RailsApps Project

Membership Site with Stripe

Ruby on Rails tutorial for Rails recurring billing using Stripe. Use for a Rails membership site, sub site, or SaaS site (software-as-a-service).

Contents

Chapter 1

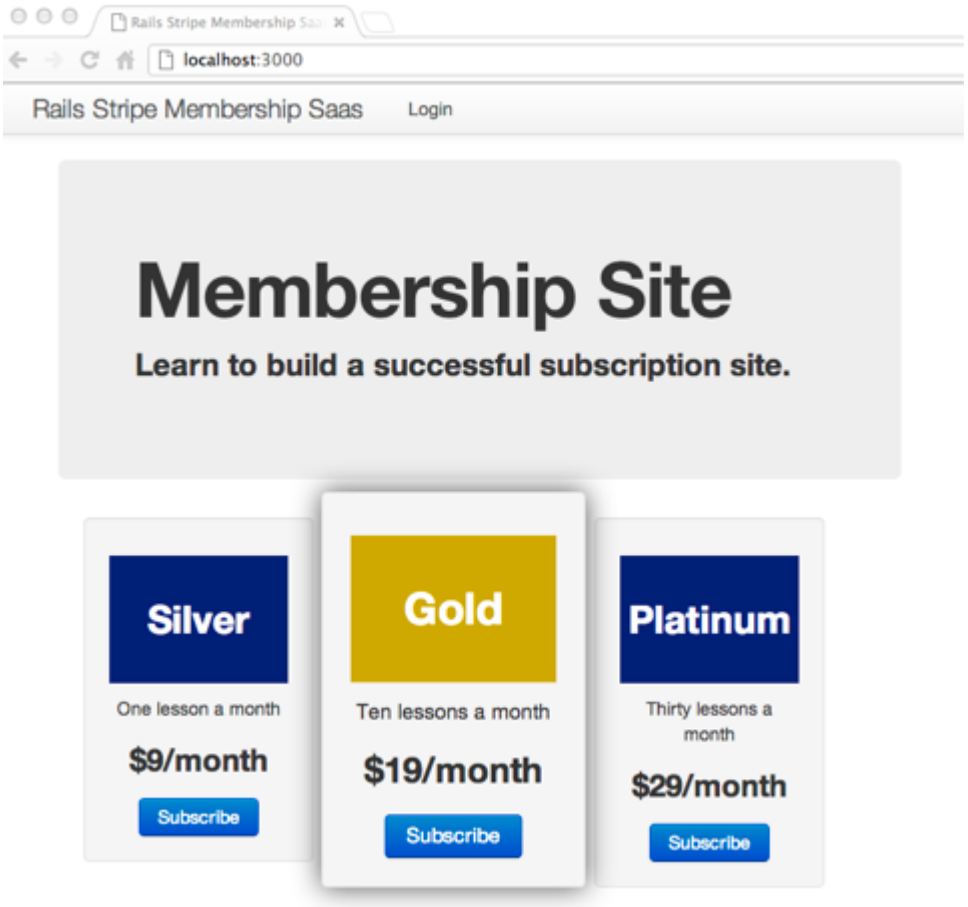
Introduction

Ruby on Rails tutorial for a web application with recurring billing using Stripe. Use for a Rails membership site, subscription site, or SaaS site (software-as-a-service).

- Stripe for recurring billing
- Devise for user management and authentication
- CanCan with Rolify for authorization
- Twitter Bootstrap front-end framework for CSS styling

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.

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 is for experienced developers as well as startup founders or hobbyist coders who are new to Rails.

Experienced developers will find the complete application on GitHub; this tutorial provides the detail and background to understand the implementation in depth. For Rails beginners, this tutorial describes each step that you must follow to create the application. Every step is documented concisely, so you can create this application without any additional knowledge. However, the tutorial assumes you've already been introduced to Rails, so if you are a beginner, you may be overwhelmed unless you've been introduced to Rails elsewhere. If

you're new to Rails, see recommendations for a Rails tutorial and resources for getting started with Rails.

This is one in a series of Rails example apps and tutorials from the RailsApps Project. See a list of similar Rails examples, tutorials, and starter apps.

This application is based on two simpler example apps:

- rails3-devise-rspec-cucumber
- rails3-bootstrap-devise-cancan

The first example shows how to set up Devise for user authentication. It also shows how to set up the app to use RSpec and Cucumber for testing.

The second example shows how to set up Devise and add CanCan and Rolify to manage access to administrative pages. It also shows how to set up Twitter Bootstrap as a front-end framework for CSS styling.

You can use this tutorial without studying these example applications; if you find you are lost, it may be helpful to look at the two simpler examples.

If you want to use the MongoDB datastore instead of ActiveRecord and a SQL database, look at the rails3-mongoid-devise example.

You might also be interested in the rails-prelaunch-signup example if you are planning prelaunch promotion. Use it to announce your plans and collect email addresses from visitors for future notification of the site's launch.

How to Support the Project

If you haven't subscribed to the RailsApps Tutorials Pro Plan, please consider purchasing a monthly subscription to support the project.

The code for the example applications is open source and unrestricted; your purchase of a subscription plan supports maintenance of the project. Rails applications need frequent updates as Rails and its gems change. Subscription sales support the project so we can keep the applications current so you'll always have an up-to-date reference implementation.

Functionality

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
- optional "free trial" subscription as well as free accounts using Stripe
- 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)
- administrator can change subscription plan or delete user

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 database segmentation

These features are not included in this application. See the rails3-subdomains example application for help with subdomains. For multitenancy, try Brad Robertson's Apartment gem.

About the Gems

RubyGems is a package manager for the Ruby programming language that provides a standard format for distributing Ruby programs and libraries (in a self-contained format called a "gem"). Gems add functionality to a Rails app.

We use these gems:

- Devise for user management and authentication
- CanCan with Rolify for authorization
- Stripe for recurring billing

Devise

Devise provides authentication, a system to securely identify users, making sure the user is who he represents himself to be.

We use Devise because it offers a full set of features used in more complex applications, such as recovering a user's forgotten password or allowing users to invite friends. Should you need help in troubleshooting or customizing the implementation, you'll be able to get help from a large community of developers using Devise.

CanCan with Rolify

CanCan provides a system for authorization to determine if an authenticated user should have access to secured resources. CanCan is often used to restrict access to administrative pages. This application will use CanCan to restrict access to content based on the price a user has paid for a subscription.

CanCan provides a mechanism for limiting access at the level of controller and controller method and expects you to set permissions based on user attributes you define. CanCan doesn't provide default user attributes such as user roles based on subscription price; you must implement this outside of CanCan. There are many ways to implement role-based authorization for use with CanCan. For this example, we use Florent Monbillard's Rolify gem to create a Role model, add methods to a User model, and generate a migration for a roles table.

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). Unfortunately, Stripe is only available to developers who have bank accounts in the US or Canada, though you can receive payments from customers internationally.

Stripe is the best choice for low-volume businesses based in the US or Canada. You may want to also look at the Recurly billing service. The RailsApps project provides the rails-recurly-subscription-saas application if you'd prefer to use Recurly. Unlike Stripe, Recurly requires you to obtain a merchant bank account from a third party which requires an additional step and a few days lead time. Recurly charges a monthly minimum fee of $69 per month which makes the per-transaction cost higher than Stripe for low-volume businesses. Above a threshold of about 600 monthly users (at an average transaction of $10), Recurly will be less expensive than Stripe (see the BillingSavvy calculator). Unlike Stripe, Recurly offers dunning management (sending emails to users with expired credit cards) that can increase customer retention. Stripe is available to businesses with US and Canadian bank accounts only; Recurly is available to businesses in many more countries.

