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

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

Creating Subscription Plans

From Stripe's perspective, a subscription is a relationship between a *plan* and a *customer*, both of which are data objects defined by the Stripe API. We need information about both plans and customers in our own database, so we'll need analogs of both data objects.

As far as Stripe is concerned, a plan has the following parts:

- An amount, which is what the user is charged per subscription term. In U.S. currency, the amount is in cents.
- An id, which we'll store in our database as remote_id to distinguish it from our local ActiveRecord database ID. This ID is generated by us; Stripe doesn't care what it is, as long as it's unique within our list of plans.
- An interval and interval_count, which specify the length of the term between user billings. The interval is one of daily, weekly, monthly, or yearly. The interval count, which defaults to 1, specifies the number of intervals in a term. If you want to bill the user every 12 days, 3 months, or 7 years, for whatever reason, you'd specify that information by using the interval_count.
- A human-friendly name, which can be seen on the Stripe administrative dashboard.
- A number of trial_period_days, which defaults to nil (meaning 0). By setting this value for a plan, you are stating that all subscribers have this number of days as a trial period. Effectively, you are delaying the date of their first payment.

Although you can create plans via the Stripe dashboard, we're going to create them from inside our application and send that information to Stripe because we need to keep track of the plans locally, especially since we want to add some additional information to plans. First, we need a database migration to create an ActiveRecord model for plans:

```
% rails generate model plan remote_id:string name:string \
  price:monetize interval:integer interval_count:integer
  tickets_allowed:integer ticket_category:string \
  status:integer description:text
```

For our local plan object, we've added the attributes tickets_allowed and ticket_category to deal with the local logic of managing how many tickets a particular subscription entitles a user to obtain. Eventually, we'll have to incorporate this logic into the payment workflow, because a user with a subscription won't need to pay for tickets that are covered.

To use these plan objects to get subscriptions working, we'll need not just a model but also a workflow. Right now, however, the only thing we need in the model is a way to get the remote Stripe object:

subscription/01/app/models/plan.rb

```
class Plan < ApplicationRecord

  enum status: {inactive: 0, active: 1}
  enum interval: {day: 0, week: 1, month: 2, year: 3}

  monetize :price_cents

  def remote_plan
    @remote_plan ||= Stripe::Plan.retrieve(remote_id)
  end

  def end_date_from(date = nil)
    date ||= Date.current.to_date
    interval_count.send(interval).from_now(date)
  end

end
```

We can use a simple workflow to create plans and coordinate them with Stripe. All this workflow does is take a bunch of incoming parameters and pass them along to Stripe, along with some defaults, such as currency:

subscription/01/app/workflows/creates_plan.rb

```
class CreatesPlan

  attr_accessor :remote_id, :name,
      :price_cents, :interval,
      :tickets_allowed, :ticket_category,
      :plan

  def initialize(remote_id:, name:,
      price_cents:, interval:,
      tickets_allowed:, ticket_category:)
    @remote_id = remote_id
    @name = name
    @price_cents = price_cents
    @interval = interval
    @tickets_allowed = tickets_allowed
    @ticket_category = ticket_category
  end
  def run
    remote_plan = Stripe::Plan.create(
        id: remote_id, amount: price_cents,
        currency: "usd", interval: interval,
        name: name)
    self.plan = Plan.create(
        remote_id: remote_plan.id, name: name,
        price_cents: price_cents, interval: interval,
        tickets_allowed: tickets_allowed, ticket_category: ticket_category,
        status: :active)
  end

end
```

If the Stripe creation is successful, the next step is to create a local plan. If the Stripe creation isn't successful, the Stripe gem will throw an exception and the whole thing will stop.

We're not wiring this to a controller or anything yet; that'll be a part of the administration tool that we will discuss in Chapter 7, *The Administration Experience*, on page ?. Odds are you'll rarely create plans. Let's start with a simple Rake task that will create a few plans:

subscription/01/lib/tasks/plan_creation.rake

```
namespace :theater do

  task create_plans: :environment do
    plans = [
        {remote_id: "orchestra_monthly", plan_name: "Orchestra Monthly",
         price_cents: 10_000, interval: "monthly", tickets_allowed: 1,
         ticket_category: "Orchestra"},
        {remote_id: "balcony_monthly", plan_name: "Balcony Monthly",
         price_cents: 60_000, interval: "monthly", tickets_allowed: 1,
         ticket_category: "Balcony"},
        {remote_id: "vip_monthly", plan_name: "VIP Monthly",
         price_cents: 30_000, interval: "monthly", tickets_allowed: 1,
         ticket_category: "VIP"}
    ]
    Plan.transaction do
      plans.each { |plan_data| CreatesPlan.new(**plan_data).run }
    end
  end

end
```

All this does is invoke the CreatesPlan workflow for each of a set of sample plans.

Creating Subscription Customers

To actually create a subscription and have it start charging, we need to create a customer object in Stripe and associate it with a plan. Stripe will create an internal subscription object for that customer-to-plan relationship. So, we need to coordinate user objects between our database and Stripe's API.

To register a customer with Stripe, all we need to do is say, "Hey, Stripe, give me a customer ID" (technically, `Stripe::Customer.create`). Once the Stripe customer is created, we can actually see it on the Stripe dashboard—we can even do some administration there should we choose. It's our responsibility to coordinate the customer records by adding the Stripe ID to our customer record:

subscription/01/db/migrate/20160730192814_add_stripe_customer_to_user.rb

```ruby
class AddStripeCustomerToUser < ActiveRecord::Migration[5.0]

  def change
    add_column :users, :stripe_id, :string
  end

end
```

There are a few things we can do to make the customer object more useful. First, we can send it a `description` string and an arbitrary `metadata` argument. Now when we browse the users in the Stripe dashboard, we'll have a little more context.

Second, we can assign payment sources, such as credit cards, to a customer. This associates the source with the customer within Stripe so that the customer can be charged at a later date. This is useful for subscriptions, since we'll need to keep charging the customer over and over again. It's also useful for regular payments, because it allows customers to check out without reentering their credit card information. Associating a card with a user is so common that the `Stripe::Customer.create` method can take a credit card as an argument. This credit card can either be a full set of credit card data or a token of the kind we've already been using.

Finally, we can also add subscription plans to a user by passing the Stripe ID of the plan. Once a user has a subscription plan and a payment source associated with that user, Stripe will charge the user and then charge the user again when the plan interval expires. Associating a user with a plan is so common that the `Stripe::Customer.create` method can take a subscription plan ID as an argument.

For our part, we're still responsible for managing subscription data locally, so we need a place to store it:

```
% rails generate model subscription user:references \
  plan:references start_date:date end_date:date \
  status:integer payment_method:string remote_id:string
```

In our database, a subscription combines a user and a plan. The start date of the subscription, the current end date, and a status are also stored. For payment, we have a payment method and a remote ID, similar to what payments have. We'll continue to update the end date as new payments come in to move it further into the future.