

```
BananaWorker.perform_async("hi")
```

To work jobs, fire up Sidekiq:

```
$ bundle exec sidekiq
```

Handling Payments

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 adapt the following.

First, let's create a job class:

```
class StripeCharger
  include Sidekiq::Worker

  def perform(guid)
    ActiveRecord::Base.connection_pool.with_connection do
      sale = Sale.find_by(guid: guid)
      return unless sale
      sale.process!
    end
  end
end
```

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 the relevant `Sale`

record and tell it to process using the state machine event we set up earlier using `AASM`.

Now, in the TransactionsController, all we have to do is create the `Sale` record and queue the job:

```
class TransactionsController < ApplicationController

  def create
    product = Product.find_by!(
      permalink: params[:permalink]
    )

    token = params[:stripeToken]

    sale = Sale.new do |s|
      s.amount = product.price,
      s.product_id = product.id,
      s.stripe_token = token,
      s.email = params[:email]
    end

    if sale.save
      StripeCharger.perform_async(sale.guid)
      render json: { guid: sale.guid }
    else
      errors = sale.errors.full_messages
```

```
      render json: {
        error: errors.join(" ")
      }, status: 400
    end
  end

  def status
    sale = Sale.find_by!(guid: params[:guid])

    render json: { status: sale.state }
  end
end
```

The `create` method creates a new `Sale` record and then queues the transaction to be processed by `StripeCharger`. Note that you may be tempted to do this in an `after_create` hook, but don't do that. `after_create` will run before the record is committed to the database, so if Sidekiq is warmed up it will start trying to process jobs before they're ready. By explicitly queuing the job you'll save yourself a headache. Another alternative is to queue the job in an `after_commit` hook but testing this gets weird because things are never actually ever committed in an rspec test.

The `status` method simply looks up the transaction and spits back some JSON. To actually process the form we have something like this, which includes the call to `stripe.js`:

```
$(function() {
```

```
// Capture the submit event, call Stripe, and start a spinner
$('#payment-form').submit(function(event) {
  var form = $(this);
  form.find('button').prop('disabled', true);
  Stripe.createToken(form, stripeResponseHandler);
  $('#spinner').show();
  return false;
});

// Handle the async response from Stripe. On success,
// POST the form data to the create action and start
// polling for completion. On error, display the error
// to the customer.
function stripeResponseHandler(status, response) {
  var form = $('#payment-form');
  if (response.error) {
    showError(response.error.message);
  } else {
    var token = response.id;
    form.append($('<input type="hidden" name="stripeToken">').val(token));
    $.ajax({
      type: "POST",
      url: "/buy/<%= permalink %>",
      data: $('#payment-form').serialize(),
      success: function(data) { console.log(data); poll(data.guid, 30) },
      error: function(data) { console.log(data); showError(data.responseJSON.error) }
```

```
      });
    }
  }

  // Recursively poll for completion.
  function poll(guid, num_retries_left) {
    if (num_retries_left == 0) {
      showError("This seems to be taking too long. Email help@example.com and reference transaction " +
guid + " and we'll take a look.");
      return;
    }
    $.get("/status/" + guid, function(data) {
      if (data.status === "finished") {
        window.location = "/pickup/" + guid;
      } else if (data.status === "error") {
        showError(data.error)
      } else {
        setTimeout(function() { poll(guid, num_retries_left - 1) }, 500);
      }
    });
  }

  function showError(error) {
    var form = $('#payment-form');
    form.find('#payment-errors').text(error);
    form.find('#payment-errors').show();
```

```
    form.find('button').prop('disabled', false);
    form.find('#spinner').hide();
  }
});
```

Putting the call to `Stripe::Charge` in a background job and having the client poll eliminates a whole class of problems related to network failures and insulates you from problems in Stripe's backend. If charges don't go through we just report that back to the user and if the job fails for some other reason Sidekiq will retry until it succeeds.

Next

Running payments through a background worker makes it easy to scale your application as well as insulates you from failures or slowdowns at any point between your customer and Stripe's servers. In the next chapter we're going to talk about Stripe's subscription features and how to make the most out of them.

Chapter 8

Subscriptions

- Learn how to set up basic subscriptions
- Create composable service objects
- Advanced subscriptions techniques

So far in this book we've talked about how to sell products once. A customer selects a product, puts their information in, and hits the "buy now" button, and that's end of our interaction with them.

