Rails-Stripe-Membership-Saas

Last updated 25 Mar 2015

GitHub Repository (https://github.com/RailsApps/rails-stripe-membership-saas) · Issues (https://github.com/RailsApps/rails-stripe-membership-saas/issues)

Membership Site with Stripe

Introduction

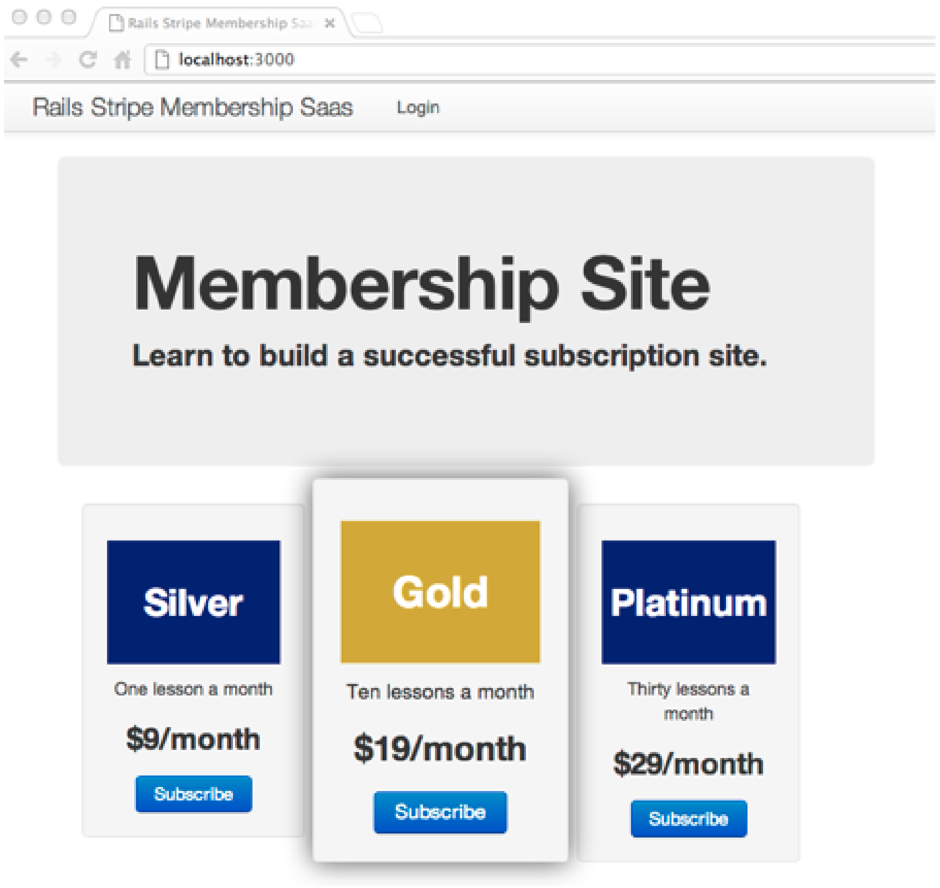
This tutorial shows how to build a Rails application with recurring billing using Stripe (https://stripe.com/) and the Payola (https://www.payola.io/) gem. After reading this tutorial, you will be able to implement payment processing for a Rails membership site, subscription site, or SaaS site (software-as-a-service). The application requires:

- Rails 4.2

See the article Updating to Rails 4.2 (http://railsapps.github.io/updating-rails.html) if you are using Rails 4.1. The application uses Active Job for background processing. You must install Rails 4.2 to use this tutorial because Active Job is not available in earlier versions of Rails.

We use Pete Keen's Payola (https://www.payola.io/) gem to implement recurring billing. The application uses Devise (https://github.com/plataformatec/devise) for authentication and user management. When a user purchases a subscription, the application creates a user account using Devise.

Screenshot



This example application exists so you don't have to build it yourself. It aims to:

- eliminate effort spent building an application that meets a common need;
- offer code that is already implemented and tested by a large community;
- provide a well-thought-out app containing most of the features you'll need.

Is It for You?

This tutorial shows how to sell a product online using the Payola (https://www.payola.io/) gem for Stripe payment processing. The application builds on other RailsApps example applications. To fully understand the application, you should read:

- Learn Ruby on Rails (https://tutorials.railsapps.org/tutorials/learn-rails)
- Devise Quickstart Guide (https://tutorials.railsapps.org/tutorials/devise-quickstart)
- Role-Based Authorization (https://tutorials.railsapps.org/tutorials/rails-devise-roles)
- Mailing List with Active Job (https://tutorials.railsapps.org/tutorials/rails-mailinglist-activejob)
- Signup and Download Tutorial (https://tutorials.railsapps.org/tutorials/rails-signup-download)
- Stripe JS With Coupons (https://tutorials.railsapps.org/tutorials/rails-stripe-coupons)

Stripe is available for businesses that have bank accounts in any of 18 countries, including the US, Canada, Australia, and the largest European nations (see a list (https://stripe.com/global)).

Functionality

Membership sites restrict access to content such as articles, videos, or user forums. Software-as-a-service (SaaS) sites limit use of web-based software to paid subscribers. The revenue model is the same whether the site provides high-value content or software as a service: A visitor purchases a subscription and gains access to restricted areas of the site. Typically, the subscription is repurchased monthly through a service that provides recurring billing.

If you’re planning to build a SaaS application, a membership site, or some other subscription-based web service, your application will need the following rudimentary functionality:

- content or web functionality to deliver value
- landing pages to convert visitors to paying customers
- user management to register or remove users
- access control to limit site-wide access to authenticated users
- authorization management to restrict access to content or services based on role or other characteristics
- account management to maintain records of subscription status
- a recurring billing system for periodic payment transactions

Features

The example application provides a complete and fully functional membership site.

- tiered pricing for multiple subscription plans
- uses Stripe for no local credit card storage
- Stripe accepts credit card payments from customers in any country or currency
- PCI compliance using the Stripe JavaScript library
- Stripe handles recurring billing, retries if payment fails, and cancels subscription if retries fail
- paid subscriptions are created only after a successful credit card transaction
- subscribers can upgrade or downgrade subscription plans
- subscribers can cancel subscription plans
- configurable subscription renewal period (defaults to one month)

What is Not Implemented

There are additional features you may want for a SaaS application, such as:

- Basecamp-style subdomains (each user gets their own subdomain)
- multitenancy (http://en.wikipedia.org/wiki/Multitenancy) database segmentation

For multitenancy, try Brad Robertson’s Apartment (https://github.com/bradrobertson/apartment) gem and see the book Multitenancy with Rails (https://leanpub.com/multi-tenancy-rails) by Ryan Bigg.

Concepts

Credit cards are the most-used method for making payments on the web. Web application developers strive to make credit card payment easy and secure. When we make it easy to make a purchase, we increase sales. Ideally, we want a customer to do nothing more than click a “Purchase” button and fill out a form with a credit card number, expiry date, and a card security code (http://en.wikipedia.org/wiki/Card_security_code). The simpler the sales process, the higher the conversion rate.

Security concerns make it difficult to take credit card payments. When credit card data is processed and stored by your application, your site becomes a target for thieves and you are liable for security breaches. Storing credit card data on a server requires compliance with Payment Card Industry Data Security Standards (https://www.pcisecuritystandards.org/) (PCI-DSS). Credit card payment was once difficult to implement, but it has become easier and more secure.

Payment Approaches

Any business needs a merchant account (http://en.wikipedia.org/wiki/Merchant_account) in order to accept credit card payments. Usually a merchant account is obtained from a bank or Independent Selling Organization (ISO) and approval requires preparation of an application and financial documents. The pricing structure for merchant accounts is notoriously opaque, hiding costs in a complex matrix of add-on fees. Unlike other providers of online billing services, Stripe provides a merchant account as part of the service, without the delay of an application process. Stripe's pricing structure is also simpler.

Web merchants rely on payment gateways (http://en.wikipedia.org/wiki/Payment_gateway) to process credit card charges. The largest providers of payment gateways are companies such as Authorize.net (http://www.authorize.net/) which provide processing for the millions of physical point-of-sale terminals found in retail stores. Stripe provides a payment gateway, eliminating the need to establish an account with a company such as Authorize.net.

The earliest ecommerce web applications were built with forms to accept credit card data, subsequently submitting the data to a third-party payment gateway such as Authorize.net, obtaining an approval code, and storing the encrypted credit card data in a database for a user's future purchases. Integration of the third-party payment gateway with a web application was often difficult. Worst of all, the web application directly handled customer credit-card information, creating risk and requiring additional work to meet PCI-DSS compliance requirements.

Services such as PayPal, Google Checkout, Amazon Payments offered an alternative approach. Instead of processing a payment form on your own website and sending credit card data to a third-party payment gateway, PayPal and the others host payment pages. A customer clicks a button on your website and is redirected to a hosted payment page. The security of the payment page and the customer's credit card data becomes the responsibility of PayPal (or Google or Amazon). No merchant account is required and the costs are easier to determine. Unfortunately, PayPal (particularly) is notorious for jeopardizing merchant revenues by freezing accounts without warning or recourse. The biggest problem is a clunky user interface and a lack of integration between your own website and the hosted payment page. The hosted payment page will not match the look and feel of your site. Even worse, you'll lose key marketing information such as referrer data or promotional codes that will no longer be associated with the transaction because the hosted payment page introduces a break in the checkout flow.

Stripe

Stripe is a third-party billing service that provides an API and gem for integration with Rails applications. There are several other third-party billing services; Stripe is the least expensive and most popular for low-volume startups. Stripe costs 2.9% + 30¢ per successful charge, with no monthly or setup fees (see Stripe pricing (https://stripe.com/help/pricing)).

Stripe offers a radical improvement over previous payment processing approaches. It provides a one-stop service that combines a merchant account and payment gateway with a simplified pricing structure. The company is developer-friendly, providing a modern API and software libraries for all major languages and platforms, including a Stripe Rails gem. Stripe payment processing can be easily integrated with a web application without hosted payment pages, avoiding disruption of the customer user experience, or breaks in the checkout flow. Finally, Stripe provides security, meeting PCI-DSS compliance by offloading credit card transactions from your site to the Stripe servers.

From a technical viewpoint, Stripe's innovation is the use of JavaScript to collect the customer credit card data and transmit it to the Stripe servers for processing. Stripe provides JavaScript that you can add to any payment page in your application. The script strips the credit card data from the form, securely transmits it to

Stripe servers, and returns an authorization token that replaces the credit card data. When your application processes the form, there is no credit card data, only an authorization token. The authorization token has no value to thieves so you have no vulnerability. Your application can use the authorization token to initiate a credit card transaction.

Stripe offers two versions of its JavaScript implementation. Stripe.js (https://stripe.com/docs/stripe.js) allows you to add Stripe payment processing to any form in your application. Stripe.js requires some experience with JavaScript but it offers flexibility to accommodate any form. Stripe Checkout (https://stripe.com/checkout) is a newer product that is easier to install but less flexible. With Stripe Checkout, Stripe aims to compete with the button-based payment options from Google, PayPal, or Amazon. No experience with JavaScript is necessary. One line of code adds Stripe Checkout to any page. However, Stripe Checkout is intended for the simplest and most common use cases, specifically the sale of a product that requires only an email address and credit card data from the customer. The Stripe Checkout pop-up form cannot be customized or modified to add any additional fields.

The Stripe JS With Coupons (https://tutorials.railsapps.org/tutorials/rails-stripe-coupons) tutorial shows how to use Stripe.js (https://stripe.com/docs/stripe.js) directly to sell access to digital products such as ebooks.

This tutorial shows how to integrate the Payola (https://www.payola.io/) gem with the Devise user registration process. The Payola gem provides the full functionality of Stripe.js (https://stripe.com/docs/stripe.js) with additional features you need for a membership site. Subscription sales are more complex than digital product sales. The Payola gem provides all of the functionality required for a subscription site, so there is less to implement. The tests that are part of the gem, and the community of support, are further reasons we use the gem instead of implementing from scratch.

Payment Transaction

This tutorial uses the Devise gem for account registration and user management. To create a user account, the visitor must provide an email address as well as a password (actually, a password and password confirmation). The form requires a credit card number, card verification code, and a card expiry date.

When the visitor submits the form, the Stripe JS script (provided as part of the Payola gem) makes a round trip to the Stripe servers, verifies the card (but does not initiate a payment yet) and gets a Stripe token. When the script receives the Stripe token, execution ends and our form is automatically submitted to our application, containing everything we need: email address, password, password confirmation, and Stripe token.

It is important to recognize that though the user sees a form with a button that says "Purchase," and the Stripe JS script submits credit card data to the Stripe servers, the actual purchase transaction is not initiated until our application receives the Stripe token. As in any Rails application, the parameters submitted from our form are received by a controller's `create` action.

We use a Registrations controller, inheriting from a Devise Registrations controller, to create the user account. We add a `subscribe` method that initiates a payment transaction and establishes recurring billing.