Database

The tutorial shows how to set up the application using a SQLite database. Rails uses the SQLite database by default. Mac OS X come with SQLite pre-installed and there's nothing to configure. On Windows, if you use a pre-assembled package to install Rails, it will likely have SQLite pre-installed. On Ubuntu Linux, you can easily install SQLite. If you prefer to use MySQL or PostgreSQL, it's easy to change the application Gemfile and no changes are required for the application.

Chapter 2

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 (log in and log out)
- authorization management with CanCan and Rolify (access determined by the subscription plan)
- account management to maintain records of subscription status
- recurring billing with Stripe
- landing pages
- content or service pages

User Management, Authentication, and Authorization

The tutorial for the rails3-bootstrap-devise-cancan example application shows how to set up user management and authentication using Devise, as well as authorization management using CanCan and Rolify. The first step in the tutorial will be to generate the rails3-bootstrap-devise-cancan application as a starter app.

Users are managed with the User model, which has attributes for name, 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.

CanCan uses an Ability model to set access control rules. We'll modify the Ability model to set access rules based on subscription plans. We'll use the Role model required by Rolify to define roles based on subscription plans.

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.

We'll use the Stripe API to create a new customer and specify a subscription plan. When the customer's subscription expires due to failed payment, we'll use Stripe "webhooks" to update our application's user records.

We'll provide options for the user to change credit cards, 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 we can 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 to minimize risk of customer credit card exposure. Using Stripe, your server will never receive sensitive credit card details. Instead you'll use a the Stripe JavaScript library on your 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, credit card number, 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. You could make "Subscription" or "Plan" an object associated with a user. To keep this application simple, we'll define everything we need as attributes of the user, rather than separate objects.

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. The Stripe customer id will be an attribute of the user that substitutes for a credit card number.

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. We'll use the Role model supplied by the Rolify gem as an object that corresponds to a "Subscription Plan." Each user will have a role id that describes the subscription plan (or access level) that he or she has purchased.

Chapter 3

Accounts You May Need

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

Billing

Many providers of billing services want your business:

- Stripe (2011)
- SaaSy (2011)
- Fusebill (2011)
- Recurly (2010)
- SubscriptionBridge (2010)
- Chargify (2009)
- CheddarGetter (2009)
- Braintree (2007)
- Spreedly (2007)
- Zuora (2007)
- Adyen (2006)
- Vindicia (2003)
- Aria Systems (2003)

The list shows the year each service was founded. In general, since the market is highly competetive, the newer services are less expensive and offer better integration, interfaces, and features.

Several blog posts compare services and pricing:

- Peeling the onion called recurring billing (July 2012)
- Comparing Recurring Payment Solutions (March 2011)

You can use a web-based calculator to compare pricing of some services:

- BillingSavvy

Looking specifically at Recurly and Stripe, it appears that costs for Recurly or Stripe are similar for low-volume web businesses. Costs for either service are within $20 per month of each other for websites generating revenue of $4,000 to $20,000 per month with volume of 200 to 1000 transactions at an average price of $20 per transaction. If you anticipate revenue greater than $20,000/month, especially with a high average transaction price or a large number of transactions, you should carefully analyze the costs before deciding to choose either Stripe or Recurly.

Looking at other factors, Recurly is available to businesses located outside of the US and Canada (Stripe is not). And Recurly claims to offer better features for customer retention. Recurly's dunning managagement feature sends emails to the customer when the customer's credit card expires to encourage continuing the subscription. You'll need to implement this feature yourself with Stripe.

The RailsApps project provides example applications for either Stripe or Recurly.

This tutorial shows to set up recurring billing using Stripe. Before you start, go to the Stripe website 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.

Merchant Account

Your business will need a merchant account in order to accept credit card payments. Here's an explanation from Phillip Parker of 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."

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

Hosting

For easy deployment, use a "platform as a service" provider such as:

- Heroku
- CloudFoundry
- EngineYard

- OpenShift

Instructions are provided for deployment to Heroku.

It's common for technically skilled people to want to set up their own servers. Please, do yourself a favor, and unless system administration is your most dearly loved recreation, let the platform providers do it for you.

SSL

Visitors to your website will be sending 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 regsitration 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 webapp 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 for a fee of $20/month. You'll need to purchase a signed certificate from a certificate provider 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 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. 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 Cloudfare with their web browsers, providing a complete SSL connection through Cloudfare 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 Cloudfare 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 Cloudfare account.

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 for suggestions for various types of email service providers.

Domain Registration

You’ve likely already selected and registered a domain name. If not, you’ll need a domain before you start sending email messages from the application. If you’re disgusted by GoDaddy, consider NameCheap and other popular alternatives.

GitHub

Get a free GitHub account if you don’t already have one. You’ll use git for version control and you should get a GitHub account for remote backup and collaboration. See GitHub and Rails if you need more information about working with git and GitHub for code source control.

Chapter 4

Getting Started

Before You Start

Most of the tutorials from the RailsApps project take about an hour to complete. This tutorial is more complex; it will take you about three hours to build the complete app.

If you follow this tutorial closely, you'll have a working application that closely matches the example app in this GitHub repository. The example app in the rails-stripe-membership-saas repository is your reference implementation. If you find problems with the app you build from this tutorial, download the example app (in Git speak, clone it) and use a file compare tool to identify differences that may be causing errors. On a Mac, good file compare tools are FileMerge, DiffMerge, Kaleidoscope, or Ian Baird's Changes.

If you find problems or wish to suggest improvements, please create a GitHub issue. It's best to clone and check the example application from the GitHub repository before you report an issue, just to make sure the error isn't a result of your own mistake.

The online edition of this tutorial contains a comments section at the end of the tutorial. I encourage you to offer feedback to improve this tutorial.

Assumptions

Before beginning this tutorial, you need to install

- The Ruby language (version 1.9.3)
- Rails 3.2

Check that appropriate versions of Ruby and Rails are installed in your development environment:

```
$ ruby -v
$ rails -v
```

Be sure to read Installing Rails to make sure your development environment is set up properly.

I recommend using rvm, the Ruby Version Manager to manage your Rails versions and create a dedicated gemset for each application you build.

Chapter 5

Create the Application

You have several options for getting the code. You can *copy from the tutorial*, *fork*, *clone*, or *generate*.

If you want to add this code to an existing application, you can follow the tutorial and cut and paste the code into your existing application, resolving any conflicts as needed.

Copy from the Tutorial

To create the application, you can cut and paste the code from the tutorial into your own files. It's a bit tedious and error-prone but you'll have a good opportunity to examine the code closely. Before you start, the tutorial will ask you to use the Rails Composer tool to generate a starter app to save some steps. Then you can follow the tutorial step-by-step to build the complete application.

Other Options

Fork

If you'd like to add features (or bug fixes) to improve the example application, you can fork the GitHub repo and make pull requests. Your code contributions are welcome!

Clone

If you want to copy and customize the app with changes that are only useful for your own project, you can download or clone the GitHub repo. You'll need to search-and-replace the project name throughout the application. You probably should generate the app instead (see below). To clone:

```
$ git clone git://github.com/RailsApps/rails-stripe-membership-saas.git
```

You'll need git on your machine. See Rails and Git.

Generate

If you wish to skip the tutorial and build the application immediately, use the Rails Composer tool to generate the complete example app. You'll be able to give it your own project name when you generate the app. Generating the application gives you additional options.

To build the complete example application immediately, see the instructions in the README for the rails-stripe-membership-saas example application.