The majority of SaaS businesses don't operate that way. Most of them will want their customers to pay on a regular schedule. Stripe has built-in support for these kinds of subscription payments that is easy to work with and convenient to set up. In this chapter, we're going to go through a basic integration, following the same priciples that we've laid out for the one-time product sales. Then, we'll explore some more advanced topics, including how to effectively use the subscription webhook events, in-depth coverage of Stripe's plans, and options for reporting.

Basic Integration

For this example, we're going to add a simple subscription system where people can sign up to receive periodic download links, like magazine articles.

Let's start by making a few models. We'll need models to keep track of our pricing plans and each user's subscriptions, since they may sign up for one or more magazine.

```
$ rails g model plan \
    stripe_id:string \
    name:string \
    description:text \
    amount:integer \
    interval:string \
    published:boolean
```

```
$ rails g model subscription \
    user:references \
    plan:references \
    stripe_id:string
```

```
$ rails g migration AddStripeCustomerIdToUser \
    stripe_customer_id:string
```

Open up the models and add audit trails:

```
class Plan < ActiveRecord::Base
  has_paper_trail
  validates :stripe_id, uniqueness: true
end
```

```
class Subscription < ActiveRecord::Base
  belongs_to :user
  belongs_to :plan

  has_paper_trail
end
```

Notice we also added a uniqueness constraint to `Plan`. Re-using Stripe plan IDs is technically allowed but it's not a very good idea.

Service Objects

In this integration we're going to be using service objects to encapsulate the business logic of creating users and subscriptions. In our usage, a service object lives in `/app/services` and contains one main class method named `call` which receives all of the dependencies that the object needs to do it's job.

Here's our `CreateUser` service, in `/app/services/create_user.rb`:

```
class CreateUser
  def self.call(email_address)

    user = User.find_by(email: email_address)

    return user if user.present?

    raw_token, enc_token = Devise.token_generator.generate(
      User, :reset_password_token)
    password = SecureRandom.hex(32)

    user = User.create!(
      email: email_address,
      password: password,
      password_confirmation: password,
      reset_password_token: enc_token,
      reset_password_sent_at: Time.now
    )

    return user, raw_token
  end
end
```

In our signup flow, we are going to have the user provide their email address at the same time they give us their credit card. Internally, Devise will set up a password reset token for us if we ask, but there's no way to get out the raw token so we can send it to the user so we have to do it ourselves. `NOTE <-- wtf`

`CreateUser.call` takes an email address and first attempts to look up the user with that email address. If there isn't one, it proceeds to generate the Devise password reset token, create the user, and then return both the user and the token.

Now that we can create a user, let's create a subscription:

```
class CreateSubscription
  def self.call(plan, email_address, token)
    user, raw_token = CreateUser.call(email_address)

    subscription = Subscription.new(
      plan: plan,
      user: user
    )

    begin
      stripe_sub = nil
      if user.stripe_customer_id.blank?
        customer = Stripe::Customer.create(
          card: token,
          email: user.email,
```

```
        plan: plan.stripe_id,
      )
      user.stripe_customer_id = customer.id
      user.save!
      stripe_sub = customer.subscriptions.first
    else
      customer = Stripe::Customer.retrieve(user.stripe_customer_id)
      stripe_sub = customer.subscriptions.create(
        plan: plan.stripe_id
      )
    end

    subscription.stripe_id = stripe_sub.id

    subscription.save!

    if subscription.errors.empty?
      UserMailer.send_receipt(user.id, plan.id, raw_token)
    end

  rescue Stripe::StripeError => e
    subscription.errors[:base] << e.message
  end

  subscription
end
```

```
end
```

One of the best things about service objects is how easy it is to compose them. We can just use the `CreateUser` service we set up to create a user wherever we want, including in other service objects.

First we create the user and then a `Subscription` object. Next, we actually talk to Stripe. All we have to do is create a `Stripe::Customer` object with the plan, token, and email address of the user. We store the customer ID onto our `Subscription` object for later reference then send a receipt email which will contain a link for the user to set up their password.