Other Features

We use the Devise registration mechanism to sign up the user for an account. When Devise creates a user object and adds it to the database, we'll also connect to MailChimp and add the user's email address to a mailing list. This functionality is provided by the rails-signup-download (https://github.com/RailsApps/rails-signup-download) starter application.

Architecture and Implementation

Here is a high-level abstraction of the application, as a list of systems:

- user management with Devise (to register or remove users)
- authentication with Devise (sign in and sign out)
- access control using simple role-based authorization (http://railsapps.github.io/rails-authorization.html)
- subscription account management
- recurring billing with Stripe
- landing pages
- content or service pages

User Management, Authentication, and Authorization

The Devise Tutorial (https://tutorials.railsapps.org/tutorials/devise-quickstart) for the rails-devise (https://github.com/RailsApps/rails-devise) example application shows how to set up user management and authentication using Devise.

Users are managed with the User model, which has attributes for email and password, as well as some fields provided by Devise such as `sign_in_count`. The Devise gem provides its own controllers for managing sessions, registration, email confirmation and similar functions. You won't see these controllers as they are hidden in the gem itself.

The Role-Based Authorization Tutorial (https://tutorials.railsapps.org/tutorials/devise-quickstart) for the rails-devise-roles (https://github.com/RailsApps/rails-devise-roles) example application shows how to set up role-based authorization. We use a feature of Active Record, Enum (http://api.rubyonrails.org/classes/ActiveRecord/Enum.html), to add roles to a User model. Each role will correspond to a subscription plan, so different plans can access different content or services.

Account Management and Recurring Billing

An account management system keeps subscription records so the access control system can determine which users are current subscribers. We'll combine services offered by Stripe with user management provided by Devise for our account management system.

Stripe will provide the recurring billing system to store the users' credit card data and initiate payment transactions.

The Payola gem uses the Stripe API to create a new customer and specify a subscription plan. When the customer's subscription expires due to failed payment, the Payola gem uses Stripe "webhooks" to update the application's user records.

We'll provide options for the user to upgrade or downgrade subscription plans, and cancel a subscription as an extension of the user management system provided by Devise.

Two approaches are possible in building a recurring billing system. You could implement a complete billing management system as part of the application. This would require building a mechanism to check for expiring subscriptions (typically a daily cron job) and initiate payment requests through Stripe when a user's account comes due. With this approach, you would use Stripe only for processing credit card transactions. But there's no reason to implement recurring billing yourself. Stripe provides a complete, well-tested, and hosted mechanism for recurring billing. We'll use Stripe's API to supply the recurring billing services we need.

A key requirement for the application is to keep the recurring billing and account management systems in sync. We face a problem if we establish a new subscription, hand off recurring billing to Stripe, and then months later find that the subscriber's credit card has expired and can no longer be billed. We need a mechanism to update our subscription status when Stripe encounters a declined transaction. Stripe provides "webhooks" to set the status of a subscription. When Stripe encounters a declined transaction it will initiate an HTTP request to our application which the Payola gem will decode to change a subscription status.

If we didn't use the Stripe webhooks, we'd have to either query the Stripe API on each login or run a repeating cron job to check for subscription expiration. The application will be notified immediately by Stripe so there is no need for the overhead of checking on each login. The Stripe webhook mechanism is very robust: If for some reason it cannot make an HTTP request to our application, it will retry several times with exponential backoff.

A key requirement for any ecommerce site that takes credit cards is PCI compliance (http://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard) to minimize risk of customer credit card exposure. Using Stripe, your server will never receive sensitive credit card details. Instead the Payola gem uses the Stripe JavaScript library on our subscription payment form which sends the credit card details directly to the Stripe servers. Your business can easily meet PCI compliance requirements of the "PCI DSS Self-Assessment Questionnaire A" if you solely accept payment information through the Stripe JavaScript library and serve your payment page over SSL.

Landing Pages

Landing pages serve to describe the value of the content or service and convince the visitor to purchase a subscription. For our example application, the home page of the application is our landing page.

Content or Service

We'll create placeholder pages for content.

For your application, the content can be anything you like: photo galleries, videos, downloadable ebooks. For a SaaS site, subscribers would gain access to a web application.

The Object Model

Software engineering attempts to model real-world entities and behaviors. As developers, we try to choose descriptive names for objects and methods to reduce ambiguity and increase understanding. For this application, a User is our most important object. In other projects, we might call this object an "Account" or "Member."

Users have several important attributes: email address, password, subscription plan. A user also has less important attributes such as name or creation date. Any of these attributes could be separate objects that are associated with the user through an id or key. We'll make "Plan" a separate object associated with a user.

We won't include a credit card number as an attribute of a user because we don't want the vulnerability of storing a credit card number in our database. Instead, we'll send the credit card number directly to Stripe and obtain a Stripe customer id that serves as an indirect reference to the credit card number when we need to ask Stripe to begin billing a user.

Our authorization system is based on the concept of roles. The user's role constrains his or her access to the website's content pages. Though "Subscription Plan" and "Role" appear to be distinct concepts, in this application they functionally overlap. Each user will have a role id that describes the subscription plan (or access level) that he or she has purchased.

Accounts You May Need

Before you start, you will need accounts for *recurring billing*, a *merchant account*, *hosting*, and *email*

Billing

This tutorial shows to set up recurring billing using Stripe. Before you start, go to the Stripe website (https://stripe.com/) and set up an account. You don't need a credit card merchant account or payment gateway. There's no approval process to delay getting started.

You may wonder if Stripe is the best choice for a payment provider. Many providers of billing services want your business:

- Stripe (https://stripe.com/) (2011)
- SaaSy (http://saasy.com/) (2011)
- Fusebill (http://www.fusebill.com/) (2011)
- Recurly (http://recurly.com/) (2010)
- SubscriptionBridge (http://www.earlyimpact.com/subscriptionbridge/) (2010)
- Chargify (http://chargify.com/) (2009)
- CheddarGetter (https://cheddargetter.com/) (2009)
- Braintree (http://www.braintreepayments.com/) (2007)
- Spreedly (http://spreedly.com/) (2007)
- Zuora (http://www.zuora.com/) (2007)
- Adyen (http://www.adyen.com/) (2006)
- Vindicia (http://www.vindicia.com/) (2003)
- Aria Systems (http://www.ariasystems.com/) (2003)

The list shows the year each service was founded. In general, since the market is highly competitive, the newer services are less expensive and offer better integration, interfaces, and features. Here is a blog post from 2011 comparing Stripe's pricing to other services: Stripe's New Online Payments Service (http://expletiveinserted.com/2011/10/02/stripes-new-online-payments-service-wheres-the-catch/).

Merchant Account

Unlike other providers of billing services, Stripe provides a merchant account as part of the service.

You'll need a merchant account in order to accept credit card payments. Here's an explanation from Phillip Parker of CardPaymentOptions.com (http://www.cardpaymentoptions.com/): "A merchant account is a line of credit account that allows a business to accept card payments from its customers. Similar to how a checking account allows you to deposit another person's check into your checking account, a merchant account allows you to accept a card payment from a customer. Unlike a checking account, a merchant account doesn't hold money. Instead, a card payment passes through the merchant account and is deposited into a checking account after the funds have been cleared through the merchant account."

Hosting and SSL

Visitors to your website will send credit card information from their browser to Stripe's servers when they sign up for a subscription. The Stripe JavaScript library will open an SSL connection to Stripe's servers when the form is submitted.

You can host your membership site without SSL and your users' credit card numbers will be protected on the way to Stripe's servers. However, your security-conscious visitors will be uneasy if they see that the web URL for your registration page begins with `http://` and not `https://` (indicating an SSL connection). For their peace of mind (and the higher conversion rate that comes with trust), you should host your website with an SSL connection. Additionally, as a general practice, it is wise to host any web application that requires login over an SSL connection.

If you're deploying with Heroku, you can access any Heroku app over SSL at `https://myapp.herokuapp.com/`. For your custom domain, Heroku offers the SSL Endpoint add-on (https://devcenter.heroku.com/articles/ssl-endpoint) for a fee of $20/month. You'll need to purchase a signed certificate from a certificate provider (https://devcenter.heroku.com/articles/ssl-certificate) for an annual fee (typically $20 a year). Setting up an SSL certificate for a custom domain on Heroku can be a hassle but there's a convenient alternative that is a better value. You can purchase CloudFlare (http://cloudflare.com/) for $20/month and get SSL without purchasing or installing an SSL certificate. CloudFlare is a content delivery network (CDN) and website optimizer; the $20/month CloudFlare Pro plan includes SSL (http://blog.cloudflare.com/easiest-ssl-ever-now-included-automatically-w). If you use Cloudflare in combination with Heroku hosting, you can use the Heroku piggyback SSL to encrypt the traffic between Heroku and Cloudflare, and your website visitors will connect to Cloudflare with their web browsers, providing a complete SSL connection through Cloudflare to Heroku with your custom domain. Not only do you get SSL for no more than you'd pay at Heroku to use an SSL certificate, but you get the Cloudflare CDN services as part of the bargain.

If you're deploying on Heroku, you can wait until you've deployed to sign up for a Cloudflare account. See the article Configuring CloudFlare DNS for a Heroku App (http://www.higherorderheroku.com/articles/cloudflare-dns-heroku/) for instructions.

If you're deploying elsewhere, do your research early to find out how to set up SSL and apply for an SSL certificate if necessary.

Email Service Providers

You'll need infrastructure for three types of email:

- company email
- email sent from the app ("transactional email")
- broadcast email for newsletters or announcements

No single vendor is optimal for all three types of email; you likely will use several vendors. See the article Send Email with Rails (http://railsapps.github.io/rails-send-email.html) for suggestions for various types of email service providers.

Starter App

We'll start by building the rails-signup-download (https://github.com/RailsApps/rails-signup-download) example application using the Rails Composer (http://railsapps.github.io/rails-composer/) tool. The Rails Composer tool creates a complete starter app including:

- PDF download
- roles for authorization

- Devise for authentication
- subscribes user to MailChimp mailing list
- Bootstrap or Foundation front-end framework
- RSpec testing framework

To understand the code generated by Rails Composer, refer to:

- Devise Quickstart Guide (https://tutorials.railsapps.org/tutorials/devise-quickstart)
- Role-Based Authorization (https://tutorials.railsapps.org/tutorials/rails-devise-roles)
- Mailing List with Active Job (https://tutorials.railsapps.org/tutorials/rails-mailinglist-activejob)
- Signup and Download Tutorial (https://tutorials.railsapps.org/tutorials/rails-signup-download)

The starter application will give you sign-up and sign-in pages nicely styled for Bootstrap or Foundation, plus a full test suite for the authentication and authorization features. Using Rails Composer, your starter app will include all this without extra work.

After Rails Composer generates the starter app, we'll customize the application for subscription billing.

Building the Application

See the article Installing Rails (http://railsapps.github.io/installing-rails.html) for instructions about setting up Rails and your development environment. Rails 4.2 must be installed in your development environment.

If you are using RVM, and you've created a gemset named "rails4.2" for Rails 4.2, select the gemset using this command:

```
$ rvm gemset use rails4.2
```

Check that Rails 4.2 is available:

```
$ rails -v
Rails 4.2.0
```

To build the starter app, run the command:

```
$ rails new rails-stripe-membership-saas -m https://raw.github.com/RailsApps/rails-composer/master/comp
oser.rb
```

You'll see a prompt:

```
option  Build a starter application?
    1)  Build a RailsApps example application
    2)  Contributed applications
    3)  Custom application
```

Enter "1" to select **Build a RailsApps example application**. You'll see a prompt:

```
option  Choose a starter application.
    1)  learn-rails
    2)  rails-bootstrap
    3)  rails-foundation
    4)  rails-mailinglist-activejob
    5)  rails-omniauth
    6)  rails-devise
    7)  rails-devise-roles
    8)  rails-devise-pundit
    9)  rails-signup-download
   10)  rails-stripe-checkout
   11)  rails-stripe-coupons
   12)  rails-stripe-membership-saas
```

Choose **rails-signup-download**. Be careful: Don't choose "rails-stripe-membership-saas." We want to build the basic "rails-signup-download" application and then add the custom code we need. If you select "rails-stripe-membership-saas" all the work will be done for you and you won't have any fun!

Choose these options to create the starter application for this tutorial:

- Web server for development? **WEBrick (default)**
- Web server for production? **Same as development**
- Database used in development? **SQLite**
- Template engine? **ERB**
- Test framework? **RSpec with Capybara**
- Continuous testing? **None**
- Front-end framework? **Bootstrap 3.3**
- Add support for sending email? **Gmail**
- Admin interface for database? **None**
- Install page-view analytics? **None**
- Prepare for deployment? **no**
- Set a robots.txt file to ban spiders? **no**
- Create a GitHub repository? **no**
- Use or create a project-specific rvm gemset? **yes**

If you have problems creating the starter app, ask for help on Stack Overflow (http://stackoverflow.com/questions/tagged/railsapps). Use the tag "railsapps" on Stack Overflow for extra attention. If you find a problem with the rails-signup-download (https://github.com/RailsApps/rails-signup-download) example application, open an issue on GitHub (https://github.com/RailsApps/rails-signup-download/issues).

