Extracted from:

Take My Money

Accepting Payments on the Web

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Take My Money

Accepting Payments on the Web

Noel Rappin

edited by Katharine Dvorak

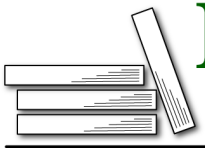
Take My Money

Accepting Payments on the Web

Noel Rappin

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Katharine Dvorak (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-199-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2017

Disclaimer: This book is intended only as an informative guide on setting up a financial transaction website. Information in this book is general, and the author, editors, and The Pragmatic Programmers, LLC disclaim all liability for compliance to federal, state, and local laws in connection with the use of this book. This work is sold with the understanding that the author, editors, and The Pragmatic Programmers, LLC do not offer legal, financial, or other professional services. If professional assistance is required, the services of a competent professional person should be sought.

Throughout this book we'll look at many different aspects of financial transactions. We'll talk about reporting, administration, legal requirements, and a host of other things. But the key part of taking money from customers is actually taking money from them.

It turns out you can send an API call with a particular set of authentication information to a particular server and this will have the real-world effect of removing money from one person's account and adding it to your account. I don't know about you, but the fact that this is possible still seems magical, even though it happens zillions of times a day all across the world.

In the previous chapter we set up our shopping cart feature. Our users are able to select tickets to a specific performance and are then taken to a shopping cart page that shows the newly entered items. In this chapter we'll use a payment gateway to walk through what you might call traditional credit card processing: users enter their credit card information into a form, then submit it to our server, and our server interacts with a payment gateway to process the payments.

The key to security in payment is to hold on to as little secure information as possible. Persisting your users' credit card data is not actually recommended unless you have either a strong need to hold onto your users' secure information or a strong desire to be hacked and find yourself on the wrong end of a lot of embarrassing headlines. Instead, we'll keep the credit card information briefly in our server's memory as we process the request. It is important to note that even this is not a recommended security practice. Under PCI compliance guidelines (see *Compliance*, on page ?), your server is just as suspect if the credit card information touches it even if the data is not persisted. We'll explore a little bit about why that might be as we go along.

The older server-side credit card processing is the easiest method to think about conceptually, so we'll start there, even though it is *not recommended* from a security standpoint. Once we've got that set up, we'll tweak the process slightly in Chapter 3, *Client-Side Payment Authentication*, on page ?, to allow for a two-step process, where the customer's information is authenticated by the client via an Ajax callback to the payment gateway. This method is even more secure, since the client's private information doesn't even reach our server. Specifically, using the client-side method places you in a much simpler situation when it comes to PCI compliance. Later, in Chapter 4, *PayPal*, on page ?, we'll talk about PayPal, the most commonly used online payment service, which has a different workflow that also keeps users' credit card information off our servers.

To be absolutely clear, the server-side setup in this chapter is just to make it a easier to talk about the Stripe API and the transaction separate from the details of the client-side code. In a real application you absolutely should use a tokenized client-side setup as shown in Chapter 3, *Client-Side Payment Authentication*, on page ?. This chapter's implementation is not the best practice from a security standpoint for a production environment.

Let's get started by looking at payment gateways.

What's a Payment Gateway and Why Do I Need One?

A *payment gateway*, such as Stripe,[1] Braintree,[2] Authorize.net,[3] and others, is a service that mediates financial transactions. On one side of the transaction is the entire global financial system in all its complexity. On the other side is me, some clown with a website. The transaction is just slightly asymmetric.

From our point of view, we want to be able to say, "Here's a customer with a credit card and a transaction," and have the customer's money show up in our account. On the other side, a lot of companies and complexities are involved in that transaction. A typical credit card transaction might involve five or more financial organizations:

- The bank where the purchaser keeps their money
- The bank that issued the purchaser's credit card
- The credit card network (such as Visa or MasterCard)
- The bank that handles credit card transactions on behalf of the merchant
- The bank where the merchant keeps its money

That's a minimum—there may also be interbank networks, insurance companies, governments, and the like. It's brain-melting.

The payment gateway enables us not to have to think about any of this process. We send an API call or two to the gateway, they tell us whether the payment has been successful, and at some slightly future date, the money shows up in our bank account, minus a transaction fee. Gateways also often provide additional services, such as recurring billing, their own fraud protection, some reporting, mediation services for disputes, and on and on.