How do we get those plans, though? Let's create one more service object for creating new plans in our database and propagating them to Stripe:

```
class CreatePlan
  def self.call(options={})
    plan = Plan.new(options)

    if !plan.valid?
      return plan
    end

    begin
      Stripe::Plan.create(
        id: options[:stripe_id],
```

```
        amount: options[:amount],
        currency: 'usd',
        interval: options[:interval],
        name: options[:name],
      )
    rescue Stripe::StripeError => e
      subscription.errors[:base] << e.message
    end

    plan.save!

    return plan
  end
end
```

All this does is pass the options hash through to `Plan#new` and then attempts to create a Stripe-level plan with those same options. If everything goes well, it then saves our new plan and returns it. It's very easy to use this service object in the console so we're not going to build out a controller here. Here's an example of creating a plan from the console:

```
irb(main):001:0> CreatePlan.call(stripe_id: 'test_plan', name: 'Test Plan', amount: 500, interval:
'month', description: 'Test Plan', published: false)
```

Controller

The next thing we have to do is actually use the service objects. Thankfully, that's pretty simple:

```
class SubscriptionsController < ApplicationController
  skip_before_filter :authenticate_user!

  before_filter :load_plans

  def index
  end

  def new
    @subscription = Subscription.new
    @plan = Plan.find(params[:plan_id])
  end

  def create
    @subscription = CreateSubscription.call(
      params[:email_address],
      Plan.find(params[:plan_id]),
      params[:stripeToken]
    )
    if @subscription.errors.blank?
```

```
      flash[:notice] = 'Thank you for your purchase!' +
        'Please click the link in the email we just sent ' +
        'you to get started.'
      redirect_to :root
    else
      render :new
    end
  end

protected

  def load_plans
    @plans = Plan.where(published: true).order('price')
  end

end
```

Before we do anything else, we have to load the published plans so they're available for the actions. Other than that, this is a normal, ordinary, every day Rails controller. We use the service object we created previously to actually create the subscription, and we check that it made it all the way through the process without any errors. Let's fill out the views:

`/app/views/subscriptions/index.html.erb`:

```
<% @plans.each do |plan| %>
  <%= link_to "#{plan.name} (#{plan.price})",
        new_subscription_path(@plan) %>
<% end %>
```

`/app/views/subscriptions/new.html.erb`:

```
<% unless @subscription.errors.blank? %>
  <%= @subscription.errors.full_messages.to_sentence %>
<% end %>

<h2>Subscribing to <%= @plan.name %></h2>

<%= form_for @subscription do |f| %>
  <span class="payment-errors"></span>

  <div class="form-row">
    <label>
      <span>Email Address</span>
      <input type="email" size="20" name="email_address"/>
    </label>
  </div>

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

```
    <label>
      <span>Card Number</span>
      <input type="text" size="20" data-stripe="number"/>
    </label>
  </div>

  <div class="form-row">
    <label>
      <span>CVC</span>
      <input type="text" size="4" data-stripe="cvc"/>
    </label>
  </div>

  <div class="form-row">
    <label>
      <span>Expiration (MM/YYYY)</span>
      <input type="text" size="2" data-stripe="exp-month"/>
    </label>
    <span> / </span>
    <input type="text" size="4" data-stripe="exp-year"/>
  </div>

  <button type="submit">Pay Now</button>
<% end %>

<%= javascript_tag do %>
```

```
  Stripe.setPublishableKey('<%= Rails.configuration.stripe['publishable_key'] %>');
<% end %>
```

`/app/assets/javascripts/subscriptions.js`:

```
jQuery(function($) {
  $('#payment-form').submit(function(event) {
    var $form = $(this);

    $form.find('button').prop('disabled', true);

    Stripe.card.createToken($form, stripeResponseHandler);

    return false;
  });
});

function stripeResponseHandler(status, response) {
  var $form = $('#payment-form');

  if (response.error) {
    // Show the errors on the form
    $form.find('.payment-errors').text(response.error.message);
    $form.find('button').prop('disabled', false);
  } else {
```