Running the Application

After you create the application, switch to its folder to work directly in the application:

```
$ cd rails-stripe-membership-saas
```

Launch the application by entering the command:

```
$ rails server
```

To see your application in action, open a web browser window and navigate to http://localhost:3000/ (http://localhost:3000). Try out the starter application by clicking the "Sign up" button. You'll fill out a form with your name, email address, and password. After submitting the form, you'll see an acknowledgment message and a button which allows you to download a PDF file.

You can remove the account you created by clicking "Edit account" in the navigation bar. Or reset the database:

```
$ rake db:reset
```

Devise requires each account to have a unique email address, so it's necessary to remove the account or reset the database if you plan on creating the same user again.

If you've set up Unix environment variables with a MailChimp API key and a mailing list ID, the application will automatically subscribe the new user to a MailChimp mailing list. For details about obtaining a MailChimp API key, see the README for the rails-signup-download (https://github.com/RailsApps/rails-signup-download) starter application.

For our new application, our goal is to modify the sign-up page to require the visitor to enter credit card information before gaining access to the digital product.

Gems

Next, add the following gems to the application Gemfile:

- payola (https://github.com/stripe/payola) – Rails engine for Stripe payments

The Payola gem includes the Stripe (https://github.com/stripe/stripe-ruby) gem as a dependency, so it will be installed automatically.

Open your **Gemfile** and add the following:

```
gem 'payola-payments'
```

Install the Required Gems

Install the required gems on your computer:

```
$ bundle install
```

Keep in mind that you have installed these gems locally. When you deploy the app to another server, the same gems (and versions) must be available (this happens automatically on Heroku).

Configuration

We're using gems that provide access to servers for Stripe and MailChimp. We've just installed the Payola gem. The gibbon (https://github.com/amro/gibbon) gem, which provides an API wrapper for MailChimp, was already installed by Rails Composer for the rails-signup-download starter application.

We need to provide credentials to access the Stripe and MailChimp servers.

Credentials

We'll enter our Stripe and MailChimp credentials in the **config/secrets.yml** file.

Stripe

When users make a payment, the application will connect with the Stripe server. To complete the transaction, we'll need a Stripe API key. You can get the Stripe credentials when you log in to view the Stripe Dashboard (https://dashboard.stripe.com/account/apikeys). Look under "Account Settings" for the "API Keys" tab. If you use the "Test Secret Key" and "Test Publishable Key" during development, you can enter payments without getting charged. When you are ready to deploy the application to production, you'll need the "Live" keys.

We'll add these variables to the **config/secrets.yml** file:

- `stripe_api_key: <%= ENV["STRIPE_API_KEY"] %>`
- `stripe_publishable_key: <%= ENV["STRIPE_PUBLISHABLE_KEY"] %>`

MailChimp

When an account is created for a new user, the application will connect to MailChimp and add the user's email address to a mailing list. To access MailChimp, we'll need a MailChimp API key. Log in to MailChimp (https://admin.mailchimp.com/) to get your API key. Click your name at the top of the navigation menu, then click "Account". Click "Extras," then "API keys." You have to generate an API key; MailChimp doesn't create one automatically.

You'll also need to create a MailChimp mailing list. The MailChimp "Lists" page has a button for "Create List." The list name and other details are up to you. We'll need the `MAILCHIMP_LIST_ID` for the mailing list you've created. To find the list ID, on the MailChimp "Lists" page, click the "down arrow" for a menu and click "Settings." At the bottom of the "List Settings" page, you'll find the unique ID for the mailing list.

Set these variables in the **config/secrets.yml** file:

- `mailchimp_api_key: <%= ENV["MAILCHIMP_API_KEY"] %>`
- `mailchimp_list_id: <%= ENV["MAILCHIMP_LIST_ID"] %>`

Secrets File

I recommend that you set local environment variables, rather than exposing the credentials in the **config/secrets.yml** file. See the article Rails Environment Variables (http://railsapps.github.io/rails-environment-variables.html) if you need help with setting environment variables. Using environment variables, replace the **config/secrets.yml** file with this:

```
development:
  admin_name: First User
  admin_email: user@example.com
  admin_password: changeme
  email_provider_username: <%= ENV["GMAIL_USERNAME"] %>
  email_provider_password: <%= ENV["GMAIL_PASSWORD"] %>
  domain_name: example.com
  mailchimp_api_key: <%= ENV["MAILCHIMP_API_KEY"] %>
  mailchimp_list_id: <%= ENV["MAILCHIMP_LIST_ID"] %>
  stripe_api_key: <%= ENV["STRIPE_API_KEY"] %>
  stripe_publishable_key: <%= ENV["STRIPE_PUBLISHABLE_KEY"] %>
  secret_key_base: very_long_random_string

test:
  domain_name: example.com
  secret_key_base: very_long_random_string

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  admin_name: <%= ENV["ADMIN_NAME"] %>
  admin_email: <%= ENV["ADMIN_EMAIL"] %>
  admin_password: <%= ENV["ADMIN_PASSWORD"] %>
  email_provider_username: <%= ENV["GMAIL_USERNAME"] %>
  email_provider_password: <%= ENV["GMAIL_PASSWORD"] %>
  domain_name: example.com
  mailchimp_api_key: <%= ENV["MAILCHIMP_API_KEY"] %>
  mailchimp_list_id: <%= ENV["MAILCHIMP_LIST_ID"] %>
  stripe_api_key: <%= ENV["STRIPE_API_KEY"] %>
  stripe_publishable_key: <%= ENV["STRIPE_PUBLISHABLE_KEY"] %>
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Be sure to use spaces, not tabs. Make sure there is a space after each colon and before the value for each entry or you will get a message “Internal Server Error: mapping values are not allowed” when you start the web server.

Now that we’ve configured our application, we can begin adding code.

Payola

The Payola (https://github.com/stripe/payola) gem provides a Rails engine (http://guides.rubyonrails.org/engines.html) for interaction with Stripe. In effect, Payola is a modular web application that you’ll add to you application to handle subscriptions. We’ve already added the gem to the Gemfile. Now we’ll set up Payola.

Payola Generator

The Payola gem includes a generator. The generator will:

- set up an initializer file
- add JavaScript to the application
- add the Payola Rails engine to the application routes

Run the Payola generator:

```
$ rails g payola:install
initializer  payola.rb
     insert  app/assets/javascripts/application.js
      route  mount Payola::Engine => '/payola', as: :payola
```

The generator adds a file, modifies the **application.js** file, and modifies the application routes.

Migrate

The Payola application contains several models that require database tables. The Payola gem contains migrations that set up the necessary tables.

Apply the migrations by running:

```
$ rake db:migrate
```

Examine the file **db/schema.rb** to see the result of the migrations. You'll see several new tables corresponding to various Payola models:

- payola_affiliates — Affiliate
- payola_coupons — Coupon
- payola_sales — Sale
- payola_stripe_webhooks — StripeWebhook
- payola_subscriptions — Subscription

The new models are contained within a Payola module.

Payola Initializer

We'll modify the initializer file to set the Stripe API keys.

Replace the file **config/initializers/payola.rb**:

```
Payola.configure do |config|
  config.secret_key = Rails.application.secrets.stripe_api_key
  config.publishable_key = Rails.application.secrets.stripe_publishable_key
end
```

We could hardcode the Stripe API key directly in the initializer file. However, that would expose the API key if we commit the file to Git and other people view the repository (for example, in a public GitHub repository). Instead of hardcoding the API key, you should set a Unix environment variable with the API key, so it is only present on your local machine. It is good practice to set all credentials in the **config/secrets.yml** file, so we'll just include a variable that refers to credentials we've set in the **config/secrets.yml** file.

JavaScript

Take a look at the file **app/assets/javascripts/application.js**:

```
//= require jquery
//= require payola
//= require jquery_ujs
//= require turbolinks
//= require bootstrap-sprockets
//= require_tree .
```

It now includes a reference to a directory of Payola JavaScript files that are contained in the Payola gem.

Routes

Take a look at the file **config/routes.rb**:

```
Rails.application.routes.draw do
  mount Payola::Engine => '/payola', as: :payola
  root to: 'visitors#index'
  get 'products/:id', to: 'products#show', :as => :products
  devise_for :users
  resources :users
end
```

It now contains a directive that mounts the Payola Rails engine. You can use the `rake routes` command to see the routes that have been added to the application:

```
$ rake routes
```

The application responds to many URL requests. Among the URLS are:

- /buy/:product_class/:permalink — payola/transactions#create
- /confirm/:guid — payola/transactions#show
- /status/:guid — payola/transactions#status
- /subscribe/:plan_class/:plan_id — payola/subscriptions#create
- /confirm_subscription/:guid — payola/subscriptions#show
- /subscription_status/:guid — payola/subscriptions#status
- /cancel_subscription/:guid — payola/subscriptions#destroy
- /change_plan/:guid — payola/subscriptions#change_plan
- /change_quantity/:guid — payola/subscriptions#change_quantity
- /update_card/:guid — payola/subscriptions#update_card
- /events — StripeEvent::Engine

The new routes process payment transactions, manage subscriptions, and respond to event messages sent by Stripe.

Application Layout

Rails will use the layout defined in the file **app/views/layouts/application.html.erb** as a default for rendering any page. See the article Rails Default Application Layout (http://railsapps.github.io/rails-default-application-layout.html) for an explanation of each of the elements in the application layout.

The Payola gem provides a view partial to load the JavaScript needed to obtain a Stripe token from the Stripe server. You can see the view partial (https://github.com/peterkeen/payola/blob/master/app/views/payola/transactions/_stripe_header.html.erb) in the Payola GitHub repository:

```
<script type="text/javascript" src="https://js.stripe.com/v2/"></script>
<script type="text/javascript">
  Stripe.setPublishableKey("<%= Payola.publishable_key %>");
</script>
```

The Payola `_stripe_header` partial obtains a Stripe JavaScript file from the Stripe content delivery network. It also sets the `stripe_publishable_key` which is set in the **config/initializers/payola.rb** file.

We want to add the Payola view partial to any page where it is needed. We can't add it to specific view templates directly, because the Stripe JavaScript must be added to the HTML `<head>` section. We'll modify the default application layout file so we can insert the Payola view partial as needed.

Modify the file **app/views/layouts/application.html.erb**:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= content_for?(:title) ? yield(:title) : "Rails Stripe Membership" %></title>
    <meta name="description" content="<%= content_for?(:description) ? yield(:description) : "Rails Str
ipe Membership" %>">
    <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
    <%= yield(:head) %>
  </head>
  <body>
    <header>
      <%= render 'layouts/navigation' %>
    </header>
    <main role="main">
       <%= render 'layouts/messages' %>
       <%= yield %>
    </main>
  </body>
</html>
```

We add the directive `<%= yield(:head) %>` so we can insert code in the HTML `<head>` section from any view template. The `<%= yield(:head) %>` directive must come *after* the `javascript_include_tag 'application'` helper. The **application.js** file will load jQuery, which must be loaded before our Payola view partial.

Subscription Plans

We're building an application for a subscription-based service. Most membership-based websites offer more than one membership level, or subscription plan, typically offered at differing price points. Stripe easily accommodates multiple subscription plans, but each must be configured in the Stripe dashboard.

Implementation

When we create a subscription, we must inform Stripe which subscription plan to use for the new customer. to do so, we'll add a class that models the subscription plans. The Payola gem expects to find an object that includes a `Payola::Plan` module when it is time to initiate a payment transaction. We'll create a Plan model to implement our subscription plans.

Each User object will have an associated Plan object. We'll use Active Record associations (http://guides.rubyonrails.org/association_basics.html) to define the relationship between the objects. In this case, we'll have a `has_many` relationship. The User model `belongs_to` the Plan model and the Plan `has_many` Users.

Configure Stripe for Subscription Plans

We must set up our subscription plans in Stripe.

We'll configure Stripe for three plans named "Silver", "Gold", and "Platinum" that will be billed monthly at rates of $9, $19, and $29. Once a customer is created and assigned a plan, Stripe will do all the work of initiating monthly billing and retrying the transaction when a credit card is declined or expires.

Go to the Stripe plan management page (https://manage.stripe.com/#plans) to create a subscription plan. Stripe offers documentation about creating a plan (https://stripe.com/docs/subscriptions) and additional detail about plans (https://stripe.com/docs/api#plans).

Look for the toggle switch "Live/Test" and set it to "Test." Click the button to "Create your first plan."

Create three different plans with the following values:

ID	Name	Amount	Interval
silver	Silver	9.00	monthly
gold	Gold	19.00	monthly
platinum	Platinum	29.00	monthly

"ID" is a unique string of your choice that is used to identify the plan when subscribing a customer. In our application, each plan will have an ID that corresponds to a role we'll create to manage access. "Name" is displayed on invoices and in the Stripe web interface. "Amount" is the subscription price in US dollars. "Interval" is the billing frequency. Optionally, you can specify a trial period (in days). If you include a trial period, the customer won't be billed for the first time until the trial period ends. If the customer cancels before the trial period is over, she'll never be billed at all.