Building from Scratch

Before you write any code, you'll start by generating a starter app using the Rails Composer tool.

If you've developed other applications in Rails, you'll know that the `rails new` command creates a basic Rails application. Here we'll use the Rails Composer tool ("like the 'rails new' command on steroids") to create a starter app. The starter app saves us some steps. Devise will be installed with Cancan for authorization. Twitter Bootstrap will be set up as a front end for CSS styling. If you want to learn how the starter app is put together, see the rails3-bootstrap-devise-cancan tutorial.

For the starter app we need, use the command:

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

Use the `-T` flag to skip Test::Unit files since we'll be using RSpec.

The `$` character indicates a shell prompt; don't include it when you run the command.

This creates a new Rails app named `rails-stripe-membership-saas` on your computer. You can use a different name if you wish.

You'll see a prompt:

```
question  Install an example application?
      1)  I want to build my own application
      2)  membership/subscription/saas
      3)  rails-prelaunch-signup
      4)  rails3-bootstrap-devise-cancan
      5)  rails3-devise-rspec-cucumber
      6)  rails3-mongoid-devise
      7)  rails3-mongoid-omniauth
      8)  rails3-subdomains
```

Choose **rails3-bootstrap-devise-cancan**. The Rails Composer tool may give you other options (other choices may have been added since this tutorial was written). **Note:** Don't choose "membership/subscription/saas" (unless you want to skip the tutorial).

The application generator template will ask you for additional preferences:

```
question  Web server for development?
      1)  WEBrick (default)
      2)  Thin
      3)  Unicorn
      4)  Puma
question  Web server for production?
      1)  Same as development
      2)  Thin
      3)  Unicorn
      4)  Puma
question  Template engine?
      1)  ERB
      2)  Haml
      3)  Slim
  extras  Set a robots.txt file to ban spiders? (y/n)
  extras  Use or create a project-specific rvm gemset? (y/n)
  extras  Create a GitHub repository? (y/n)
```

Web Servers

Use the default WEBrick server for convenience. If you plan to deploy to Heroku, select "thin" as your production webserver.

Template Engine

The example application uses the default "ERB" Rails template engine. Optionally, you can use another template engine, such as Haml or Slim. See instructions for Haml and Rails.

Other Choices

Set a robots.txt file to ban spiders if you want to keep your new site out of Google search results.

It is a good idea to use rvm, the Ruby Version Manager, and create a project-specific rvm gemset (not available on Windows). See Installing Rails.

If you choose to create a GitHub repository, the generator will prompt you for a GitHub username and password.

Troubleshooting

If you get an error "OpenSSL certificate verify failed" or "Gem::RemoteFetcher::FetchError: SSL_connect" see the article OpenSSL errors and Rails.

If you get an error like this:

```
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem is
installed.
    composer  Running 'after bundler' callbacks.
The template [...] could not be loaded.
Error: You have already activated ..., but your Gemfile requires ....
Using bundle exec may solve this.
```

It's due to conflicting gem versions. See the article Rails Error: "You have already activated (…)".

Begin Development

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

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

Replace the READMEs

Please edit the README files to add a description of the app and your contact info. Changing the README is important if your app will be publicly visible on GitHub. Otherwise, people will think I am the author of your app. If you like, add an acknowledgment and a link to the RailsApps project.

Set Up Source Control (Git)

When you generate the starter app, the template sets up a source control repository and makes an initial commit of the code.

At your request, the template will also create a GitHub repository for your project.

See detailed instructions for Git and Rails.

Git has already been initialized by the application template script. If you've selected the GitHub option, the template commits your code to your GitHub repository.

Set Up Gems

The Rails Composer program sets up your Gemfile and (if you are using rvm) creates a project-specific gemset.

Open your **Gemfile** and you should see the following. Gem version numbers may differ:

```
source 'https://rubygems.org'
gem 'rails', '3.2.13'
gem 'sqlite3'
group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.1'
  gem 'uglifier', '>= 1.0.3'
end
gem 'jquery-rails'
gem "rspec-rails", ">= 2.11.0", :group => [:development, :test]
gem "email_spec", ">= 1.2.1", :group => :test
gem "cucumber-rails", ">= 1.3.0", :group => :test, :require => false
gem "database_cleaner", ">= 0.9.1", :group => :test
gem "launchy", ">= 2.1.2", :group => :test
gem "capybara", ">= 1.1.2", :group => :test
gem "factory_girl_rails", ">= 4.1.0", :group => [:development, :test]
gem "bootstrap-sass", ">= 2.1.0.0"
gem "devise", ">= 2.1.2"
gem "cancan", ">= 1.6.8"
gem "rolify", ">= 3.2.0"
gem "simple_form", ">= 2.0.4"
gem "quiet_assets", ">= 1.0.1", :group => :development
gem "better_errors", ">= 0.0.8", :group => :development
gem "binding_of_caller", ">= 0.6.8", :group => :development
```

Add the following gems which will be needed for the rails-stripe-membership-saas application:

```
gem "stripe"
gem "stripe_event"
```

This tutorial requires Rails version 3.2.13.

Note: Rails Composer templates are created by the Rails Apps Composer Gem. For that reason, groups such as `:development` or `:test` are specified inline. You can reformat the Gemfile to organize groups in an eye-pleasing block style. The functionality is the same.

For more information about the Gemfile, see Gemfiles for Rails 3.2.

Install the Required Gems

When you add a new gem to the Gemfile, you should run the `bundle install` command to install the required gems on your computer. Run:

```
$ bundle install
```

You can check which gems are installed on your computer with:

```
$ gem list
```

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.

Test the App

You can check that your app runs properly by entering the command

```
$ rails server
```

To see your application in action, open a browser window and navigate to http://localhost:3000/. You should see the home page created by the starter app.

Stop the server with Control-C.

Git

Commit your changes to git:

```
$ git add -A
$ git commit -m "add gems"
```

Chapter 6

Test-Driven Development

This example application uses Cucumber for integration testing and RSpec for unit testing.

Testing is at the center of any robust software development process. Integration tests determine whether the application's features work as expected, testing the application from the point of view of the user. Unit tests confirm that small, discrete portions of the application continue working as developers add features and refactor code. RSpec is a popular choice for unit testing. The rails3-devise-rspec-cucumber tutorial shows how to set up RSpec and provides example specs for use with Devise. Cucumber is a popular choice for integration testing and behavior driven development. The rails3-devise-rspec-cucumber tutorial shows how to set up Cucumber and provides example scenarios for use with Devise.

To learn more about using RSpec, refer to The RSpec Book.

To learn more about using Cucumber, refer to The Cucumber Book or the free introduction to Cucumber, The Secret Ninja Cucumber Scrolls.

This tutorial assumes you've learned to write tests elsewhere (see a list of recommended resources for Rails). I won't spend time showing you how to write tests but you can use tests to make sure the application works as expected.

Tests Installed by Rails Composer

The Rails Composer tool creates a starter app that is set up for RSpec and Cucumber test frameworks.

The starter app includes RSpec and Cucumber test suites designed for the features of the rails3-bootstrap-devise-cancan example application.

Running RSpec Tests

The starter app script sets up RSpec for unit testing. You should be able to run `rake spec` to run all specs provided with the example app after the database is set up.

Running Cucumber Tests

The starter app script sets up Cucumber for specifications and acceptance testing.

You should be able to run `rake cucumber`, or more simply, `cucumber`, to run the Cucumber scenarios and steps provided with the example app after the database is set up. You can run a single Cucumber feature with a command such as:

```
$ cucumber features/visitors/request_invitation.feature
```

If you've used Cucumber, you may know that you need to add `--require features` to run a single Cucumber feature. Here it is not necessary to do so; the starter app script sets up the **config/cucumber.yml** file so it is not necessary to add `--require features`.