```
    // response contains id and card, which contains additional card details
    var token = response.id;
    // Insert the token into the form so it gets submitted to the server
    $form.append($('<input type="hidden" name="stripeToken" />').val(token));
    // and submit
    $form.get(0).submit();
  }
};
```

With those in place, you should be able to click through and create paying users.

Multiple Subscriptions

Stripe allows a customer to have multiple subscriptions. Because of the way we've set up our `Subscription` class, this is trivial to accomplish in our application. All you have to do is call `CreateSubscription.call()`, passing in the user's email address, the plan, and a blank token, like this:

```
CreateSubscription.call(
  current_user.email_address,
  some_plan,
  ''
)
```

Upgrading and Downgrading Subscriptions

What about when a user wants to change their plan? For example, a user wants to go from the 10 frobs a month plan to one with 1000. Or maybe go the other way?

Let's wrap that up in another service object:

```
class ChangePlan
  def self.call(subscription, to_plan)
    from_plan = subscription.plan
    begin
      user = subscription.user
      customer = Stripe::Customer.retrieve(user.stripe_customer_id)
      stripe_sub = customer.subscriptions.retrieve(subscription.stripe_id)

      stripe_sub.plan = to_plan.stripe_id
      stripe_sub.save!
      subscription.plan = to_plan
      subscription.save!
    rescue Stripe::StripeError => e
      subscription.errors[:base] << e.message
    end

    subscription
  end
end
```

What if the user wants to change or update their card? Again, pretty simple. Just set up a form like above but just with the card attributes, then create another service object to handle the action:

```
class ChangeSubscriptionCard
  def self.call(subscription, token)
    begin
      user = subscription.user
      customer = Stripe::Customer.retrieve(user.stripe_customer_id)
      stripe_sub = customer.subscriptions.retrieve(subscription.stripe_id)

      stripe_sub.card = token
      stripe_sub.save!
    rescue Stripe::StripeError => e
      subscription.errors[:base] << e.message
    end

    subscription
  end
end
```

The controller actions for both of these are self-explanatory. Just grab the subscription in question and the plan or token and pass them to the appropriate service object's `call` method.

Dunning

Sometimes customers don't pay their bill, often through no fault of their own. The process of communicating with your customers to get them to pay is called "dunning" and it's vital for any type of business. For a subscription SaaS using Stripe where the customer's card is billed automatically every period the dunning process kicks in when a charge fails for some reason and we send them an email. The next month we send them another, more strongly worded email, eventually leading to cancelling their account.

Really, though, you don't want to let the process even get started. The number one reason why subscription charges start getting declined is that the customer's card expires. Since you're saving the customer's card expiration in your database (if you're not, you should start), it's a trivial matter to find all of the customers that have an expiration coming up and send them a short reminder email:

```
expiring_customers = Customer.where(
  'date_reminded is null and expiration_date <= ?',
  Date.today() + 30.days
)

expiring_customers.each do |customer|
  StripeMailer.card_expiring(customer).deliver
  customer.update_attributes(date_reminded: Date.today)
end
```

Andrew Culver is the author of [Koudoku][subscriptions-koudoku] and currently is developing a product to automate this process. He has had phenomenal success reducing churn using this method:

> In one product where this approach has been taken, the campaign stops 50% of expiring credit card accounts from having a failed payment. For the remaining 50% we have another campaign that kicks in once the payment fails. Then after a couple days an email is sent to the sales team. Before we automated this process it was a major source of pain for us to manage these accounts going delinquent. It's still a source of work for our sales team and a source of customer churn for us, but it's much smaller and more manageable overall.

Andrew's campaign sends emails at 30, 15, and three days before card expiration, as well as the day of. Make sure to describe what's going on and give them an easy way to login to your app and update their card. If the payment does eventually fail, make sure to contact them again. According to [Patrick McKenzie][subscriptions-patio11-rainy-day] you should also include a P.S. to the effect that you're a small business, not a bank, and that they're not in trouble or anything. You're sure it's a mistake so you won't be cutting them off for a few days.

Speaking of cutting them off, you really shouldn't automatically cancel anyone's account without a manual review process. Charges fail sometimes and it's nobody's fault, which is why Stripe automatically retries for you for a configurable number of days. After that's up and the charge finally fails, send yourself an email and follow up with the customer, either by email or over the phone.

There's one more aspect to dunning: following up on cancelled accounts. If a high value customer decides to cancel, give them a call and ask if there's anything you can do to change

their mind. It's worth a shot, and most of the time you can work something out.

Utility-style Usage Billing

Handling a basic subscription is straight forward and well covered in the example apps. Let's say, however, you're building an app where you want metered billing like a phone bill. You'd have a basic subscription for access and then monthly invoicing for anything else. Stripe has a feature they call [Invoices][subscriptions-stripe-invoices] that makes this easy. For example, you want to allow customers to send email to a list and base the charge it on how many emails get sent. You could do something like this:

```ruby
class EmailSend < ActiveRecord::Base
  # ...

  belongs_to :user
  after_create :add_invoice_item

  def add_invoice_item
    Stripe::InvoiceItem.create(
      customer: user.stripe_customer_id,
      amount: 1,
      currency: "usd",
      description: "email to #{address}"
    )
  end
end
```

At the end of the customer's billing cycle Stripe will tally up all of the `InvoiceItems` that you've added to the customer's bill and charge them the total plus their subscription plan's amount.

Stripe will also send you a webook detailing the customer's entire invoice right before they initiate the charge. Instead of creating an invoice item for every single email as it gets sent, you could just create one invoice item for the number of emails sent in the billing period:

```ruby
StripeEvent.configure do |events|
  events.subscribe 'invoice.created' do |event|
    invoice = event.data.object

    num_emails = EmailSend.where(
      'created_at between ? and ?',
      [Time.at(invoice.period_start), Time.at(invoice.period_end)]
    ).count
    Stripe::InvoiceItem.create(
      invoice: invoice.id,
      amount: num_emails,
      currency: 'usd',
      description: "#{num_emails} emails sent @ $0.01"
    )
  end
end
```

Note that this can get kind of complicated if invoice items can be charged at different rates. You can either add one `InvoiceItem` per indivicual charge, or you can add one `InvoiceItem` per item type with the amount set to `num_items * item amount`.

Free Trials

Stripe supports adding free trials to your subscription plans. You can either set `trial_period_days` on the plan itself, or you can set `trial_ends` to a timestamp on the customer's subscription when you create it. `trial_ends` overrides `trial_days`, which means it's trivial to give a particular customer an extra long or extra short trial.

While a free trial is ongoing, you can manipulate the `trial_ends` attribute on a subscription. You can set it to a future time to extend the trial, or you can set it to the special value "now" to force it to end immediately.

If you want to prevent users from getting muliple trials, you'll need to do the deduplication for yourself. Stripe doesn't handle it. The good news is, Devise won't let multiple accounts share an email address, so we're good to go.

Reporting

Accepting payments with Stripe is only half of the battle. The other half is making sure you know if you're getting paid properly. I advise a "trust but verify" posture. Of course Stripe is going to be better at actually triggering payments, but we should record them as they happen so we know if Stripe or our bank messes up somehow.

The easiest way to do that is to record transactions as they happen by catching Stripe's webhooks. Let's add an `InvoicePayment` model:

```
$ rails g model InvoicePayment \
    stripe_id:string,
    amount:string,
    fee_amount:string,
    user:references,
    subscription:references
```

We can populate these by adding another StripeEvent subscription:

```ruby
StripeEvent.configure do |events|
  events.subscribe('invoice.payment_succeeded') do |event|
    invoice = event.data.object
    user = User.find_by(stripe_id: invoice.customer)
    invoice_sub = invoice.items.select { |i| i.type == 'subscription' }.first.id
    subscription = Subscription.find_by(stripe_id: invoice_sub)

    charge = invoice.charge

    balance_txn = Stripe::BalanceTransaction.retrieve(charge.balance_transaction)