Model for Subscription Plans

We'll create a Plan model. The attributes will be set with the same values we set in the Stripe dashboard:

- name — Silver, Gold, or Platinum
- stripe_id — silver, gold, or platinum
- interval — monthly
- amount — 9.00, 19.00, or 29.00

We'll use a generator to create a model and accompanying database migration:

```
$ rails generate model Plan name stripe_id interval amount:integer --no-test-framework
invoke  active_record
create  db/migrate/..._create_plans.rb
create  app/models/plan.rb
```

The name of the model is "Plan." You could name it "SubscriptionPlan" or "Level" if you prefer, but we'll use "Plan" for this tutorial. The attributes `name`, `stripe_id`, and `interval` will be strings by default. We specify that the attribute `amount` will be an integer (Stripe likes prices in cents, without any decimal points). We specify

`--no-test-framework` because we'll create tests later.

Association Between Plan and User Models

We'll use a generator to create a migration that sets up an association between the User model and Plan model:

```
$ rails generate migration AddPlanRefToUsers plan:references
```

The option `plan:references` will create a foreign key `plan_id` and an index to associate the User model and the Plan model.

Plan Model

Modify the file **app/models/plan.rb**:

```
class Plan < ActiveRecord::Base
  include Payola::Plan

  has_many :users
  validates :stripe_id, inclusion: { in: Plan.pluck('DISTINCT stripe_id'),
      message: "not a valid subscription plan" }

  def redirect_path(subscription)
    '/'
  end

end
```

Adding `has_many :users` allows us to make database queries to get a list of all users with a specific subscription plan. We won't use it in this implementation but it is there if you need it.

Our validation rule prevents the user from entering an invalid subscription plan. When we use a select menu on the sign-up page, only valid subscription plans are available. But this validation rule adds a check. The statement `Plan.pluck('DISTINCT stripe_id')` returns an array of valid plans. The validation forces the display of an error message if the submitted plan is not included in the array of valid subscription plans.

The method `redirect_path` sets the page that is displayed on completion of a successful payment transaction. The Payola gem JavaScript code uses this URL to redirect to a success page when a transaction is complete.

User Model

We'll need to modify the User model to add the corresponding `belongs_to :plan` association.

Replace the file **app/models/user.rb**:

```
class User < ActiveRecord::Base
  enum role: [:user, :vip, :admin]
  after_initialize :set_default_role, :if => :new_record?
  after_initialize :set_default_plan, :if => :new_record?
  # after_create :sign_up_for_mailing_list

  belongs_to :plan
  validates_associated :plan

  def set_default_role
    self.role ||= :user
  end

  def set_default_plan
    self.plan ||= Plan.last
  end

  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  def sign_up_for_mailing_list
    MailingListSignupJob.perform_later(self)
  end

  def subscribe
    mailchimp = Gibbon::API.new(Rails.application.secrets.mailchimp_api_key)
    result = mailchimp.lists.subscribe({
      :id => Rails.application.secrets.mailchimp_list_id,
      :email => {:email => self.email},
      :double_optin => false,
      :update_existing => true,
      :send_welcome => true
    })
    Rails.logger.info("Subscribed #{self.email} to MailChimp") if result
  end

end
```

The `after_create :sign_up_for_mailing_list` callback is commented out so you won't create mailing list subscriptions during the development process. You can restore it when you are ready to deploy to production.

We've added `belongs_to :plan` to associate the Plan model with the User model. This allows us to obtain values with statements such as `user.plan.price`. We've also added a validation requirement, `validates_associated :plan`.

It may seem counterintuitive that the User model belongs to the Plan model. After all, it seems a plan belongs to each User. But that's the weird world of database associations. In this case, each Plan has many Users.

We want every user to have a plan, even when the User has just been instantiated. That makes it possible to display a default value for the plan on a subscription form. We add an `after_initialize` callback that calls a method, `set_default_plan`. You can change the default plan, if you wish. We simply select the last plan that is found on the database.

Drop Name Attribute

Our rails-signup-download (https://github.com/RailsApps/rails-signup-download) starter application contains a `name` attribute in the User model. It serves to illustrate how to add an attribute to the Devise sign-up form. It is not needed in our application, so we'll remove it.

We'll use a generator to create a migration that removes the `name` field from the Users table:

```
$ rails generate migration RemoveNameFromUsers name
```

This will add a migration that drops the `name` field.

Apply Migrations

Now apply the migrations by running:

```
$ rake db:migrate
```

Examine the file **db/schema.rb** to see the result of the migrations.

You'll see the Users table contains the `plan_id` which is the foreign key that ties a plan record to each user record.

Seed Subscription Plans

We can use the Rails console to create subscription plans for testing our application:

```
$ rails console
Loading development environment (Rails 4.2)
>> s1 = Plan.new(name:'Silver',stripe_id:'silver',interval:'monthly',amount: 900)
=> #<Plan id: nil, name: "Silver", stripe_id: "silver", interval: "monthly", amount: 900, created_at: n
il, updated_at: nil>
>> s1.save
   (0.3ms)  begin transaction
  SQL (0.5ms)  INSERT INTO "subscription_plans" ("name", "stripe_id", "interval", "amount", "created_a
t", "updated_at") VALUES (?, ?, ?, ?, ?, ?)  [["name", "Silver"], ["stripe_id", "silver"], ["interval",
"monthly"], ["amount", 900], ["created_at", "2015-02-20 14:49:27.639268"], ["updated_at", "2015-02-20 1
4:49:27.639268"]]
   (9.2ms)  commit transaction
=> true
>> exit
```

When we reset the database, we'll lose the subscription plans and we'll need to enter them again. It's better to set up the subscription plans using the **db/seeds.rb** file, so we seed the database with the `rake db:seed` or `rake db:reset` commands. We're already using the **db/seeds.rb** file to create an administrator account. We could create the subscription plans directly in the **db/seeds.rb** file, but instead let's use the pattern we've used to create the administrator account.

Create a file **app/services/create_plan_service.rb**:

```
class CreatePlanService
  def call
    p1 = Plan.find_or_create_by!(name: 'Platinum') do |p|
      p.amount = 2900
      p.interval = 'month'
      p.stripe_id = 'platinum'
    end
    p2 = Plan.find_or_create_by!(name: 'Gold') do |p|
      p.amount = 1900
      p.interval = 'month'
      p.stripe_id = 'gold'
    end
    p3 = Plan.find_or_create_by!(name: 'Silver') do |p|
      p.amount = 900
      p.interval = 'month'
      p.stripe_id = 'silver'
    end
  end
end
```

Replace the **db/seeds.rb** file:

```
user = CreateAdminService.new.call
puts 'CREATED ADMIN USER: ' << user.email
CreatePlanService.new.call
puts 'CREATED PLANS'
```

Now seed the database by running:

```
$ rake db:seed
```

You can run `rake db:reset` if you need to delete the data and seed again.

Roles and Plans

Our application is already set up to support multiple user roles, such as ordinary user, VIP, or administrator. The Role-Based Authorization (https://tutorials.railsapps.org/tutorials/rails-devise-roles) tutorial describes how we can restrict access to content based on roles. The tutorial, and the accompanying “rails-devise-roles” starter application, shows how we use a feature of Active Record, Enum (http://api.rubyonrails.org/classes/ActiveRecord/Enum.html), to add roles to a User model.

We’ve created a Plan model for our subscription plans. We’ve configured Stripe for the subscription plans. Now we must configure our application to include roles for each subscription plan. This will allow us to restrict access to content based on the user’s subscription plan.

Modify the User Model

The User model contains an array of roles:

```
enum role: [:user, :vip, :admin]
```

We’ll modify this list to drop the VIP role and add silver, gold, and platinum roles corresponding to our three subscription plans.

Replace the file **app/models/user.rb**:

```
class User < ActiveRecord::Base
  enum role: [:user, :silver, :gold, :platinum, :admin]
  after_initialize :set_default_role, :if => :new_record?
  after_initialize :set_default_plan, :if => :new_record?
  # after_create :sign_up_for_mailing_list

  belongs_to :plan
  validates_associated :plan

  def set_default_role
    self.role ||= :user
  end

  def set_default_plan
    self.plan ||= Plan.last
  end

  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  def sign_up_for_mailing_list
    MailingListSignupJob.perform_later(self)
  end

  def subscribe
    mailchimp = Gibbon::API.new(Rails.application.secrets.mailchimp_api_key)
    result = mailchimp.lists.subscribe({
      :id => Rails.application.secrets.mailchimp_list_id,
      :email => {:email => self.email},
      :double_optin => false,
      :update_existing => true,
      :send_welcome => true
    })
    Rails.logger.info("Subscribed #{self.email} to MailChimp") if result
  end

end
```

We've simply dropped the VIP role and added three new roles. The default role will continue to be "user" but we'll design a Registrations controller so each user is assigned a role corresponding to a subscription plan when an account is created.

Content Pages

This application can be used for a Software-as-a-Service (SaaS) website or it can be used to limit access to pages containing content such as articles, photos, or video. For purposes of demonstration, we'll set up the site so a membership is required to view some placeholder content.

Let's add pages for our placeholder content.

Create the Content Controller and Views

Use the `rails generate` command to create a controller and associated views:

```
$ rails generate controller content silver gold platinum --skip-stylesheets --skip-javascripts
```

We've named the controller the "ContentController." The default route will put our content pages in an apparent "content" directory with the URL path http://localhost:3000/content/ (http://localhost:3000/content/). You could give the controller another name if you want a different URL path but it's easier to keep the same controller name and change the path in the **config/routes.rb** file (described below).

We've asked for three views, corresponding to the three subscription plans we'll offer. We'll use `--skip-stylesheets --skip-javascripts` to avoid cluttering our application with stylesheet and JavaScript files we don't need.

The Rails generator will create these files for you:

```
app/controllers/content_controller.rb
app/helpers/content_helper.rb
app/views/content/gold.html.erb
app/views/content/platinum.html.erb
app/views/content/silver.html.erb
spec/controllers/content_controller_spec.rb
```

It also modifies the **config/routes.rb** file to add three routes:

```
get "content/silver"
get "content/gold"
get "content/platinum"
```

If you want a different URL path, you could specify a different path like this: `get "articles/silver" => "content#silver", :as => :content_silver`. Visitors will see a URL path http://localhost:3000/articles/silver (http://localhost:3000/articles/silver) but you won't need to make any other changes to the application. We won't do this; we'll just use the supplied path.

If you look at **app/controllers/content_controller.rb** file, you'll see it is very simple:

```
class ContentController < ApplicationController

  def silver
  end

  def gold
  end

  def platinum
  end
end
```

It may be odd to see a controller that doesn't contain the familiar `index`, `show`, etc. methods of a RESTful controller. This is a case where a RESTful controller is not needed or appropriate. By default, the controller will render a view corresponding to each action.

Check the Content Views

Open each of the view files to see the placeholder content.

app/views/content/silver.html.erb

```
<h1>Content#silver</h1>
<p>Find me in app/views/content/silver.html.erb</p>
```

app/views/content/gold.html.erb

```
<h1>Content#gold</h1>
<p>Find me in app/views/content/gold.html.erb</p>
```

app/views/content/platinum.html.erb

```
<h1>Content#platinum</h1>
<p>Find me in app/views/content/platinum.html.erb</p>
```

If you're building a real application, you'll want to provide content that is more useful than our placeholders. For a membership site that delivers content such as ebooks or videos, you could use a structure such as this, where we'll restrict access to pages in a **content** directory based on the user's subscription plan. For a site that delivers software as a service, the structure of your application will necessarily be more complex.

Limit Access

We'll modify the Content controller to set access limits. Without access control, users can access any content if they guess at the correct URL.

Replace the **app/controllers/content_controller.rb** file:

```
class ContentController < ApplicationController
  before_action :authenticate_user!

  def silver
    redirect_to root_path, :notice => "Access denied." unless current_user.silver?
  end

  def gold
    redirect_to root_path, :notice => "Access denied." unless current_user.gold?
  end

  def platinum
    redirect_to root_path, :notice => "Access denied." unless current_user.platinum?
  end

end
```

We add `before_action :authenticate_user!` (provided by Devise) to force a visitor to log in before any action.

Access control is simple to implement. Our Role-Based Authorization (https://tutorials.railsapps.org/tutorials/rails-devise-roles) tutorial explains how Active Record Enum (http://api.rubyonrails.org/classes/ActiveRecord/Enum.html) gives us the methods we need. We redirect to the home page with an "Access denied" message if the current user does not have the correct role.

Registrations Controller

Before we implement anything else, we'll need a Registrations controller to handle the subscription transaction. We're piggybacking on top of the Devise registration feature. We want to use the ready-made Devise registration process to create an account. Then we'll initiate credit card payment after creating an account.

The rails-signup-download starter application doesn't have a Registrations controller that you can see. For the starter application, the Registrations controller is provided by the Devise gem. It does all the work of processing a sign-up form and creating a new user account. It is hidden in the gem. For this application, we need to override the Devise Registrations controller with our own Registrations controller that will initiate a credit card payment.

Before we modify our sign-up form to include fields for credit card data, we'll set up a new Registrations controller.