Installing Tests from the Example Application

The starter app only installs tests designed for the features of the rails3-bootstrap-devise-cancan example application. However, a full suite of tests for the rails-stripe-membership-saas example application are available in the GitHub repository.

You can copy the RSpec unit tests and Cucumber integration tests from the GitHub repository. Replace both the **spec** and **features** directories entirely. Copying all the files will include necessary configuration and helper files.

Run `rake -T` to check that rake tasks for RSpec and Cucumber are available. You won't be able to run `rake spec` or `rake cucumber` until the database is set up.

Chapter 7

Configuration

Configuration File

See the article Rails Environment Variables for more information.

The application uses the figaro gem to set environment variables. The starter app sets up the figaro gem and generates a **config/application.yml** file and lists it in your **.gitignore** file.

Credentials for your administrator account and email account are set in the **config/application.yml** file. The **.gitignore** file prevents the **config/application.yml** file from being saved in the git repository so your credentials are kept private.

Modify the file **config/application.yml**:

```
# Add account credentials and API keys here.
# See http://railsapps.github.io/rails-environment-variables.html
# This file should be listed in .gitignore to keep your settings secret!
# Each entry sets a local environment variable and overrides ENV variables in the Unix
shell.
# For example, setting:
# GMAIL_USERNAME: Your_Gmail_Username
# makes 'Your_Gmail_Username' available as ENV["GMAIL_USERNAME"]
# Add application configuration variables here, as shown below.
#
GMAIL_USERNAME: Your_Username
GMAIL_PASSWORD: Your_Password
ADMIN_NAME: First User
ADMIN_EMAIL: user@example.com
ADMIN_PASSWORD: changeme
ROLES: [admin, silver, gold, platinum]
STRIPE_API_KEY: Your_Stripe_API_key
STRIPE_PUBLIC_KEY: Your_Stripe_Public_Key
```

Set the user name and password needed for the application to send email.

If you wish, set your name, email address, and password for an administrator's account. If you prefer, you can use the default to sign in to the application and edit the account after deployment. It is always a good idea to change the administrator's password after the application is deployed.

The roles you specify in the configuration file are the subscription plans that will be available to the application's users. You will need an "admin" role. Keep the "silver", "gold", and "platinum" roles while you are testing the application. You can change these roles later, after you familiarize yourself with the application and begin to customize it for your own needs.

The Stripe gem requires an API key to operate. You also need to supply a public key when initiating a transaction. You can find both keys on your Stripe account page. Two sets of keys are available: one for testing, one for live transactions. Use the testing keys on your development machine. When you deploy, use the live keys.

All configuration values in the **config/application.yml** file are available anywhere in the application as environment variables. For example, `ENV["GMAIL_USERNAME"]` will return the string "Your_Username".

If you prefer, you can delete the **config/application.yml** file and set each value as an environment variable in the Unix shell.

Configure Email

The starter app script sets up a default email configuration. You must configure the application for your email account. See the article Send Email with Rails.

The starter app has already configured ActionMailer but you must set your email account details.

Replace `example.com` in the **config/environments/production.rb** file:

```
config.action_mailer.default_url_options = { :host => 'example.com' }
# ActionMailer Config
# Setup for production - deliveries, no errors raised
config.action_mailer.delivery_method = :smtp
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = false
config.action_mailer.default :charset => "utf-8"
```

The example application will deliver email in production. Email messages are visible in the log file so there is no need to send email in development.

In production, you should use an email service provider such as Mandrill to increase deliverability for email messages from your app.

Use Gmail for experimenting, if you want to keep things simple.

The file **config/environments/production.rb** is set to use:

```
config.action_mailer.smtp_settings = {
  address: "smtp.gmail.com",
  port: 587,
  domain: "example.com",
  authentication: "plain",
  enable_starttls_auto: true,
  user_name: ENV["GMAIL_USERNAME"],
  password: ENV["GMAIL_PASSWORD"]
}
```

You can replace `ENV["GMAIL_USERNAME"]` and `ENV["GMAIL_PASSWORD"]` with your Gmail username and password. However, committing the file to a public GitHub repository will expose your secret password. Instead, use local environment variables from the **config/application.yml** file to keep email account passwords secret.

Configure Devise for Email

Complete your email configuration by modifying

config/initializers/devise.rb

and setting the `config.mailer_sender` option for the return email address for messages that Devise sends from the application.

Stripe Initializer

You've set the Stripe API key and public key in your **config/application.yml** file. The file is there for security so your credentials won't be exposed publicly on a GitHub repo.

We'll use an initializer file to set the API key for use by the Stripe gem. We'll also set a constant for the public key which we can use in JavaScript code.

Create a file **config/initializers/stripe.rb**:

```
Stripe.api_key = ENV["STRIPE_API_KEY"]
STRIPE_PUBLIC_KEY = ENV["STRIPE_PUBLIC_KEY"]
```

The `Stripe.api_key` value and the `STRIPE_PUBLIC_KEY` constant are set using local environment variables from the **config/application.yml** file. The `Stripe.api_key` value should be kept secret so no other developer can use your Stripe account. The `STRIPE_PUBLIC_KEY` constant does not need to be kept secret but we set it in the **config/application.yml** file for convenience, so all your configuration settings are in one location.

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

Git

Commit your changes to git:

```
$ git add -A
$ git commit -m "configure"
```

Chapter 8

Layout and Stylesheets

This tutorial shows code using ERB, the default Rails templating language. If you prefer, you can generate the starter app with Haml instead of ERB. Then convert the ERB in the tutorial to Haml. See instructions for Haml and Rails.

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 for an explanation of each of the elements in the application layout.

The starter app:

- installs Twitter Bootstrap
- updates the application layout
- adds navigation links
- styles Rails flash messages
- installs and configures SimpleForm

See the article Twitter Bootstrap and Rails and review the tutorial for the rails3-bootstrap-devise-cancan example application for details.

The file **app/views/layouts/_navigation.html.erb** contains navigation links.

The file **app/views/layouts/_messages.html.erb** contains flash messages.

Base Errors Helper for SimpleForm

In this application, some failed transactions set model errors that are not matched to a specific form field. These are called "base errors." For example, the User model sets an error like this: `errors.add :base, "Credit card declined"`. SimpleForm does not provide a helper to display base errors; to accommodate base errors, we use a custom view helper.

The starter application installs this view helper in **app/helpers/application_helper.rb**:

```
module ApplicationHelper

  def display_base_errors resource
    return '' if (resource.errors.empty?) or (resource.errors[:base].empty?)
    messages = resource.errors[:base].map { |msg| content_tag(:p, msg) }.join
    html = <<-HTML
    <div class="alert alert-error alert-block">
      <button type="button" class="close" data-dismiss="alert">&#215;</button>
      #{messages}
    </div>
    HTML
    html.html_safe
  end

end
```

The view helper will be used in the files **app/views/devise/registrations/new.html.erb** and **app/views/devise/registrations/edit.html.erb**:

```
<%= simple_form_for ... %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
.
.
.
<% end %>
```

Without the `display_base_errors` view helper, the application's users will not see a "Credit card declined" message if a transaction is declined.

Chapter 9

Authentication

This application uses Devise for user management and authentication. Devise provides a system to securely identify users, making sure the user is really who he represents himself to be. Devise provides everything needed to implement user registration with log in and log out.