    InvoicePayment.create(
      stripe_id: invoice.id,
      amount: invoice.amount,
      fee_amount: balance_txn.fee,
      user_id: user.id,
      subscription_id: subscription.id
    )
  end
end
```

And now we can run queries against the `invoice_payments` table to see, for example, how much a given user has paid in the last year, or how much revenue a particular subscription plan has generated. There are a few tools that make this easier to work with:

- Groupdate makes it trivial to group by various date dimensions
- Chartkick generates wonderful charts from the data generated by Groupdate.

3rd Party Services

There is an entire ecosystem of Stripe reporting services these days. Here's a few examples:

- Baremetrics gives amazing dashboards and drill-down reports
- Hookfeed builds customer-level analytics and generates email reports
- FirstOfficer provides insightful reports that tell you why your business is behaving how it is.

All three of these are driven directly from your Stripe event feed and hook into your account via Stripe Connect. To see how to build a service like that, read on to the Marketplaces chapter.

Chapter 9

Connect and Marketplaces

- Use OAuth to connect to your customer's accounts
- Make payments to third-party bank accounts with Receipients and Transfers

Marketplaces let multiple people sell goods and services on the same site at the same time. Stripe lets your SaaS app implement a marketplace in two different ways. With Stripe Connect your users connect their Stripe account with yours, allowing you to make charges with their Stripe account securely and passing fees through directly. This is great for a technical audience or for a marketplace who's participants want to be able to manage their own Stripe account.

Stripe recently added the ability to send transfers to any authorized US checking account via ACH transfers using a feature called Payouts. This enables for more nuanced interactions with your marketplace participants. For example, you could send their collected payments to them once a month, or only if they've passed a certain threshold. Payouts also allows non-technical people to easily participate in your marketplace since they don't have to leave your site to create a Stripe account via Stripe's OAuth flow.

Both Connect and Payouts are easily integrated into a Rails application with a few minor

changes. In fact, depending on your use case you may want to hook up both and let market participants decide which is best for them.

Connect

Connect is Stripe's OAuth2-based way for your application to create transactions, customers, subscription plans, and everything else Stripe has to offer on behalf of another user. Hooking it up to your Rails application is easy because someone else has done all of the hard work for you in a gem named OmniAuth::StripeConnect. This gem uses the OmniAuth OAuth abstraction library to allow you to connect to your users' Stripe accounts by simply sending them to `/auth/stripe_connect`, which will direct them through the OAuth2 dance and bring them back to your site.

To start hooking this up, simply add the gem to your `Gemfile`:

```
gem 'omniauth-stripe-connect', '>= 2.4.0'
```

Then, add the middleware in an initializer `config/initializers/omniauth.rb`:

```
Rails.application.config.middleware.use OmniAuth::Builder do
  provider :stripe_connect, ENV['STRIPE_CONNECT_CLIENT_ID'], ENV['STRIPE_SECRET_KEY']
end
```

`STRIPE_SECRET_KEY` is the same key we set up in `config/initializers/stripe.rb` way back in the beginning. To get the value for `STRIPE_CONNECT_CLIENT_ID` you need to register an

application. This should be a simple process but I've had trouble with it before. Feel free to contact Stripe's very helpful support if you hit any snags.

When you send the user through `/auth/stripe_connect`, after registering them or logging them in Stripe will send them back to `/auth/stripe_connect/callback` (via OmniAuth). Add a route to handle this:

```
get '/auth/stripe_connect/callback', to: 'stripe_connect#create'
```

Let's set up that controller in `app/controllers/stripe_connect.rb`:

```
class StripeConnectController < ApplicationController

  def create
    auth_hash = request.env['omniauth.auth']
    current_user.stripe_id = auth_hash['uid']
    current_user.stripe_access_key = auth_hash['credentials']['token']
    current_user.stripe_publishable_key = auth_hash['info']['stripe_publishable_key']
    current_user.stripe_refresh_token = auth_hash['credentials']['refresh_token']
    current_user.save!
    flash[:notice] = "Stripe info saved"
    redirect_to '/'
  end

end
```

OmniAuth will run before your controller and populate a key in the request env named `omniauth.auth` with the OAuth2 information that Stripe returned. The salient bits are the user's `uid`, their `stripe_access_key`, `stripe_publishable_key`, and refresh token. You need to save the refresh token so you can regenerate your access key if necessary. For example, if you were vulnerable to the Heartbleed SSL attack you should have rolled your customers' access tokens.

Before any of this will work we need to add the fields to `User`:

```
$ rails g migration AddStripeConnectFieldsToUser \
  stripe_id:string \
  stripe_access_key:string \
  stripe_publishable_key:string \
  stripe_refresh_token:string
$ rake db:migrate
```

Make Charges with a User's Credentials

To actually charge cards with an authenticated user's credentials all you have to do is pass the user's access key to the create call:

```
charge = Stripe::Charge.create(
  {
    amount:          1000,
    application_fee: 100,
    currency:        'usd',
    card:            params[:stripeToken],
    description:     'customer@example.com',
  },
  user.stripe_access_key
)
```

Note also in this example that we're passing the `application_fee` option. This subtracts that

amount from the total `amount` after Stripe subtracts it's fee. So, in this example the user would get $8.41 deposited in their account seven days later:

```
Amount              1000
Stripe Fee:          59 (1000 * 0.029) + 30
Application Fee     100
-----------------------
Total Paid to User  841
```

You'll also need to use the user's publishable key instead of your own when tokeninzing cards with `stripe.js`. For a custom form that would be:

```
<script type="text/javascript">
$(function(){
  Stripe.setPublishableKey('<%= current_user.stripe_publishable_key %>');
});
</script>
```

While for a Stripe Checkout form you'd use something like this:

```
<script src="https://checkout.stripe.com/v2/checkout.js"
    class="stripe-button"
    data-key="<%= current_user.stripe_publishable_key %>"
    data-description="<%= @product.name %>"
    data-amount="<%= @product.price %>"></script>
```

Webhooks

Stripe will send webhook events to your application for any connected users. There's one gotcha here, in that Stripe will by default send both live and test mode events to your application's live end point. If you don't filter those events out you'll see a lot of errors. Here's an extension of our StripeEvent retriever from the Webhooks chapter that filters out test-mode events:

```
StripeEvent.event_retriever = lambda do |params|
  return nil if Rails.env.production? && !params[:livemode]

  return nil if StripeWebhook.exists?(stripe_id: params[:id])

  StripeWebhook.create!(stripe_id: params[:id])
  Stripe::Event.retrieve(params[:id])
end
```

This would also be an excellent place to filter out any events for users that don't have active

subscriptions. This filter would go after you retrieve the event from Stripe so you can look up the account ID in your database.

Transfers

Stripe Transfers are another, more flexible way to implement marketplaces with Stripe. Instead of connecting to a user's Stripe account and making charges through it, you get authorization to make deposits directly into their checking account. Charges run through your Stripe account and you decide when to pay out to the user. This is useful if your marketplace particpants don't care that you're using Stripe, or if signing them up for an account is more burdensome than you want to deal with. For example, one of the initial customers for Stripe Transferss was Lyft, a do-it-yourself taxi service. Drivers give Lyft their checking account info who then create `Stripe::Recipient`s. Passengers pay with their credit cards through Lyft's mobile app, which uses Stripe behind the scenes to actually run payments. Drivers never have to deal with Stripe directly, instead Lyft just pays out to their accounts periodically.

One thing to keep in mind is that once you create a `Stripe::Recipient` and turn on production transfers in Stripe's management interface your account will no longer receive automatic payouts. Instead, Stripe will hold funds in your account until you tell them where to send them.

Collect Account Info

Theoretically you could collect marketplace participants' checking account information via a normal Rails action because PCI-DSS does not consider them sensitive information. However, `stripe.js` provides the capability to tokenize the numbers the same way it tokenizes credit card numbers and you really should take advantage of it. That way sensitive information never touches your server:

```
<%= form_tag update_checking_account_path(id: @user.id), :class => 'form-horizontal', :id =>
'account-form' do %>
  <div class="control-group">
    <label class="control-label" for="fullName">Full Name</label>
    <div class="controls">
      <input type="text" name="fullName" id="fullName" />
    </div>
  </div>

  <div class="control-group">
    <label class="control-label" for="number">Routing Number</label>
    <div class="controls">
      <input type="text" size="9" class="routingNumber" id="number" placeholder="*********"/>
    </div>
  </div>

  <div class="control-group">
```

```
    <label class="control-label">Account Number</label>
    <div class="controls">
      <input type="text" class="accountNumber" />
    </div>
  </div>

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

```
$('#account-form').submit(function() {
  Stripe.bankAccount.createToken({
    country: 'US',
    routingNumber: $('.routingNumber').val(),
    accountNumber: $('.accountNumber').val(),
  }, stripeResponseHandler);
  return false;
});

function stripeResponseHandler(response) {
  var form = $('#account-form');
  form.append("<input type='hidden' name='stripeToken' value='" + response.id + "'/>")
  form.get(0).submit();
}
```

This is a simplified form of the normal Stripe card tokenizing form and works basically the same way. You call `Stripe.bankAccount.createToken` with the routing number, account number, and country of the account which we're hard coding to 'US'. `createToken` takes a callback which then appends a hidden input to the form and submits it using the DOM method instead of the jQuery method so we avoid creating loops.