Registrations controller

Create a file **controllers/registrations_controller.rb**:

```
class RegistrationsController < Devise::RegistrationsController
  include Payola::StatusBehavior

  def create
    build_resource(sign_up_params)
    plan = Plan.find_by!(id: params[:user][:plan_id].to_i)
    resource.role = User.roles[plan.stripe_id] unless resource.admin?
    resource.save
    yield resource if block_given?
    if resource.persisted?
      if resource.active_for_authentication?
        set_flash_message :notice, :signed_up if is_flashing_format?
        sign_up(resource_name, resource)
        subscribe
      else
        set_flash_message :notice, :"signed_up_but_#{resource.inactive_message}" if is_flashing_format?
        expire_data_after_sign_in!
        subscribe
      end
    else
      clean_up_passwords resource
      render json:
        {error: resource.errors.full_messages.to_sentence},
        status: 400
    end
  end

  private

  def sign_up_params
    params.require(:user).permit(:email,
    :password, :password_confirmation, :plan_id)
  end

  def subscribe
    return if resource.admin?
    params[:plan] = current_user.plan
    subscription = Payola::CreateSubscription.call(params, current_user)
    render_payola_status(subscription)
  end

end
```

If we wanted a simple controller, we'd specify `class RegistrationsController < ApplicationController` . In this case, we want all the methods and behavior provided by the Devise Registrations controller, so we specify `class RegistrationsController < Devise::RegistrationsController` . If you look at the code for the Devise Registrations controller in the file app/controllers/devise/registrations_controller.rb (https://github.com/plataformatec/devise/blob/master/app/controllers/devise/registrations_controller.rb) in the GitHub repository, you'll see the RESTful controller actions `new` , `create` , `edit` , `update` , and `destroy` . The Devise implementation is complex and you don't need to fully understand it. It is sufficient to know that our new Registrations controller automatically includes all these methods by inheritance.

The Payola gem injects a `render_payola_status` method into the Registrations controller when we include the `Payola::StatusBehavior` module from the gem.

In the rails-signup-download starter application, there is a sign-up form in the file **app/views/devise/registrations/new.html.erb**. The `new` action in the Devise Registrations controller instantiates a `@user` instance variable and renders the form. We don't need a `new` action in our Registrations controller because it is already there, inherited from the Devise Registrations controller. If we wanted our own version of the `new` action, we could add it to our controller. But we can use the inherited action and modify the **app/views/devise/registrations/new.html.erb** file to suit our needs.

When the user submits the form, the `create` action in the Devise Registrations controller creates a user account. It'd be great if we could use the default Devise `create` action without changes, but Payola needs to render a response with appropriate messages after initiating the subscription payment. We use all the code from the default Devise `create` action, and substitute a call to a `subscribe` method for the default `respond_with resource` method that is called when an account is successfully created.

However, before we save the User object to the database and initiate a subscription payment, we obtain a Plan object using the `params[:user][:plan_id]` parameter provided by the sign-up form. Unless the new user is an administrator, we set the user's role to match the subscription plan. This will allow us to restrict access to content based on the plan and the user's assigned role. The role is saved along with all other provided parameters by the `resource.save` method.

To initiate a subscription payment, we add a private `subscribe` method that is called after an account is successfully created. The `subscribe` method returns without action if a new user is an administrator. We call the `Payola::CreateSubscription` service, providing the current user and submitted parameters as arguments. The `Payola::CreateSubscription` service returns a status message, and Payola will redirect to a page that displays a success message.

At this point, we encounter a particularly difficult aspect of integrating the Payola gem with Devise. Devise gracefully handles errors such as a mismatched password or missing email address by rendering the **app/views/devise/registrations/new.html.erb** page with error messages. However, the Payola gem inserts a JavaScript polling mechanism that freezes the page until a JSON object is received.

When an account is successfully created by Devise, and we call the `subscribe` method and complete a successful subscription transaction, the `render_payola_status` method will return a JSON object that includes a success flag and the JavaScript code will redirect to a page of our choice to display a success message. That's easy.

However, if we never create a Devise user account because of a mismatched password or other error, we never call the `subscribe` method and the JavaScript polling mechanism continues waiting for a JSON response. To accommodate the JavaScript polling mechanism that Payola inserts on the registration page, we have to render a JSON response when we have a Devise error.

Here's an outline of the standard Devise Registrations controller `create` action:

```
def create
  ...
  if resource.persisted?
      ...
    end
  else
    clean_up_passwords resource
    set_minimum_password_length
    respond_with resource
  end
end
```

Here's a `create` action that accommodates the Payola gem:

```
def create
  ...
  if resource.persisted?
      ...
    end
  else
    clean_up_passwords resource
    render json:
      {error: resource.errors.full_messages.to_sentence},
      status: 400
  end
end
```

When the User object cannot be saved (`resource.persisted?`), we render a JSON object with a `status: 400` that triggers an error handler in the JavaScript script. The JSON object contains the error messages generated by Devise. When we customize the **app/views/devise/registrations/new.html.erb** view, we'll add an element to display the error messages.

Routes

We've added a Registrations controller without changing our routes. We'll change the routes now.

Open the file **config/routes.rb**. Replace the contents with this:

```
Rails.application.routes.draw do
  mount Payola::Engine => '/payola', as: :payola
  root to: 'visitors#index'
  get 'products/:id', to: 'products#show', :as => :products
  devise_for :users, :controllers => { :registrations => 'registrations' }
  resources :users
end
```

The route `get 'products/:id'` is from the rails-signup-download starter application. It delivers the PDF file using the Products controller `show` action.

We've added a route to override the default Devise Registrations controller with our own Registrations controller:

```
devise_for :users, :controllers => { :registrations => 'registrations' }
```

The route for `resources :users` implements the administrator's access to a list of user accounts, as provided by the rails-signup-download starter application. And the route `root :to => 'visitors#index'` displays the home page.

We've got a new Registrations controller, including routing. Now we'll customize the **app/views/devise/registrations/new.html.erb** view.

Sign Up Page

We've installed the Payola gem, set up subscription plans, and added a Registrations Controller to initiate a payment transaction. Now we'll create a sign up page.

We are piggybacking on the Devise registration mechanism, so we'll use the existing Devise sign up page as a beginning point.

The TurboLinks Problem

Rails 4.0 introduced Turbolinks (https://github.com/rails/turbolinks/) which improves the apparent speed of Rails applications by loading the content of the HTML `<head>` tag only once, and replacing only the contents of the `<body>` tag when the user visits a new page.

We need to make sure that any link that opens the sign-up page avoids the Turbolinks link feature. You can disable Turbolinks for the entire application, if you wish. A simpler solution is to disable Turbolinks for the links that lead to the sign-up page.

Replace the file **app/views/layouts/_navigation_links.html.erb**:

```
<%# add navigation links to this file %>
<% if user_signed_in? %>
  <li><%= link_to 'Edit account', edit_user_registration_path %></li>
  <li><%= link_to 'Sign out', destroy_user_session_path, :method=>'delete' %></li>
<% else %>
  <li><%= link_to 'Sign in', new_user_session_path %></li>
  <li><%= link_to 'Sign up', new_user_registration_path, data: { no_turbolink: true } %></li>
<% end %>
<% if user_signed_in? %>
  <% if current_user.admin? %>
    <li><%= link_to 'Users', users_path %></li>
  <% end %>
<% end %>
```

We've added an attribute `data: { no_turbolink: true }` to the `link_to` helper that points to the sign-up page. This forces the new page to load completely, including the HTML `<head>` section, so the Payola gem JavaScript code will be included.

Sign Up Page

We need a form to submit to the Registrations controller `create` action. It must contain the visitor's email address, password and password confirmation, credit card information, and a subscription plan id. Additionally, the form must be configured for interaction with the Payola gem JavaScript code. The Payola gem JavaScript will intercept the credit card information, obtain a Stripe token from the Stripe server, and add a hidden field to the form for the Stripe token.

Replace the file **app/views/devise/registrations/new.html.erb**:

```
<% content_for :head do %>
  <%= render 'payola/transactions/stripe_header' %>
<% end %>
<div class="authform">
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html => { :ro
le => 'form',
        :class => 'payola-onestep-subscription-form',
        'data-payola-base-path' => payola_path,
        'data-payola-plan-type' => resource.plan.plan_class,
        'data-payola-plan-id' => resource.plan.id}) do |f| %>
    <h3>Sign up</h3>
    <div>
      <br />
      <span id="error_explanation" class="payola-payment-error"></span>
      <br />
      <br />
    </div>
    <div class="form-group">
      <%= f.label 'Subscription plan' %>
      <%= f.collection_select(:plan_id, Plan.all, :id, :name) %>
    </div>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control', data: { payola: 'email' }  %>
    </div>
    <div class="form-group">
      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password_confirmation %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= label_tag :card_number, "Credit Card Number" %>
      <%= text_field_tag :card_number, nil, name: nil, class: 'form-control', data: { stripe: 'number'
} %>
    </div>
    <div class="form-group">
      <%= label_tag :card_code, "Card Security Code" %>
      <%= text_field_tag :card_code, nil, name: nil, class: 'form-control', data: { stripe: 'cvc' } %>
    </div>
    <br />
    <div class="form-group">
      <%= label_tag :card_month, "Card Expiry" %>
      <%= select_month nil, { use_two_digit_numbers: true}, { name: nil, 'data-stripe': 'exp-month' }
%>
      <%= select_year nil, {start_year: Date.today.year, end_year: Date.today.year+10}, { name: nil, 'd
ata-stripe': 'exp-year' } %>
    </div>
    <%= f.submit 'Sign up', :class => 'button right' %>
  <% end %>
</div>
```

We've replaced the sign-up form provided by the rails-signup-download starter application. The form is wrapped in a div with the `class="authform"` attribute to provide some CSS styling. The `form_for` helper is identical to the default Devise form helper. Devise doesn't use `@user`. When using Devise, `resource` means `@user`. The variable `resource` is a proxy for a user object, allowing a developer to create a user model named Account, Profile, Member, or anything else (though User is most common).

The Payola gem's documentation for subscriptions (https://github.com/peterkeen/payola/wiki/Subscriptions) shows how a form should be configured for interaction with the Payola gem JavaScript code. A JavaScript function expects to find a form assigned the CSS class `payola-onestep-subscription-form`, so we assign the class to the form. We set additional data values that will be available to the Payola gem JavaScript code:

- `data-payola-base-path` is used to create URLs for Payola pages
- `data-payola-plan-type` is the name of the Plan class
- `data-payola-plan-id` will be the id of the subscription plan

We've added `<span id="error_explanation" class="payola-payment-error"></span>` to accommodate the error messages displayed by the error handler in the Payola gem JavaScript script. With the id `error_explanation`, we add the CSS styling that is used by the `devise_error_messages!` helper. This span replaces the `devise_error_messages!` helper that is used for the rails-signup-download starter application. In the previous chapter, we looked at the complications of integrating Devise and the Payola gem, and saw how the Payola error handler consumes a JSON response and displays error messages on the sign up form.

We've added a form element to set the subscription plan. We use the Rails form helper `collection_select` to create a drop-down menu of available subscription plans. The collection_select (http://apidock.com/rails/ActionView/Helpers/FormOptionsHelper/collection_select) helper takes several arguments:

- User attribute to submit: `plan_id`
- array of Plan objects: `Plan.all`
- Plan attribute to use for a submitted value: `id`
- Plan attribute to show in the menu: `name`

The visitor will see a list of Plan names, make a selection, and the Plan id will be submitted to become the `plan_id` attribute of the User object.

A drop-down menu is one of several approaches you can use to submit a `plan_id` to the Registrations controller `create` action. You might want to offer only a single subscription plan on the sign up page. If that's the case, submit the `plan_id` with a hidden field.

The form fields for `email`, `password`, and `password_confirmation` are the standard fields required by Devise.

We add credit card data fields for `card_number`, `card_code`, and the card expiration date. Notice the differences between the credit card data fields and the previous data fields. The email and password fields correspond to attributes of the User model. We use the Rails form builder object (the `f` variable) to bind the form to the User model. Methods such as `f.email_field` create form fields using the form builder object which automatically populate the attributes of the User model. The credit card data fields do not correspond to attributes of the User model so we can't use the form builder object. We use the more primitive `label_tag` and `text_field_tag` instead. Notice we specify the `name:` of the element as nil. Thus the fields will be available to our JavaScript code but will not be submitted to the server. Each field is given a `data-stripe` attribute that will be used by the Stripe JS script.

The submit button is typical for a Rails form. However, keep in mind that the Payola gem JavaScript intercepts the action of the submit button. The script will disable the submit button to prevent duplicate clicks and then obtain a Stripe token from the Stripe server. Then the script sets several hidden fields and submits the form to the Rails application. The page remains frozen by the script until the Rails application returns a JSON object containing either error messages or a success message. If the JSON object indicates success, the script will redirect to a designated success page. The location of the success page is defined by the `redirect_path` method in the Plan model. To understand the Payola JavaScript, you can look at the subscription_form_onestep.js (https://github.com/peterkeen/payola/blob/master/app/assets/javascripts/payola/subscription_form_onestep.js) script on GitHub.