The starter app script sets up Devise:

- adds the Devise gem to the Gemfile
- runs `$ rails generate devise:install`
- uses Devise to generate a User model and database migration
- prevents logging of passwords
- adds a sign-in form that uses SimpleForm and Twitter Bootstrap

For details about how Devise is used in the starter application, see the tutorials:

- rails3-devise-rspec-cucumber
- rails3-bootstrap-devise-cancan

Chapter 10

Authorization

This application uses CanCan for authorization, to restrict access to pages that should only be viewed by an authorized user. CanCan offers an architecture that centralizes all authorization rules (permissions or "abilities") in a single location, the CanCan `Ability` class. CanCan provides a mechanism for limiting access at the level of controller and controller method and expects you to set permissions based on user attributes you define. This application uses Florent Monbillard's Rolify gem to create a Role model and add methods to a User model that are used to set CanCan permissions.

The starter app script sets up CanCan and Rolify:

- adds the CanCan and Rolify gems to the Gemfile
- creates the CanCan `Ability` class
- configures CanCan exception handling
- sets up User roles with Rolify

For details about how authorization is implemented in the starter application, see the tutorial:

- rails3-bootstrap-devise-cancan

Chapter 11

User Management

By default, Devise uses an email address to identify users. The starter application adds a "name" attribute as well.

Devise provides all the functionality for a user to log in and view and edit the user's profile. The user's profile only includes an email address, a name, and a password. You'll likely customize the User model and user pages for your own application.

The starter application:

- adds a name attribute to the User model
- limits mass-assignment operations with the `attr_accessible` method
- provides custom views for registering and editing users

For details about how user management is set up in the starter application, see the tutorials:

- rails3-devise-rspec-cucumber
- rails3-bootstrap-devise-cancan

Add Fields for Stripe

We will need two extra fields in the database to accommodate Stripe:

- `customer_id` will store the Stripe customer ID
- `last_4_digits` will store the last four digits of the user's credit card number

The `last_4_digits` field isn't needed for integration with Stripe but we can use it to show the user which credit card was used for the subscription.

We'll change our User model and database schema accordingly.

Migration for the User Model

Create a database migration with this command:

```
$ rails generate migration AddStripeToUsers customer_id:string last_4_digits:string
```

After you've created the migration, update the database:

```
$ rake db:migrate
```

If you get an error `uninitialized constant Stripe`, your Gemfile may be missing the Stripe gem or you may have negelcted to run `bundle install`.

If everything works, you can modify the User model.

Modify the User Model

There's no need to add the attributes `customer_id` or `last_4_digits` to the `attr_accessible` parameters because we won't be setting those vales from a form. We will remove the `attr_accessible :role_ids, :as => :admin` statement.

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

```
class User < ActiveRecord::Base
  rolify
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # Setup accessible (or protected) attributes for your model
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me

end
```

We remove the `attr_accessible :role_ids, :as => :admin` statement. This statement gives the administrator the power to override the mass assignment protection for the role attribute. The statement is no longer necessary because we will add a method that allows both the user and an administrator to reset the role.

With this change to our User model, we are ready to seed the database.

Git

Commit your changes to git:

```
$ git add -A
$ git commit -m "update user model"
```

Chapter 12

Initial Data

Set Up a Database Seed File

You'll want to set up default users so you can test the application. We'll add three users with different subscription plans.

The **db/seeds.rb** file initializes the database with default values. To keep some data private, and consolidate configuration settings in a single location, we use the **config/application.yml** file to set environment variables and then use the environment variables in the **db/seeds.rb** file.

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

```
puts 'ROLES'
YAML.load(ENV['ROLES']).each do |role|
  Role.find_or_create_by_name({ :name => role }, :without_protection => true)
  puts 'role: ' << role
end
puts 'DEFAULT USERS'
user = User.find_or_create_by_email :name => ENV['ADMIN_NAME'].dup, :email =>
ENV['ADMIN_EMAIL'].dup, :password => ENV['ADMIN_PASSWORD'].dup, :password_confirmation
=> ENV['ADMIN_PASSWORD'].dup
puts 'user: ' << user.name
user.add_role :admin
user2 = User.find_or_create_by_email :name => 'Silver User', :email =>
'user2@example.com', :password => 'changeme', :password_confirmation => 'changeme'
user2.add_role :silver
user3 = User.find_or_create_by_email :name => 'Gold User', :email =>
'user3@example.com', :password => 'changeme', :password_confirmation => 'changeme'
user3.add_role :gold
user4 = User.find_or_create_by_email :name => 'Platinum User', :email =>
'user4@example.com', :password => 'changeme', :password_confirmation => 'changeme'
user4.add_role :platinum
puts "users: #{user2.name}, #{user3.name}, #{user4.name}"
```

The **db/seeds.rb** file reads a list of roles from the **config/application.yml** file and adds the roles to the database. In fact, any new role can be added to the roles datatable with a statement such `user.add_role :superhero`. Setting the roles in the **db/seeds.rb** file simply makes sure each role is listed and available should a user wish to change roles.

We've added a user with an administrator role using vales from the **config/application.yml** file. You can log in with this account for access as an administrator.

You can change the administrator name, email, and password in this file but it is better to make the changes in the **config/application.yml** file to keep the credentials private. If you decide to include your private password in the **db/seeds.rb** file, be sure to add the filename to your **.gitignore** file so that your password doesn't become available in your public GitHub repository.

Note that it's not necessary to personalize the **db/seeds.rb** file before you deploy your app. You can deploy the app with an example user and then use the application's "Edit Account" feature to change name, email address, and password after you log in. Use this feature to log in as an administrator and change the user name and password to your own.

We've added three users and assigned "silver," "gold," and "platinum" roles corresponding to a tiered subscription plan.

Using "example.com" Email Addresses

We want the application to handle the administrator's account and any "example.com" email addresses as a special case. Users with an "example.com" domain will not be added to Stripe as subscribers; they will only be added to the application database. This makes it possible to run `rake db:seed` to add the administrator and set up sample users for development and testing.

Later we'll add two statements in an `update_stripe` method in the **app/models/user.rb** file implement the special case so that the Stripe server is not contacted:

```
def update_stripe
  return if email.include?(ENV['ADMIN_EMAIL'])
  return if email.include?('@example.com') and not Rails.env.production?
  .
  .
  .
end
```

As you can see from the code, the "example.com" email addresses will not be special cased in production. That means `rake db:seed` will fail with errors if you attempt to run it after deployment. You should remove the sample users (but not the administrator) from the **db/seeds.rb** file before deploying to production. See the section "Customize, Test, and Deploy" for advice about deploying to Heroku.

Seed the Database

The starter app script has already set up the database and added the default user by running:

```
$ rake db:migrate
$ rake db:seed
```

We'll need to reset the database because we've added new users:

```
$ rake db:reset
```

You can run `$ rake db:reset` whenever you need to recreate the database.

You'll also need to set up the database for testing:

```
$ rake db:test:prepare
```

If you're not using rvm, you should preface each rake command with `bundle exec`. You don't need to use `bundle exec` if you are using rvm version 1.11.0 or newer.

Commit to Git

Commit your changes to git:

```
$ git add -A
$ git commit -m "initial data"
```

Test the Starter App

At this point, the app is almost identical to the rails3-bootstrap-devise-cancan starter app.

You can check that the example app runs properly by entering the command:

```
$ rails server
```

To see your application in action, open a browser window and navigate to http://localhost:3000/. You should see the default users listed on the home page. When you click on a user's name, you should be required to log in before seeing the user's detail page.