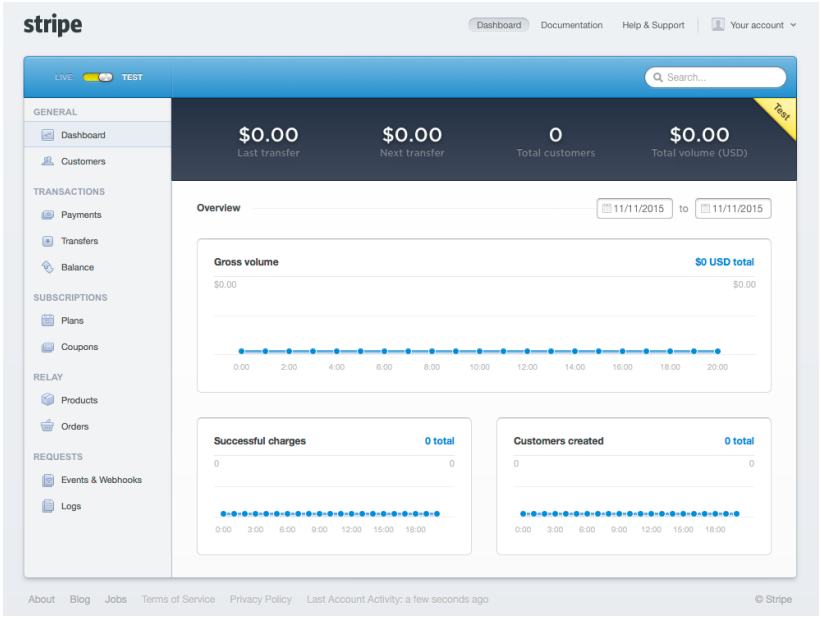
1. https://stripe.com
2. https://www.braintreepayments.com
3. http://www.authorize.net

Setting Up Our First Payment Gateway

We'll be using Stripe as our first payment gateway. In my experience, Stripe is the easiest of the gateways to get started with, but the basic API concepts are the same across most of the gateways that are not PayPal.

Before we do anything with Stripe, we need to set up an account. The account will provide us with API keys that we'll use for authentication, and we'll also get a dashboard with some nice reporting services.

Creating our stripe account is pretty simple. We go to http://dashboard.stripe.com/register and enter an email address and a password. Stripe then sends us an email to authenticate the account, which you must do before you start accepting payments. Once we do so, Stripe takes us to our dashboard, as shown in the following figure.

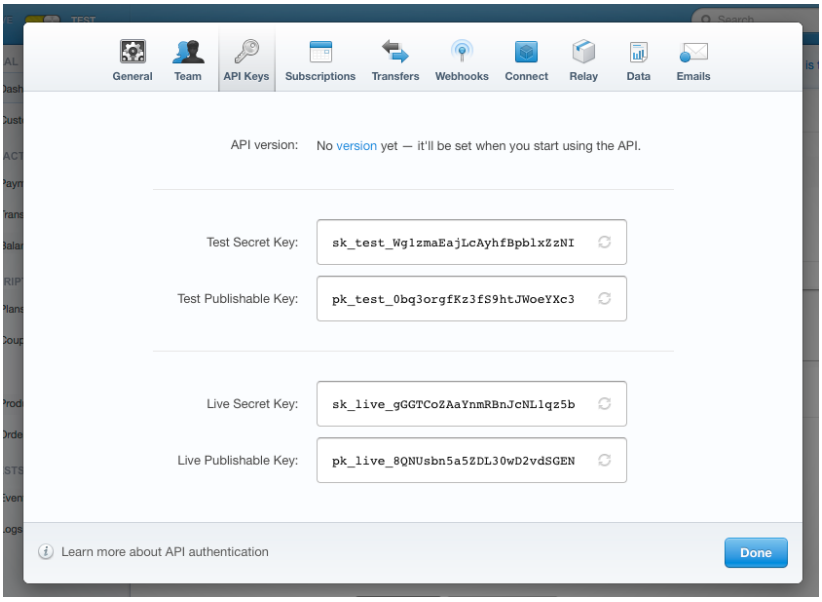


We'll look at the specifics of this dashboard UI later in Chapter 7, *The Administration Experience*, on page ?, but for the moment there are two particular bits worth noting.

First, in the upper-left corner and below the Stripe logo, is a big-ol' switch that is currently pointed at "Test." Stripe accounts come with a test mode, which has its own API keys that we can use in development and staging so that we can test out the account without racking up actual credit card charges. We're going to stay in Test mode for pretty much all the work we're doing in

this book. To start taking real charges in Live mode, we'll need to fill out a form and get approval from Stripe, but we'll talk more about this when we discuss going to production in Chapter 13, *Going to Production*, on page ?.

Second, in the upper-right corner is a menu labeled "Your Account." Within that pull-down menu is a menu item called "Account Settings." If we go there, we see rather a lot of settings. The tab we are interested in is "API Keys." Select this tab, and, as you see in the following figure, we see four API keys.



The circle icon to the right of each key gives you a new random key. (Don't worry, I updated all four keys right after I took this screenshot.) We have separate key sets for Test mode and Live mode. And, as is typical with this kind of API key, the key is split into a publishable key, which we can send out to a client, and a secret key, which should stay on our server. Stripe kindly sets up each key with a prefix telling you what kind of key it is.

As of right now, we're ready to take test credit card purchases with Stripe, and the only thing standing between us and real purchases is creating a short form and getting Stripe's approval.

Now, let's wire our Stripe account into our Rails application.

Telling Rails About Stripe

Now that we have a Stripe account, we need to make sure our Rails application uses that account. As is common for API authentication, we have been given long, random tokens that will be recognized by the API servers. One of those

tokens, the public one, will eventually be sent via the browser to the client and used to identify our application to Stripe's servers. The other token, the secret one, will only live on our server and be sent to Stripe. You can't get Stripe to transfer any money without both the public token and the secret token being sent to it.