Styling for Error Messages

The element `<span id="error_explanation" class="payola-payment-error"></span>` should be invisible if there are no error messages. If it contains errors, we'd like to apply a red background or other style to make it stand out. Examine the file **app/assets/stylesheets/framework_and_overrides.css.scss** and you'll see a CSS directive that applies styling for `id="error_explanation"`:

```
#error_explanation:not(:empty) {
  @extend .alert;
  @extend .alert-danger;
}
```

This is the CSS for use with the Bootstrap front-end framework. The pseudo-class `not(:empty)` makes sure the styling is only applied when the element is not empty.

Update the Edit View

Our rails-signup-download (https://github.com/RailsApps/rails-signup-download) starter application contains a `name` attribute in the User model. It serves to illustrate how to add an attribute to the Devise sign-up form. It is not needed in our application. We'll remove the field from our edit view.

Replace the file **app/views/devise/registrations/edit.html.erb**:

```
<div class="authform">
  <h3>Edit <%= resource_name.to_s.humanize %></h3>
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html => { :me
thod => :put, :role => 'form'}) do |f| %>
    <%= devise_error_messages! %>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>
      <% if devise_mapping.confirmable? && resource.pending_reconfirmation? %>
        <div>Currently waiting confirmation for: <%= resource.unconfirmed_email %></div>
      <% end %>
    </div>
    <fieldset>
      <p>Leave these fields blank if you don't want to change your password.</p>
      <div class="form-group">
        <%= f.label :password %>
        <%= f.password_field :password, :autocomplete => 'off', class: 'form-control' %>
      </div>
      <div class="form-group">
        <%= f.label :password_confirmation %>
        <%= f.password_field :password_confirmation, class: 'form-control' %>
      </div>
    </fieldset>
    <fieldset>
      <p>You must enter your current password to make changes.</p>
      <div class="form-group">
        <%= f.label :current_password %>
        <%= f.password_field :current_password, class: 'form-control' %>
      </div>
    </fieldset>
    <%= f.submit 'Update', :class => 'button right' %>
  <% end %>
</div>
<div class="authform">
<h3>Cancel Account</h3>
<p>Unhappy? We'll be sad to see you go.</p>
<%= button_to "Cancel my account", registration_path(resource_name), :data => { :confirm => "Are you su
re?" }, :method => :delete, :class => 'button right' %>
</div>
```

We've simply removed the `name` field from the form. We won't ask the visitor for a name in our application because it adds unnecessary complexity.

Cancel Subscription

We've completed integration with Stripe so we can sign up new subscribers. Stripe will automatically bill their credit cards every month. But what happens when a subscriber wants to cancel their subscription?

It's easy to handle a subscription cancellation request. Devise does the work of deleting the user account and we just piggyback on the Devise registration controller action with a `before_action` callback so we can notify Stripe to cancel subscription billing.

Replace the file **controllers/registrations_controller.rb**:

```ruby
class RegistrationsController < Devise::RegistrationsController
  include Payola::StatusBehavior
  before_action :cancel_subscription, only: [:destroy]

  def create
    build_resource(sign_up_params)
    resource.save
    yield resource if block_given?
    if resource.persisted?
      if resource.active_for_authentication?
        set_flash_message :notice, :signed_up if is_flashing_format?
        sign_up(resource_name, resource)
        subscribe
      else
        set_flash_message :notice, :"signed_up_but_#{resource.inactive_message}" if is_flashing_format?
        expire_data_after_sign_in!
        subscribe
      end
    else
      clean_up_passwords resource
      render json:
        {error: resource.errors.full_messages.to_sentence},
        status: 400
    end
  end

  private

  def sign_up_params
    params.require(:user).permit(:email,
    :password, :password_confirmation, :plan_id)
  end

  def subscribe
    return if resource.admin?
    params[:plan] = current_user.plan
    subscription = Payola::CreateSubscription.call(params, current_user)
    current_user.save
    render_payola_status(subscription)
  end

  def cancel_subscription
    subscription = Payola::Subscription.find_by!(email: current_user.email)
    Payola::CancelSubscription.call(subscription)
  end

end
```

The `before_action` callback calls a `cancel_subscription` method before the `destroy` action is executed.

The `cancel_subscription` method uses the user's email address to find a Payola Subscription object. The `Payola::Subscription` syntax indicates that the Subscription model is contained in a Payola module in the gem. You can see the Payola Subscription model (https://github.com/peterkeen/payola/blob/master/app/models/payola/subscription.rb) on GitHub.

Once the Subscription object is found, we execute the Payola `CancelSubscription` service (https://github.com/peterkeen/payola/blob/master/app/services/payola/cancel_subscription.rb). The Payola gem connects to Stripe and cancels the subscription.

We don't see the code for the Registrations controller `destroy` action because we inherit from the Devise Registrations controller. There is no need to override the default `destroy` action.

Change Plan

We've accommodated new subscriptions and cancellations. We also need to accommodate users who wish to change their subscription plans. Depending on your business needs, you may be upselling subscribers to a more expensive plan or allowing subscribers to downgrade plans.

We'll modify the Registrations controller and add a `change_plan` action. We'll need to add a new route for this action. We'll also need to add a form on our "Edit Account" page. We'll add a form to the page so the user can choose a different plan.

Modify the Registrations Controller

We'll start by adding the `change_plan` action to the Registrations controller.

Replace the file **controllers/registrations_controller.rb**:

```
class RegistrationsController < Devise::RegistrationsController
  include Payola::StatusBehavior
  before_action :cancel_subscription, only: [:destroy]

  def create
    build_resource(sign_up_params)
    plan = Plan.find_by!(id: params[:user][:plan_id].to_i)
    resource.role = User.roles[plan.stripe_id] unless resource.admin?
    resource.save
    yield resource if block_given?
    if resource.persisted?
      if resource.active_for_authentication?
        set_flash_message :notice, :signed_up if is_flashing_format?
        sign_up(resource_name, resource)
        subscribe
      else
        set_flash_message :notice, :"signed_up_but_#{resource.inactive_message}" if is_flashing_format?
        expire_data_after_sign_in!
        subscribe
      end
    else
      clean_up_passwords resource
      render json:
        {error: resource.errors.full_messages.to_sentence},
        status: 400
    end
  end

  def change_plan
    plan = Plan.find_by!(id: params[:user][:plan_id].to_i)
    unless plan == current_user.plan
      role = User.roles[plan.stripe_id]
      if current_user.update_attributes!(plan: plan, role: role)
        subscription = Payola::Subscription.find_by!(email: current_user.email)
        Payola::ChangeSubscriptionPlan.call(subscription, plan)
        redirect_to edit_user_registration_path, :notice => "Plan changed."
      else
        flash[:alert] = 'Unable to change plan.'
        build_resource
        render :edit
      end
    end
  end

  private

  def sign_up_params
    params.require(:user).permit(:email,
    :password, :password_confirmation, :plan_id)
  end

  def subscribe
    return if resource.admin?
    params[:plan] = current_user.plan
    subscription = Payola::CreateSubscription.call(params, current_user)
    current_user.save
    render_payola_status(subscription)
  end

  def cancel_subscription
    subscription = Payola::Subscription.find_by!(email: current_user.email)
    Payola::CancelSubscription.call(subscription)
  end

end
```

The `change_plan` action obtains a Plan object using a parameter submitted from a form. There's no attempt to change the plan if the submitted plan is the same as the current plan.

This application uses roles to manage access to content for each plan. The array `User.roles[]` contains a list of all roles. We attempt to set a new role that corresponds to the subscription plan.

We use `current_user.update_attributes!(plan: plan, role: role)` to update the User attributes for plan and role. If the update is successful, we use the user's email address to find a Payola Subscription object. The `Payola::Subscription` syntax indicates that the Subscription model is contained in a Payola module in the gem. You can see the Payola Subscription model (https://github.com/peterkeen/payola/blob/master/app/models/payola/subscription.rb) on GitHub.

Once the Subscription object is found, we execute the Payola::ChangeSubscriptionPlan service (https://github.com/peterkeen/payola/blob/master/app/services/payola/change_subscription_plan.rb). The Payola gem connects to Stripe and changes the subscription. Finally, we redirect back to the "Edit Account" page with a success message.

If we are unable to update the attributes, we render the "Edit Account" page with an error message. Notice that we must first use the Devise `build_resource` helper method to instantiate the User object before rendering the "Edit Account" page.

Add a “Change Plan” Route

We’ve added a `change_plan` action to the Registrations controller to accommodate the “Change Plan” features. We need to add a route to invoke the new action.

Modify the **config/routes.rb** file to add a route:

```
Rails.application.routes.draw do
  mount Payola::Engine => '/payola', as: :payola
  root to: 'visitors#index'
  get 'products/:id', to: 'products#show', :as => :products
  devise_for :users, :controllers => { :registrations => 'registrations' }
  devise_scope :user do
    put 'change_plan', :to => 'registrations#change_plan'
  end
  resources :users
end
```

The `devise_scope` method (http://rdoc.info/github/plataformatec/devise/ActionDispatch/Routing/Mapper:devise_scope) applies Devise.mappings (http://rubydoc.info/github/plataformatec/devise/master/Devise/Mapping) so the new routes are associated with other routes specified by `devise_for` (http://rubydoc.info/github/plataformatec/devise/master/ActionDispatch/Routing/Mapper#devise_for-instance_method). Our Registrations controller subclasses the Devise Registrations controller and it will raise an “Unknown action” error if the scope is not set by `devise_scope`. Note that `devise_for` takes a `:users` argument (plural) and `devise_scope` takes a `:user` argument (singular).

We add the `change_plan` route to respond to an HTTP “put” request (a form submission). This generates the route that we use in our “Change Plan” form:

- change_plan_path

Next we’ll add a “Change Plan” form to the “Edit Account” page.

Change the “Edit Account” Page

Our “Edit Account” page is very similar to the registration page. In its rudimentary form, it contains a simple form to change the user’s name, email address, or password. We will add an additional form with a “Change Plan” button.

Replace the file **app/views/devise/registrations/edit.html.erb**:

```
<% content_for :head do %>
  <%= render 'payola/transactions/stripe_header' %>
<% end %>
<div class="authform">
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html => { :ro
le => 'form',
        :class => 'payola-onestep-subscription-form',
        'data-payola-base-path' => payola_path,
        'data-payola-plan-type' => resource.plan.plan_class,
        'data-payola-plan-id' => resource.plan.id}) do |f| %>
    <h3>Sign up</h3>
    <div>
      <br />
      <span id="error_explanation" class="payola-payment-error"></span>
      <br />
      <br />
    </div>
    <div class="form-group">
      <%= f.label 'Subscription plan' %>
      <%= f.collection_select(:plan_id, Plan.all, :id, :name) %>
    </div>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control', data: { payola: 'email' }  %>
    </div>
    <div class="form-group">
      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password_confirmation %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= label_tag :card_number, "Credit Card Number" %>
      <%= text_field_tag :card_number, nil, name: nil, class: 'form-control', data: { stripe: 'number'
} %>
    </div>
    <div class="form-group">
      <%= label_tag :card_code, "Card Security Code" %>
      <%= text_field_tag :card_code, nil, name: nil, class: 'form-control', data: { stripe: 'cvc' } %>
    </div>
    <br />
    <div class="form-group">
      <%= label_tag :card_month, "Card Expiry" %>
      <%= select_month nil, { use_two_digit_numbers: true}, { name: nil, 'data-stripe': 'exp-month' }
%>
      <%= select_year nil, {start_year: Date.today.year, end_year: Date.today.year+10}, { name: nil, 'd
ata-stripe': 'exp-year' } %>
    </div>
    <%= f.submit 'Sign up', :class => 'button right' %>
  <% end %>
</div>
```

We've added code before the `<div class="authform">` that precedes the `<h3>Cancel my account</h3>` statement:

```
<div class="authform">
  <h3>Subscription Plan</h3>
  <%= form_for(resource, :as => resource_name, :url => change_plan_path, :html => { :method => :put, :r
ole  => 'form'}) do |f| %>
    <div class="form-group">
      <%= f.label 'Subscription plan' %>
      <%= f.collection_select(:plan_id, Plan.all, :id, :name) %>
    </div>
    <%= f.submit 'Change Plan', :class => 'button right' %>
  <% end %>
</div>
```

It looks as if we bind the form to an object (http://guides.rubyonrails.org/form_helpers.html#binding-a-form-to-an-object) named "resource"; in fact, we bind the form to the User object. Devise abstracts the User object and names it "resource" which makes it possible to use objects with other names (such as Account or Member) in similar implementations. "Binding the form to the object" means the values for the form fields will be set with the attributes stored in the database.

We use the same `collection_select` helper that we provided for the user to pick a plan on the sign-up page. The Rails form helper `collection_select` creates a drop-down menu of available subscription plans. The collection_select (http://apidock.com/rails/ActionView/Helpers/FormOptionsHelper/collection_select) helper takes several arguments:

- User attribute to submit: `plan_id`
- array of Plan objects: `Plan.all`
- Plan attribute to use for a submitted value: `id`
- Plan attribute to show in the menu: `name`

The visitor will see a list of Plan names, make a selection, and the Plan id will be submitted to become the `plan_id` attribute of the User object.