If you sign in as the first user, you will have administrative privileges. You'll see an "Admin" link in the navigation bar. Clicking the "Admin" link will display the administrative dashboard. Each user will be listed with buttons to "Change role" or "Delete user."

Stop the server with Control-C.

When to Restart

If you install new gems, you'll have to restart the server to see any changes. The same is true for changes to configuration files in the config folder. This can be confusing to new Rails developers because you can change files in the app folders without restarting the server. As a rule, remember to restart the server when you add gems, change routes, or change anything in the config folder; leave the server running when you change models, controllers, views or anything else in app folder.

Chapter 13

Home Page

Replace the Home Page

If you've tested the example app, you've seen that any user who logs in will see a list of all the users on the home page. That's fine for an example app but it's not what we want for a subscription site.

Home Page with Subscription Plans

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

```
<div id="welcome" class="hero-unit span7">
  <h1>Membership Site</h1>
  <h3>Learn to build a successful subscription site.</h3>
</div>
<div class="row span8 plans">
  <div class="span2 well">
    <div class="plan"><h2>Silver</h2></div>
    <ul class="unstyled">
      <li>One lesson a month</li>
    </ul>
    <h3>$9/month</h3>
    <%= link_to 'Subscribe', content_silver_path, :class => 'btn btn-primary' %>
  </div>
  <div class="span2 well featured">
    <div class="plan featured-plan"><h2>Gold</h2></div>
    <ul class="unstyled">
      <li>Ten lessons a month</li>
    </ul>
    <h3>$19/month</h3>
    <%= link_to 'Subscribe', content_gold_path, :class => 'btn btn-primary' %>
  </div>
  <div class="span2 well">
    <div class="plan"><h2>Platinum</h2></div>
    <ul class="unstyled">
      <li>Thirty lessons a month</li>
    </ul>
    <h3>$29/month</h3>
    <%= link_to 'Subscribe', content_platinum_path, :class => 'btn btn-primary' %>
  </div>
</div>
```

We apply CSS classes from Twitter 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 a content page. The content pages don't yet exist; we'll need to create a Content controller, routes, and views to implement the content pages. Later, we'll change the links to open a subsciption purchase page instead of a content page.

CSS for Subscription Plans

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

First, modify the **app/assets/stylesheets/application.css.scss** file to remove the following CSS rules. These rules were useful for the starter app but will not be used in our application:

```
.content {
  background-color: #eee;
  padding: 20px;
  margin: 0 -20px; /* negative indent the amount of the padding to maintain the grid system */
  -webkit-border-radius: 0 0 6px 6px;
  -moz-border-radius: 0 0 6px 6px;
  border-radius: 0 0 6px 6px;
  -webkit-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  -moz-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  box-shadow: 0 1px 2px rgba(0,0,0,.15);
}
```

Next we'll add CSS assets to style the home page.

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

```
.plans{
  text-align: center;
}
.featured{
  -webkit-transform:scale(1.15);
  box-shadow: 0 0 30px rgba(0, 0, 0, 0.67);
}
.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 with an enlarged box for a "featured 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 Twitter Bootstrap themes available in

the WrapBootstrap marketplace. The theme for CSS3 Pricing Tables is particularly interesting. It is similar to our home page but adds animated effects. The theme only costs $6 and you can easily integrate it into our application by copying the contents of the designer's **css/custom.css** file into our **app/assets/stylesheets/pricing.css.scss** file and replacing styles in our **app/views/home/index.html.erb** file.

Modify the Home Controller

Modify the file **app/controllers/home_controller.rb** to remove the `index` method:

```
class HomeController < ApplicationController
end
```

Commit to Git

Commit your changes to git:

```
$ git add -A
$ git commit -m "home page update"
```

Chapter 14

Administrative Page

This application provides the **Users#index** page as an administrative dashboard. The CanCan authorization system restricts access to this page to only users in the “admin” role.

The starter application sets up the administrative page:

- lists users and shows the date each registered
- displays email addresses and roles
- provides buttons to change roles and delete users

For details about how the administrative page is set up by the starter application, see the tutorial:

- rails3-bootstrap-devise-cancan

Chapter 15

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.

First let's consider a git workflow for adding a new feature.

Git Workflow

When you are using git for version control, you can commit every time you save a file, even for the tiniest typo fixes. If only you will ever see your git commits, no one will care. But if you are working on a team, either commercially or as part of an open source project, you will drive your fellow programmers crazy if they try to follow your work and see such "granular" commits. Instead, get in the habit of creating a git branch each time you begin work to implement a feature. When your new feature is complete, merge the branch and "squash" the commits so your comrades see just one commit for the entire feature.

Create a new git branch for this feature:

```
$ git checkout -b content-pages
```

The command creates a new branch named "content-pages" and switches to it, analogous to copying all your files to a new directory and moving to work in the new directory (though that is not really what happens with git).

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/. 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 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** controller, 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 familar `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.

Test the Content Pages

You can check that the example app runs properly by entering the command:

```
$ rails server
```

Visit http://localhost:3000/ to see your subscription offer.

Visit http://localhost:3000/content/silver.html to see one of the content pages.

Next we'll set up access control to limit access to the content pages.

Git Workflow

If you haven't committed any changes yet, commit your changes to git:

```
$ git add -A
$ git commit -m "add content pages"
```

Since the new feature is complete, merge the working branch to “master” and squash the commits so you have just one commit for the entire feature:

```
$ git checkout master
$ git merge --squash content-pages
$ git commit -m "add content pages"
```

You can delete the working branch when you’re done:

```
$ git branch -D content-pages
```

Chapter 16

Limit Access to Content

We've got a home page with links to content pages for Silver, Gold, and Platinum subscribers. And we have created three users: Silver User, Gold User, and Platinum User. Now we'll set limits on access to the content.

Create a new git branch for this feature:

```
$ git checkout -b authorization
```

Set CanCan Ability

Modify the **app/models/ability.rb** class to set access limits:

```
class Ability
  include CanCan::Ability

  def initialize(user)
    user ||= User.new # guest user (not logged in)
    if user.has_role? :admin
      can :manage, :all
    else
      can :view, :silver if user.has_role? :silver
      can :view, :gold if user.has_role? :gold
      can :view, :platinum if user.has_role? :platinum
    end
  end
end
```

CanCan takes advantage of the Ruby language's facility to create a DSL (Domain Specific Language). If you ignore the cryptic punctuation, it's possible to read the DSL as if it was English. The statement `can :view, :silver if user.has_role? :silver` does what it says: If the user has a role of "silver" he or she can view "silver" content. Actually, a good portion of this is arbitrary. CanCan allows us to pass arbitrary parameters to the authorization method and I've simply chosen "view" and "silver" to be descriptive. All that matters is to use the same symbols in the Ability class as in the `authorize!` call in the controller. In the next step, you'll see how we add the `authorize!` call to the controller.

Set Access Limits in the Content Controller

We'll modify the **app/controllers/content_controller.rb** file to set access limits:

```
class ContentController < ApplicationController
  before_filter :authenticate_user!