The trick here is that the secret token is, well, secret. You don't want to pass it around via email or chat, and you don't want it to sit in your GitHub repo. At the same time, the production and staging servers will use different tokens, and you'll most likely want the ability to have individual developers use their own tokens in test mode.

There are several ways to make secret tokens work without exposing them. The one we're picking is simple and flexible, but other ways may make more sense depending on your development environment.

First, we'll add two gems to our Gemfile:

```
gem "dotenv-rails"
gem "stripe"
```

The dotenv-rails gem is what is going to allow us to customize the environment keys securely. This gem is a Rails wrapper for the Dotenv[4] gem, which will be installed as a dependency. The Dotenv gem allows you to set up a file named .env and populate it with a bunch of key/value pairs. When the Rails application starts, the gem places those values in Ruby's global ENV hash, just as though they were defined as systemwide environment variables.

The Stripe[5] gem is the official Ruby interface to the Stripe API. We'll be using it to communicate with Stripe, so we may as well get it in here now.

4. https://github.com/bkeepers/dotenv
5. https://github.com/stripe/stripe-ruby

Here is a quick way to add our keys into the system during development:

Create an .env file. But, before you do so, confirm that .env is in your application's .gitignore (or whatever your source control uses to not store particular files)—you do *not* want the .env file to be saved to source control, nor do you typically want it to be on your production server—you'll secure the secrets a different way in production. The format of the .env file is <key>=<value>. Yours should look something like this:

```
STRIPE_SECRET_KEY=sk_test_<YOUR KEY HERE>
STRIPE_PUBLISHABLE_KEY=pk_test_<YOUR KEY HERE>
```

You can also ensure that your environment variables are only used in specific Rails environments by creating files with names like .env.development or .env.test, which allows you to use different keys for live development versus testing.

It doesn't actually matter what you call the environment variables on the left side of each statement as long as you are consistent. As written, we'll now be able to refer to them within our Rails code as ENV["STRIPE_SECRET_KEY"] and ENV["STRIPE_PUBLISHABLE_KEY"].

We're not going to stop there, though. I'd like to add one more layer of indirection using the Rails secrets.yml file. Add the following to the file (you'll need to add it for both the development and test environments):

server_charge/01/config/secrets.yml

```
development: &default
  admin_name: First User
  admin_email: user@example.com
  admin_password: changeme
  domain_name: example.com
  secret_key_base: 837442c904d7b579c6c98618efa4b892abb377865f6e9507070787425f
  stripe_publishable_key: <%= ENV["STRIPE_PUBLISHABLE_KEY"] %>
  stripe_secret_key: <%= ENV["STRIPE_SECRET_KEY"] %>
  paypal_client_id: <%= ENV["PAYPAL_CLIENT_ID"] %>
  paypal_client_secret: <%= ENV["PAYPAL_CLIENT_SECRET"] %>
  host_name: "6e0fd751.ngrok.com"
  authy_key: <%= ENV["AUTHY_KEY"] %>
  rollbar: '7f4b0337a9c14f069cb41c1738298fee'
test:
  <<: *default
  secret_key_base: 8489bb21c9497da574dcb42fea4e15d089606d196e367f6f4e166acb281
```

The &default and <<: *default are bits of YAML syntax that allow us to use our development secrets as the basis for the other environments, so that we don't have to retype identical secrets in multiple environments. The word default is an arbitrary label, and the & is an *anchor*, which signifies that everything subordinate to the label is named by the label. The <<: indicates a merge,

and the * indicates a reference to a previous label. Together this means that all the key/value pairs belonging to the label should be merged in at the point of the symbol. Any subsequent keys below the merge override the merged-in key/value.

Now we can refer to the Stripe keys within our application as Rails.application.secrets.stripe_secret_key and Rails.application.secrets.stripe_publishable_key. Note that in production we have the choice of continuing to refer to environment variables or having the secrets.yml file actually contain the keys themselves.

Initializing Stripe

Finally, we need to initialize Stripe by telling the Stripe gem what our secret key is. The Stripe gem passes this to the Stripe server as part of all our Stripe API calls. I do this with a brief file in config/initializers/stripe.rb, which raises an exception on startup if the API value isn't there.

server_charge/01/config/initializers/stripe.rb

```
Stripe.api_key = Rails.application.secrets.stripe_secret_key
raise "Missing Stripe API Key" unless Stripe.api_key
```

That's possibly more indirection than strictly necessary—you could leave out the secrets file and just directly reference the environment variable when setting Stripe.api_key. The main reason to add the indirection is to allow the rest of the application to not care about environment variables, which are, after all, implementation details of the system.

More pragmatically, setting up the API key via the Rails secret file gives us more flexibility when deploying or setting up the application in a different environment. We can either set the environment variables or create a local secrets.yml file with a local API key. Because our application will need to run on different server environments, development environments, and test servers, this flexibility can be valuable.