Home Page

We'll put our subscription offer and pricing plan on the home page. For this tutorial, it's simplest to show the offer and prices right on the home page. For a real application, you might describe your offer on the home page and show pricing on a separate page.

Replace the Home Page

Replace the contents of the file **app/views/visitors/index.html.erb**:

```
<% if user_signed_in? %>
  <% if current_user.admin? %>
    <h3>Admin</h3>
    <p><%= link_to 'User count:', users_path %> <%= User.count %></p>
  <% else %>
    <h3>You've signed up. Download a free book.</h3>
    <%= link_to 'Download PDF', products_path('product.pdf'), class: 'btn btn-success btn-large' %>
  <% end %>
<% else %>
  <div id="welcome" class="hero-unit span7">
    <h1>Membership Site</h1>
    <h3>Learn to build a successful subscription site.</h3>
  </div>
  <div class="row col-sm-12 plans">
    <div class="col-sm-2 well">
      <div class="plan"><h2>Silver</h2></div>
      <ul class="list-unstyled">
        <li>One lesson a month</li>
      </ul>
      <h3>$9/month</h3>
      <%= link_to 'Subscribe', new_user_registration_path(plan: 'silver'), :class => 'btn btn-primary'
%>
    </div>
    <div class="col-sm-2 well featured">
      <div class="plan featured-plan"><h2>Gold</h2></div>
      <ul class="list-unstyled">
        <li>Ten lessons a month</li>
      </ul>
      <h3>$19/month</h3>
      <%= link_to 'Subscribe', new_user_registration_path(plan: 'gold'), :class => 'btn btn-primary' %>
    </div>
    <div class="col-sm-2 well">
      <div class="plan"><h2>Platinum</h2></div>
      <ul class="list-unstyled">
        <li>Thirty lessons a month</li>
      </ul>
      <h3>$29/month</h3>
      <%= link_to 'Subscribe', new_user_registration_path(plan: 'platinum'), :class => 'btn btn-primar
y' %>
    </div>
  </div>
<% end %>
```

We apply CSS classes from Bootstrap to style the page. We’ll create a few additional CSS classes in the next step.

The page contains a “hero unit” for your key marketing message plus three boxes describing subscription plans.

Each box contains a link to the sign-up page. We set an attribute in the link so the parameter `plan` will be passed to the Registrations controller.

CSS for Subscription Plans

We’ll provide some rudimentary CSS rules to style the home page. We’re using Bootstrap so we’ll get an attractive design with only a few CSS rules.

We'll add CSS assets to style the home page.

Create a file **app/assets/stylesheets/pricing.css.scss**:

```
// home page
.plans{
  text-align: center;
}
.plan{
  background-color: #111575;
}
.plan.featured-plan{
  background-color: #CCAB00;
}
.plan h2{
  line-height: 100px;
  color: #fff;
}
```

This stylesheet gets added automatically to the asset pipeline because any files in the same folder as the **app/assets/stylesheets/application.css.scss** file are added by the `*= require_tree .` statement.

This CSS will provide the design elements that are commonly seen on a pricing page: boxes for each plan.

The design is adequate for our tutorial but you may want to improve it to be more effective. If you're not a designer, you may want to look at the Bootstrap themes available in the WrapBootstrap (https://wrapbootstrap.com/) marketplace.

Modify the Registrations Controller

Clicking a button on the home page will pass a `plan` parameter to the `new` action in the Registrations controller. We must modify the Registrations controller to consume the plan parameter.

Replace the file **controllers/registrations_controller.rb**:

```
class RegistrationsController < Devise::RegistrationsController
  include Payola::StatusBehavior
  before_action :cancel_subscription, only: [:destroy]

  def new
    build_resource({})
    unless params[:plan].nil?
      @plan = Plan.find_by!(stripe_id: params[:plan])
      resource.plan = @plan
    end
    yield resource if block_given?
    respond_with self.resource
  end

  def create
    build_resource(sign_up_params)
    plan = Plan.find_by!(id: params[:user][:plan_id].to_i)
    resource.role = User.roles[plan.stripe_id] unless resource.admin?
    resource.save
    yield resource if block_given?
    if resource.persisted?
```

```
      if resource.active_for_authentication?
        set_flash_message :notice, :signed_up if is_flashing_format?
        sign_up(resource_name, resource)
        subscribe
      else
        set_flash_message :notice, :"signed_up_but_#{resource.inactive_message}" if is_flashing_format?
        expire_data_after_sign_in!
        subscribe
      end
    else
      clean_up_passwords resource
      render json:
        {error: resource.errors.full_messages.to_sentence},
        status: 400
    end
  end

  def change_plan
    plan = Plan.find_by!(id: params[:user][:plan_id].to_i)
    unless plan == current_user.plan
      role = User.roles[plan.stripe_id]
      if current_user.update_attributes!(plan: plan, role: role)
        subscription = Payola::Subscription.find_by!(email: current_user.email)
        Payola::ChangeSubscriptionPlan.call(subscription, plan)
        redirect_to edit_user_registration_path, :notice => "Plan changed."
      else
        flash[:alert] = 'Unable to change plan.'
        build_resource
        render :edit
      end
    end
  end

  private

  def sign_up_params
    params.require(:user).permit(:email,
    :password, :password_confirmation, :plan_id)
  end

  def subscribe
    return if resource.admin?
    params[:plan] = current_user.plan
    subscription = Payola::CreateSubscription.call(params, current_user)
    current_user.save
    render_payola_status(subscription)
  end

  def cancel_subscription
    subscription = Payola::Subscription.find_by!(email: current_user.email)
    Payola::CancelSubscription.call(subscription)
  end

end
```

We've added a `new` action. We'll override the `new` action from the default Devise Registrations controller. You can compare the default Devise Registrations controller (https://github.com/plataformatec/devise/blob/master/app/controllers/devise/registrations_controller.rb) on GitHub. Devise provides a `build_resource` helper to initialize the User object. If the `plan` parameter exists, we find the corresponding Plan object and set the User object plan value. Devise requires a `yield resource` statement and then we render the view template.

If you look closely at the sign-up form after clicking a button on the home page, you'll see that the visitor's preferred plan is already selected in the drop-down menu for the subscription plan. The visitor can change the plan before submitting the sign-up form. If you want to eliminate the choice, you could replace the select box with a hidden field.

If you want a hidden field, replace the drop-down menu:

```
<div class="form-group">
  <%= f.label 'Subscription plan' %>
  <%= f.collection_select(:plan_id, Plan.all, :id, :name) %>
</div>
```

And use this instead:

```
<% if @plan.nil? %>
  <div class="form-group">
    <%= f.label 'Subscription plan' %>
    <%= f.collection_select(:plan_id, Plan.all, :id, :name) %>
  </div>
<% else %>
  <%= f.hidden_field :plan_id, :value => resource.plan.id %>
<% end %>
```

We check to see if the `@plan` instance variable is nil, and show the drop-down menu if it is nil. If the plan has already been selected by the visitor, we create a hidden field for the `plan_id` attribute and set the value with the plan id.

Redirect After Sign Up or Log In

As currently implemented, when a user signs up or logs in, they see the home page with a confirmation message. Instead, we want a user to be redirected to a page that is an appropriate hub for their role or subscription tier.

Modify the Application Controller

Devise provides a helper method `after_sign_in_path_for` that we can set for the behavior we want.

Modify the file **app/controllers/application_controller.rb**:

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception

  def after_sign_in_path_for(resource)
    case current_user.roles
      when 'admin'
        users_path
      when 'silver'
        content_silver_path
      when 'gold'
        content_gold_path
      when 'platinum'
        content_platinum_path
      else
        root_path
    end
  end

end
```

The `after_sign_in_path_for(resource)` method will redirect a user to an appropriate page after sign in or sign up. The `case` statement checks the current user's role and redirects to the appropriate content page, or if an administrator, to the administrative dashboard.

Devise also offers an `after_sign_up_path_for(resource)` method that allows a different redirect after a user registers. You could implement the `after_sign_up_path_for(resource)` method if you wanted the new user to see a special page after sign up (for example, a thank you or introduction).

Test the Application

You can check that the redirect works by entering the command:

```
$ rails server
```

Visit http://localhost:3000/ (http://localhost:3000) to see your home page.

Subscribe to any subscription plan. You should be redirected to the appropriate page and see a message "Welcome! You have signed up successfully." Log out and log in. You should be redirected to the appropriate page.

Webhooks

What happens when a credit card expires or a monthly transaction is declined? Stripe will automatically retry a recurring payment after it fails. After a number of attempts (set in your Stripe account settings), Stripe will cancel the subscription. But how will your application know to deny access for a subscriber with an expired account? Stripe provides webhooks (http://en.wikipedia.org/wiki/Webhook) (push notifications) to communicate events to you (for details, see the Stripe webhooks documentation (https://stripe.com/docs/webhooks)).

A Stripe webhook is an HTTP request from Stripe's servers to your site. It is not a visit to your website from a web browser; rather it is an HTTP POST request (like a form submission) to your application from the Stripe servers. The HTTP request contains JSON data that provides data about the event, plus an event id that can be used to retrieve the data from the Stripe server. Stripe advises that you should ignore the event data (because it could be falsified) and immediately use the event id to obtain the event data from Stripe.

There are a few ways we could handle webhook requests. We could create a new controller with an index action to process webhook requests. We could use the existing Registrations or Users controller, adding a new non-RESTful action to process the webhook requests. Or we could use Danny Whalen's stripe_event (https://github.com/integrallis/stripe_event) gem. With this gem, we don't need to add a new controller or action. We're using the Payola gem, however, and it offers greater functionality. It uses the stripe_event gem and integrates it within the Payola gem API.

Send an Expiration Email

We'll use an ActionMailer method to send an email when we receive a Stripe webhook request indicating a subscription has been cancelled for payment failure.

Generate a mailer with accompanying views:

```
$ rails generate mailer UserMailer
```

Add an `expire_email` method to the mailer by editing the file **app/mailers/user_mailer.rb**:

```
class UserMailer < ActionMailer::Base
  default :from => "do-not-reply@example.com"

  def expire_email(user)
    mail(:to => user.email, :subject => "Subscription Cancelled")
  end
end
```

Replace the "do-not-reply@example.com" string with your email address.

Create a mailer view by creating a file **app/views/user_mailer/expire_email.html.erb**. This will be the template used for the email, formatted in HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
  </head>
  <body>
    <h1>Subscription Cancelled</h1>
    <p>
      Your subscription has been cancelled.
    </p>
    <p>
      We are sorry to see you go. We'd love to have you back.
      Visit example.com anytime to create a new subscription.
    </p>
  </body>
</html>
```

It is a good idea to make a text-only version for this message. Create a file **app/views/user_mailer/expire_email.text.erb**:

```
Subscription Cancelled

Your subscription has been cancelled.

We are sorry to see you go. We'd love to have you back.
Visit example.com anytime to create a new subscription.
```

When you call the mailer method, ActionMailer will detect the two templates (text and HTML) and automatically generate a multipart/alternative email. If you use the Mandrill email service, you can skip this step if you configure Mandrill to automatically generate a plain-text version of all emails.

Now we send an email message when a Stripe webhook notifies the application of a cancelled subscription.

Modify the Payola Initializer

We need to specify what happens when the Payola engine receives a webhook request. Conveniently, we can specify this in the Payola initializer file. Payola webhooks are described on the Configuration Options (https://github.com/peterkeen/payola/wiki/Configuration-options) page of the Payola documentation.

Replace the **config/initializers/payola.rb** file:

```
Payola.configure do |config|
  config.secret_key = Rails.application.secrets.stripe_api_key
  config.publishable_key = Rails.application.secrets.stripe_publishable_key
  payola.subscribe 'customer.subscription.deleted' do |event|
    sale = Sale.find_by(stripe_id: event.data.object.id)
    user = User.find_by(email: sale.email)
    UserMailer.expire_email(user).deliver
    user.destroy
  end
end
```

By default, Stripe will make three attempts to rebill after a failed payment. On each failure, Stripe will send a webhook request with an `invoice.payment_failed` event. On the third failure, Stripe will send a `customer.subscription.deleted` event.

With this code, we're telling the Payola engine to respond when the application receives a webhook with a `customer.subscription.deleted` event. We use the event data to find the appropriate User object. We send our expiration email message. Then we call the `User.destroy` method.

The Payola engine handles other events as well. Refer to the Stripe API documentation (https://stripe.com/docs/api?lang=ruby#event_types) for a list of all event types. You can configure your application to respond to other events, such as sending a thank you email in response to an "invoice.payment_succeeded" event.

Remember you'll need to restart your server before testing because you've made a change to configuration files.

Set Your Webhook Address in Your Stripe Account Settings

After you've implemented handling of Stripe webhooks, you need to set your webhook address in your Stripe account settings.