On the server side, just create a `Stripe::Recipient` and save the ID to the user's record:

```
recipient = Stripe::Recipient.create(
  name: params[:fullName],
  type: 'individual',
  bank_account: params[:stripeToken]
)

current_user.update_attributes(:stripe_recipient_id => recipient.id)
```

Now, just create charges as normal while keeping track of which recipient the charges are intended for. The easiest way to do this is to attach the `recipient_id` or `user_id` to a `Sale` record.

Create Transfers

When you're ready to pay out to a recipient, either on a schedule or when the user requests it, all you have to do is create a `Stripe::Transfer`:

```
transfer = Stripe::Transfer.create(
  amount:      10000,
  currency:    'usd',
  recipient:   user.stripe_recipient_id,
  description: 'Transfer'
)
```

This will initiate a transfer of $100 into the user's registered account. If you instead use `self` for the `recipient` option it will transfer the requested amount into the account you've attached to the Stripe account. If you've configured a callback URL Stripe will send you an event when the transfer completes named `transfer.paid`. You can use this event to send the user a receipt or a notification. You'll also get an event `transfer.failed` if there was an error anywhere along the line.

Each transfer costs a fixed $0.25 which is removed from your Stripe account at the time you create the transfer. If the transfer fails Stripe charges you $1, which will again be removed from your account.

Chapter 10

Additional Resources

- More links to docs, examples, and libraries

Hopefully this guide has given you a good overview of working with Stripe with Rails and the tips and hints I've given have been helpful. What follows is a list of links to additional resources that I've found very helpful to my Stripe implementations.

Documentation

- Stripe's documentation is truly excellent. If some question isn't answered in this book that should be the first place you look.
- Stripe Ruby API Docs: The Ruby API has it's own documentation mini site
- Stripe Ruby API code: The ruby API is open source and is a pretty good read. If you're looking to build a comprehensive HTTP API in Ruby you should definitely model it on this.

Examples

- Stripe's GitHub Projects contain lots of gems, including all of their language SDKs
- Rails Stripe Membership SaaS is a comprehensive example of a Stripe membership integration.
- RailsApps is great collection of example Rails apps in general.
- Working with Stripe Payouts is a short article about integrating Stripe Payouts into your rails app
- Monospace Rails, Stripe's official subscription example

Libraries

- Koudoku, a Rails engine for subscription billing with Stripe
- StripeEvent, a Rails engine for Stripe event handling
- StripeMock, a library for mocking and testing Stripe webhooks and API integrations
- StripeTester, another library for testing Stripe webhooks
- Devise, a Rails authentication library
- CanCan, a Rails authorization library
- jQuery Payment is a very useful set of jQuery functions for working with payment fields
- Stripe Auto Paginate sets up automatic pagination for Stripe's API responses

Additional Reading

- The Tangled Web is a guide to web security and how crazy the internet really is. Not directly applicable to a Stripe/Rails integration but it is a great read and is full of useful tips and explanations. Highly recommended.
- Developer's Guide to PCI Compliant Web Applications goes into great depth about how to write a PCI compliant web application, and how Stripe makes it easier.

Chapter 11

Colophon

About the Author

Pete Keen is a software developer who has built half a dozen different Stripe and Rails integrations over the past few years. He's been professionally developing software for seven years, most recently for a web game platform named Kongregate.

Acknowledgements

Thank you to my wife Emily for putting up with this wacky project. Thanks to Nathan Barry for writing Authority and inspiring me to write in the first place. Matt Vanderpol sent in dozens of spelling and grammar corrections, content ideas, and bug reports and was overall an awesome resource. Finally, thank you to all of the people who preordered and sent me questions and comments.

Colophon

This book is typeset using Merriweather for body text, Lato for headers, and Inconsolata for code, all from Google Web Fonts. Initial production happened in Emacs using Markdown for basic formatting. Final production uses a toolchain involving Rake and Docverter, the author's open source document conversion service.