Visit your Stripe dashboard at https://manage.stripe.com/#account/webhooks (https://manage.stripe.com/#account/webhooks) and add the URL https://www.example.com/payola/events (https://www.example.com/payola/events), replacing `example.com` with the domain where you've deployed the application.

Testing a Stripe Webhook Event

It's not easy to test a Stripe webhook event.

You can watch your development log file when you visit http://localhost:3000/payola/events (http://localhost:3000/payola/events). You should see:

```
Started GET "/payola/events" for 127.0.0.1 at ...
Processing by StripeEvent::WebhookController#event as HTML
Completed 401 Unauthorized in 0ms (ActiveRecord: 0.0ms)
```

The "401 Unauthorized" response indicates that the stripe_event gem received a request but was unable to retrieve a Stripe event from the Stripe servers.

You might notice a "Test Webhook" button on your Stripe dashboard. Stripe will send fake event to the URL you've specified in your account settings. However, your log file will again show a "401 Unauthorized" response because the "Test Webhook" button sends a fake event ("id"=>"evt_00000000000000") that can't be retrieved from Stripe.

The most expedient test is to deploy the application on a production web server and trigger an actual event on your Stripe dashboard in "Test" mode by creating a customer and then canceling a subscription. If you are not getting the results you expect, you can include the following debug code in the **config/initializers/payola.rb** file:

```
Payola.configure do |config|
  config.secret_key = Rails.application.secrets.stripe_api_key
  config.publishable_key = Rails.application.secrets.stripe_publishable_key
  payola.subscribe 'customer.subscription.deleted' do |event|
    Rails.logger.info event
  end
end
```

The debug code should show you the results of any request to the stripe_event gem in your log file.

You can use the RequestBin (http://requestb.in/) service to see what Stripe is sending. RequestBin lets you create a URL that will collect requests made to it, then lets you inspect the request headers and body.

If you need to test requests to the stripe_event gem on your local development machine, you can install the localtunnel gem (http://progrium.com/localtunnel/) (localtunnel on GitHub (https://github.com/progrium/localtunnel)) which will expose your local web server to the Internet so you can receive webhook requests from the Stripe servers.

Comments

Credits

Daniel Kehoe implemented the application and wrote the tutorial.

Did You Like the Tutorial?

Was this useful to you? Follow rails_apps (http://twitter.com/rails_apps) on Twitter and tweet some praise. I'd love to know you were helped out by the tutorial.

Any issues? Please create an issue (http://github.com/RailsApps/rails-stripe-membership-saas/issues) on GitHub. Reporting (and patching!) issues helps everyone.

36 Comments The RailsApps Project Login

Recommend Share Sort by Best

Join the discussion…

Chris Tudor · 8 months ago

Daniel,

One of the issues that people seem to be having is that their registrations.js file is not loading properly, resulting in various 'stripe_token not present' errors.

One of the causes of this /may/ be something which is slightly misleading in the tutorial. You say "If you haven't changed the default manifest file, the jquery.externalscript.js script will be automatically loaded by the//= require_tree . directive. If you've removed the //= require_tree . directive, specify the script in the manifest:

//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require jquery.readyselector
//= require jquery.externalscript

However, if people remove the //= require_tree directive and have exactly what you have above, wouldn't this result in registrations.js /not/ being loaded (given that you haven't 'required' it).

see more

Reply · Share ›

Chris Tudor · a year ago

Daniel,

Thanks for a great tutorial.

A quick note on the cancel_subscription action that you have in the user.rb file, which may need to be altered following a change to the Stripe API.

def cancel_subscription

```
unless customer_id.nil?
customer = Stripe::Customer.retrieve(customer_id)
unless customer.nil? or customer.respond_to?('deleted')
if customer.subscription.status == 'active'
customer.cancel_subscription
end
end
end
```

The Stripe API has since been updated such that:

---

see more

Reply · Share ›

Guest · a year ago

has anyone managed to get this up and running on Heroku and would like to get paid to build something similar for me? Must also incorporate the prelaunch setup tutorial- email me with quote thanks

Reply · Share ›

victor hazbun · a year ago

What webhook I could use to know that a trial period just expired?

Thank you.

Reply · Share ›

Paul · a year ago

The tutorial mentions in a couple of places that Stripe only accepts US and Canadian bank accounts. However, in September 2013 they went live in Ireland and according to the activation page they currently support 11 countries: Belgium, Canada, Finland, France, Germany, Ireland, Luxembourg, Netherlands, Spain, United Kingdom and United States.

So would be nice to update this.

Great tutorial by the way!

Reply · Share ›

John Hamman · a year ago

I am looking for someone to modify this and make it multi-tenant (apartment) and a few other small changes. Anyone available for hire?

Reply · Share ›

Browne Kester · a year ago

Can this tutorial be done with rails 4.0 and ruby 2.0.0-p247? I am having problems using RVM. After I install it and try to use it, I get rvm command not found.

Reply · Share ›

Daniel Kehoe RailsApps → Browne Kester · a year ago

This tutorial is written for Rails 3.2 and Ruby 2.0.0. I have plans to update it to Rails 4.0 but I'm tackling other tutorials first. I suggest to build it for Rails 3.2 and then make the adjustments required for Rails 4.0.

• Reply • Share ›

Albert Pak • a year ago

Got another "issue"/"mod". Anyone familiar with Devise Invitable gem?

What I'm looking to do is that when one user signs up, that user can invite another one - and when the user, that was invited, accepts and signs up, the invited user gets a role assigned, but I get stuck on the error described here

https://github.com/RailsApps/r...

Thank you in advance.

• Reply • Share ›

Albert Pak • a year ago

I'm having some issues with updating last_4_digits after user was on a "Free" account...any suggestions as to what can cause that field not update but it'll update in Stripe?

https://github.com/RailsApps/r...

• Reply • Share ›

Albert Pak Albert Pak • a year ago

Posted a "patch" for now unless someone has a better solution - feel free to comment :)

P.S. Pardon my code mess - just started learning RoR and loving it.

• Reply • Share ›

Guest • 2 years ago

I'm getting an error after adding CC form to my sign up form:

undefined method `stripe_token' for nil:NilClass

I've followed all the steps so far up to that point - tried looking it up on SO, found another outdated posting with no answers...

Has anyone come across this issue before?

Thank you in advance.

• Reply • Share ›

Daniel Kehoe RailsApps Guest • 2 years ago

Look for solutions to your issue in the closed GitHub issues for the example application: https://github.com/RailsApps/r...

• Reply • Share ›

Jasmine • 2 years ago

Will this be updated to Rails 4 anytime soon?

• Reply • Share ›

Daniel Kehoe RailsApps Jasmine • 2 years ago

My regrets but this app is the last on the list to be updated as it is the most complex of them all. I can't offer a timeframe at this time.

^ | ⌄ • Reply • Share ›

Eric Boehs • 2 years ago

For your display_base_errors in the Layout and Stylesheets section I simplified it to:

```
def display_base_errors resource
  content_tag :div, class: 'alert alert-error alert-block' do
    resource.errors[:base].map{|msg| content_tag(:p, msg)}.join.html_safe
  end if resource.errors[:base].any?
end
```

I removed the close button, because that just seems weird for an error (and it wasn't showing for me).

^ | ⌄ • Reply • Share ›

Shoff • 2 years ago

Great Tutorial. I'm trying to tweak the registration page a little, and was wondering if any one had some suggestions as to an approach. I need to turn the registration form into a two step process. Step 1 would be inputing username and password, and Step 2 would be adding billing information. Getting a little confused about the best away to do this.

^ | ⌄ • Reply • Share ›

Matt • 2 years ago

Brilliant tutorial guys. Would you be able to add a little bit on the end that deals with 'reseting' back to development after a deployment. I always have issues with the the compiled '/applications.css staying and so overwriting any subsequent changes in the css whilst back in local dev. Thanks.

^ | ⌄ • Reply • Share ›

Daniel Kehoe RailsApps ➦ Matt • 2 years ago

Glad you like the tutorial. Not sure about the issue you report. Sounds like something you could ask about on Stack Overflow.

^ | ⌄ • Reply • Share ›

Matt ➦ Daniel Kehoe • 2 years ago

Thanks for your quick reply Daniel. I have trawled SO for answers but with very little luck.

In essence, the issue that after doing a rake assets:precompile there is then an application.css file within the public/assets folder. When going back to development locally I cannot seem to get rid of this. I have tried rake assets:clean, deleted the public/assets folder manually, and others.

My point on having it as an addition to the tutorial is that currently it takes us from zero to Heroku deployment but it doesn't take us back to working locally again should further dev be needed.

Thanks again - the tutorial has helped me get a site live and deployed in no time at all!

^ | ⌄ • Reply • Share ›

Daniel Kehoe RailsApps ➦ Matt • 2 years ago

There should be no need to get rid of the generated public/assets contents as they are ignored during development. If not, maybe you've got a local configuration issue? I haven't encountered this before.

Reply • Share ›

Matt Daniel Kehoe • 2 years ago

It appears it was a caching issue and a 'rake tmp:cache' solved matters. Thanks for your help.

Reply • Share ›

John Hamman • 2 years ago

has anyone successfully gotten the Apartment gem to work with this. Particularly with Postgres?

Reply • Share ›

John Hamman • 2 years ago

I am not sure how can we pass the "StripeID" instead of the Stripe Package Name. For example, I want my silver package to be named "Silver" with the id "silver01". Where do I put that in the app? or what change do I need to make to store that id AND the name on the app side?

Reply • Share ›

Daniel Kehoe RailsApps John Hamman • 2 years ago

You can search throughout the app for "silver" and see where the plan name is used. The plan name is passed to Stripe in the User model "update_stripe" method. You could do some string manipulation on "roles.first.name".

Reply • Share ›

kevindewalt • 2 years ago

I'm not sure this is an 'issue' for GitHub, but wrt the update_stripe method of choice:

"This approach allows us to either create a new Stripe customer on create or update the Stripe customer record on update."

But it also calls this method every time a user logs in and out since timestamps are updated - conceptually weird and brittle IMHO.

Reply • Share ›

Daniel Kehoe RailsApps kevindewalt • 2 years ago

It's already open as a GitHub issue:
https://github.com/RailsApps/r...

To reduce the unneeded calls to Stripe, you could remove the :trackable parameter in the User model. There may be a better way to address this; feel free to add to the GitHub issue discussion.

Reply • Share ›

Dave Nash • 2 years ago

Thanks for the great tutorial, which I've got working as it is. However, my app is a marketplace that needs to have different kinds of users--customers and vendors, which are free and paid, respectively. Within vendors, there is a free plan as well. Can you recommend the best practice for making the Stripe integration conditional based on the plan?
Thanks again.

Reply • Share ›

Daniel Kehoe RailsApps Dave Nash • 2 years ago

Glad to hear the tutorial is serving you. It's a basic implementation and I can't help with customization but maybe another user will have advice.

Reply • Share ›

kevindewalt • 2 years ago

Any reason why we cannot also make the :name a virtual attribute on the credit card? I guess I'm wondering why it needs to be tied to the user model.

Also, with your solution isn't it possible that someone could be using a credit card with a different name? IOW he or she might use a nickname for the site - you don't specify "Name on Credit Card" or something like that.

Thanks, great stuff, saving me a ton of time.

Reply • Share ›

Seve Ballesteros • 2 years ago

It's not clear from your tutorial why you're retrieving the Stripe customer in the code below from the update_stripe method. Is it to process a new transaction? Shouldn't there be some code that creates a new payment? or are you retrieving the customer for some other reason?

customer = Stripe::Customer.retrieve(customer_id)
if stripe_token.present?
customer.card = stripe_token
end
customer.email = email
customer.description = name
customer.save

Reply • Share ›

Daniel Kehoe RailsApps → Seve Ballesteros • 2 years ago

In the User model, the update_stripe method is called every time the User is saved (triggered by the before_save filter). So we can create a new Stripe customer when we create a new user or update the Stripe customer record on update. The code you are looking at is called when a customer ID exists (on updates) so we retrieve the existing Customer object from Stripe and apply a Stripe token (if a payment was made) and reset the email and name (in case they changed). All the code that creates a new payment is handled in JavaScript from the form submission so all we ever have is a Stripe token after the payment is initiated.

Reply • Share ›

The Leasider • 2 years ago

I'm able to get the Railscast example to process transactions, but not yours. In particular, this check for the stripe_token keeps telling me the token is not present.
if customer_id.nil?
if !stripe_token.present?
raise "Stripe token not present. Can't create account."
end

Reply • Share ›

Daniel Kehoe RailsApps → The Leasider • 2 years ago

Did you resolve your issue? Please open a GitHub issue and I'll try to help.

• Reply • Share ›

The Leasider The Leasider • 2 years ago

Even stranger, in the handle response method, I'm getting a 200 response and the alert is telling me that there's a token. However, setting that value is not working

```
$('#promember_stripe_token').val(response.id)
$('#new_promember')[0].submit()
return alert('Stripe response blah: ' + response.id);
```

• Reply • Share ›

sebastian • 2 years ago

Thank you very much for this awesome tutorial! Could you make an extra tutorial for "paymill"?

• Reply • Share ›