  def silver
    authorize! :view, :silver, :message => 'Access limited to Silver Plan subscribers.'
  end

  def gold
    authorize! :view, :gold, :message => 'Access limited to Gold Plan subscribers.'
  end

  def platinum
    authorize! :view, :platinum, :message => 'Access limited to Platinum Plan
subscribers.'
  end
end
```

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

We use the CanCan `authorize!` method to check the user's role (corresponding to their subscription plan) on the actions that render the content pages. We pass two symbols to both `authorize!` (in the controller) and `can` (in the Ability class). The symbols can represent anything. By convention, the first symbol is the "action" one is trying to perform and the second symbol is the subject or target the action is being performed on. Our action is to "view" content but we could also say "see", "access", or "unlock." We've defined one target as "silver" but it could be "silver_content", "tier1", or "plan-A" as long as we are consistent between the controller and Ability class. The CanCan documentation describes CanCan with Non-RESTful Controllers.

Alternative Implementation

Keep in mind that we use CanCan only for convenience. You might not agree that CanCan provides benefit, especially when you consider that the CanCan DSL obscures the authorization mechanism. Cancan offers the advantage of collecting all authorization rules in the Ability class so the rules are easy to find and change. And it provides a familiar idiom for authorization so it is easy for other Rails developers (those who use CanCan) to understand your code.

If CanCan seems mysterious, it might help to see an alternative implementation without CanCan.

Here's the **app/controllers/content_controller.rb** file without CanCan:

```
class ContentController < ApplicationController
  before_filter :authenticate_user!

  def silver
    if (current_user.has_role? :silver) || (current_user.has_role? :admin)
      render :silver
    else
      redirect_to :back, :notice => 'Access limited to Silver Plan subscribers.'
    end
  end

  def gold
   if (current_user.has_role? :gold) || (current_user.has_role? :admin)
      render :gold
    else
      redirect_to :back, :notice => 'Access limited to Gold Plan subscribers.'
    end
  end

  def platinum
    if (current_user.has_role? :platinum) || (current_user.has_role? :admin)
      render :platinum
    else
      redirect_to :back, :notice => 'Access limited to Platinum Plan subscribers.'
    end
  end
end
```

By using the `:authenticate_user!` before_filter, Devise makes available the `current_user` instance of the User model. We still use the `has_role?` method provided by Rolify. If the current_user has an appropriate role, we render the appropriate page. If not, we redirect to the previous page and display a notice. The code is much easier to understand without CanCan but it is lengthy and repetitive.

Note: Please don't blindly paste the alternative implementation into the **app/controllers/content_controller** file if you want to follow the tutorial. We'll continue to use CanCan.

Test Authorization

You can check that the authorization limits work by entering the command:

```
$ rails server
```

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

Log in as the first user (the administrator) with “user@example.com@” (password “changeme”). Click on the “Subscribe” button for any subscription plan. CanCan grants you access as an administrator to any page. Log out.

Log in as the second user (assigned a “silver plan”) with “user2@example.com@” (password “changeme”). Click on the “Subscribe” button for any subscription plan. CanCan only grants access to the “silver” content page.

Next we’ll set up a page where a visitor can register to use the site after choosing a subscription plan.

Git Workflow

Commit your changes to git:

```
$ git add -A
$ git commit -m "add authorization"
$ git checkout master
$ git merge --squash authorization
$ git commit -m "add authorization"
$ git branch -D authorization
```

Chapter 17

Registration Page

Now that we have a home page with links to our placeholder content pages, plus authorization limits to restrict access based on subscription plan, let's create a page where a visitor can register for the site and sign up for a subscription.

For the initial version of our registration page, we'll let the visitor sign up for any subscription for free. Later, we'll integrate Stripe billing services for purchase of a subscription plan.

The Devise authentication gem provides user management, including signing up users on a registration page. Our starter app already has a User model and uses a Devise controller and views to sign up visitors, asking for a name and email address and creating a user account. We could add a new "subscription purchase" page. Instead, since we already have a sign-up form provided by Devise, it'll be easier to adapt the existing Devise registration page.

Create a new git branch for this feature:

```
$ git checkout -b registration
```

Modify the Home Page

We'll change the links on the home page to direct the visitors to the Devise registration page.

Update the contents of the file **app/views/home/index.html.erb**:

```
<div id="welcome" class="hero-unit span7">
  <h1>Membership Site</h1>
  <h3>Learn to build a successful subscription site.</h3>
</div>
<div class="row span8 plans">
  <div class="span2 well">
    <div class="plan"><h2>Silver</h2></div>
    <ul class="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="span2 well featured">
    <div class="plan featured-plan"><h2>Gold</h2></div>
    <ul class="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="span2 well">
    <div class="plan"><h2>Platinum</h2></div>
    <ul class="unstyled">
      <li>Thirty lessons a month</li>
    </ul>
    <h3>$29/month</h3>
    <%= link_to 'Subscribe', new_user_registration_path(:plan => 'platinum'), :class =>
'btn btn-primary' %>
  </div>
</div>
```

We've replaced the URL helper for each of the links:

- `content_silver_path` becomes `new_user_registration_path(:plan => 'silver')`
- `content_gold_path` becomes `new_user_registration_path(:plan => 'gold')`
- `content_platinum_path` becomes `new_user_registration_path(:plan => 'platinum')`

Notice that we append a parameter to each link to indicate the subscription plan selected by the visitor. We are initiating an HTTP GET request so you'll see a URL like this: `http://lvh.me:3000/users/sign_up?plan=silver`.

Modify the Navigation Links

Currently visitors see a “Sign up” link on the home page if they are not logged in. We’ll remove the “Sign up” link in the navigation bar because we want them to sign up by selecting a subscription plan.

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

```
<%= link_to "Rails Stripe Membership Saas", root_path, :class => 'brand' %>
<ul class="nav">
  <% if user_signed_in? %>
    <li>
    <%= link_to 'Logout', destroy_user_session_path, :method=>'delete' %>
    </li>
  <% else %>
    <li>
    <%= link_to 'Login', new_user_session_path %>
    </li>
  <% end %>
  <% if user_signed_in? %>
    <li>
    <%= link_to 'Edit account', edit_user_registration_path %>
    </li>
    <% if current_user.has_role? :admin %>
      <li>
      <%= link_to 'Admin', users_path %>
      </li>
    <% end %>
  <% end %>
</ul>
```

We’ve removed the “Sign up” link.

Override the Devise Registrations Controller

Create a file **app/controllers/registrations_controller.rb** to override the Devise registrations controller:

```
class RegistrationsController < Devise::RegistrationsController

  def new
    @plan = params[:plan]
    if @plan && ENV["ROLES"].include?(@plan) && @plan != "admin"
      super
    else
      redirect_to root_path, :notice => 'Please select a subscription plan below.'
    end
  end

  private
  def build_resource(*args)
    super
    if params[:plan]
      resource.add_role(params[:plan])
    end
  end
end
```

Devise provides a Registrations controller inside the Devise gem. It contains standard RESTful controller actions, including `new` and `create` (see Devise::RegistrationsController on GitHub).

We override the controller `new` action to set the *plan* variable. This allows us to display the name of the selected subscription plan on the registration page. When we modify the registration form, we'll include the *plan* variable in a hidden input field. The *plan* parameter is passed from the home page when the visitor clicks one of the "Subscribe" buttons.

If the *plan* parameter is present, we check to make sure the visitor has selected a plan that we're using in the application. We also make sure the visitor has not tried to create an account with "admin" acccess. A clever visitor may have noticed that the plan is specified in the URL and tried to change it.

If we have a legitimate plan, we call the `super` method to inherit the original controller `new` action. If the visitor has not selected a plan, we redirect back to the home page and display a message "Please select a subscription plan below."

Before calling `new` or `create`, the Devise Registrations controller initializes a new User with a private method named `build_resource`. The default `build_resource` method won't assign an authorization role to our user. We override the `build_resource` method, first using the `super` method to call the parent `build_resource` method. Then we check if the *plan* parameter is available. On a `new` action, it will be available as a parameter passed from the home page. On a `create` action, it will be passed as a parameter from a hidden input field on the registration form.

If you're wondering why you don't see the User model as a *@user* variable, Devise has aliased it as the *resource* instance variable (*resource* allows Devise to accommodate models with names other than User, such as Account or Person).

Rolify has added the `add_role` method to the User model, so we call `add_role` to assign the plan as an authorization role.

Instead of overriding the controller `new` action, we could use a `before_filter` to set the *plan* variable before the `new` method is called. We could do this:

```
before_filter :set_plan, :only => :new

def set_plan
  @plan = params[:plan]
end
```

Either implementation is effective; we've chosen to override the `new` method rather than using a `before_filter`.

Override the Devise Routes

Modify **config/routes.rb** to use the new controller. Replace `devise_for :users` with:

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

You've modified the routes file, so you must restart the server for any changes to take effect.

Modify the Devise Registration Page

We'll use the Devise registration page as our subscription sign-up form.

Modify the file **app/views/devise/registrations/new.html.erb** to add details about the subscription plan:

```
<h2>Sign up</h2>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name), :html => {:class => 'card_form form-vertical' }) do
|f| %>
  <h3><%= params[:plan].titleize if params[:plan] %> Subscription Plan</h3>
  <%= hidden_field_tag 'plan', params[:plan] %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= f.input :name, :autofocus => true %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :required => true %>
  <%= f.input :password_confirmation, :required => true %>
  <%= f.button :submit, 'Sign up', :class => 'btn-primary' %>
<% end %>
```

We display the name of the selected subscription plan. The `titleize` method transforms the *plan* string from lowercase to titlecase.

We add a hidden input field with `hidden_field_tag` to include a parameter identifying the selected subscription plan. The *plan* parameter is passed from the home page when the visitor clicks one of the "Subscribe" buttons and set as a variable by our Registrations controller.

We remove the `<%= render "devise/shared/links" %>` navigation links because we already have a login link in the page navigation bar.

When we submit the form, Rails does its form processing magic, including validation of the model attributes, and saves the new User record to the database.

Test Registration

You can check that registration works by entering the command:

```
$ rails server
```

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

Click on the "Subscribe" button for any subscription plan. You'll see the registration page. Fill in and submit the form. You'll see a message "Welcome! You have signed up successfully." Note that we're not using the Devise Confirmable module so the application doesn't send a confirmation email. You'll be logged in as the new user as soon as you submit the form.

If you've selected the Silver Plan for the new user, try visiting http://localhost:3000/content/silver.html. You should be able to view the page. You should see an error message if you attempt to visit http://localhost:3000/content/gold.html.

Now we have a registration page that assigns a subscription plan when a visitor signs up for the site. In the next step, we'll make sure the user gets redirected to an appropriate page after sign up or log in.

Git Workflow

Commit your changes to git:

```
$ git add -A
$ git commit -m "add registration"
$ git checkout master
$ git merge --squash registration
$ git commit -m "add registration"
$ git branch -D registration
```

Chapter 18

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.

Get started by creating a new git branch for this feature:

```
$ git checkout -b redirect
```

Modify the Application Controller

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

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  rescue_from CanCan::AccessDenied do |exception|
    redirect_to root_path, :alert => exception.message
  end

  def after_sign_in_path_for(resource)
    case current_user.roles.first.name
      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 user's role and redirects to the appropriate content page, or if an administrator, to the administrative dashboard.

When we use Rolify, each user can have multiple roles. We only assign a single role in this application, so we ask for the first role associated with the user. When a role object is displayed as a string, it will be a number, so we ask for the name attribute of the role.

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 Redirect

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

```
$ rails server
```

Visit 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.

In the next step, we'll integrate Stripe payment for subscription plans.

Git Workflow

Commit your changes to git:

```
$ git add -A
$ git commit -m "redirect after log in"
$ git checkout master
$ git merge --squash redirect
$ git commit -m "redirect after log in"
$ git branch -D redirect
```

Chapter 19

Stripe Integration

We've got a fully functional web application that serves up placeholder content to registered users and restricts access based on the subscription plan the user has selected. Now we'll integrate Stripe billing so we can charge users for subscriptions.

Remember we're using our Devise registration page for subscription sign up. We'll make this our payment page. We'll ask the visitor to enter credit card data on this page.

Here's the most obvious way to implement billing:

- the visitor enters credit card data on a form
- the visitor submits the form
- data is received by your application on your server
- an application controller `create` action initiates a request to the payment processor
- the `create` method receives an acknowledgment of a completed or declined transaction
- the `create` method saves the user data (and creates an account) or shows an error message

There's a big drawback to this approach: Credit card data is sent to your server. Your server becomes a target for thieves and you must secure the server and maintain PCI compliance (the credit card industry security standard).

Instead, Stripe offers an architecture that cuts risk and makes it much easier to meet PCI compliance requirements. Stripe provides a JavaScript file that we add to our payment page. We set a public key in the JavaScript code to identify our Stripe account. We trigger the JavaScript code when the user submits the payment form. The JavaScript code obtains the credit card data from our form, transmits the data to Stripe's servers, and returns a unique token that serves as a substitute for the credit card data. Stripe takes responsibility for the security of the credit card data and it never touches our server. We can safely use the Stripe token as a substitute for credit card data when we make a request to Stripe to bill the user.

Here is our preferred program flow:

- the visitor enters credit card data on a form
- the visitor submits the form
- a JavaScript function sends the card data to Stripe's servers
- a JavaScript callback receives a Stripe token or an error message

- the JavaScript function submits the form to our server, substituting the token for credit card data
- a User model `before_save` method sends a request (with the token) to Stripe to create a customer account
- the User model `before_save` method receives an acknowledgment of a completed or declined transaction
- a Registrations controller `create` action saves the user data or shows an error message

This hybrid approach makes implementation more complex but the result is robust and secure.

Here are useful resources for understanding the implementation:

- RailsCast Billing with Stripe
- Stripe documentation
- Stripe's monospace-rails sample Rails application
- Nettuts: Stripe Tutorial

We've followed Ryan Bates's implementation from his RailsCast on Stripe in several key areas. Thank you, Ryan!

Get started by creating a new git branch for this feature:

```
$ git checkout -b billing
```

Add Virtual Attributes to the User Model

When a new user signs up for a subscription, we'll use JavaScript on the registration page to submit credit card data to Stripe and obtain a Stripe token that substitutes for credit card data. The Stripe token will be included as a hidden field when the user submits the registration form.

Before we can include a hidden field in the form, we must modify the User model to accept this field. It doesn't need to be added to the User database schema as it will only be used when the form is processed by the Registrations controller. We'll add it as a virtual attribute. For more on virtual attributes, see the RailsCast Virtual Attributes.

We'll also add a `coupon` field as a virtual attribute.

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

```
class User < ActiveRecord::Base
  rolify
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # Setup accessible (or protected) attributes for your model
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me,
:stripe_token, :coupon
  attr_accessor :stripe_token, :coupon

end
```

We add the virtual attribute with `attr_accessor` which will automatically create getter and setter methods and store the `stripe_token` and `coupon` values as instance variables. Also we add the `stripe_token` and `coupon` fields to the `attr_accessible` list so they can be set through mass assignment from a form submission.

With the new virtual attributes in place, we will begin work on the forms that submit the credit card data.

Use a Metatag to Set the Stripe Public Key

Stripe provides a unique public key that identifies your website and authenticates a request to the Stripe servers.

You've set the Stripe public key in your **config/application.yml** file. We put it there for convenience, so your Stripe configuration settings are in one location and easier to copy to Heroku for deployment. Unlike the Stripe API key, the public key doesn't need to stay secret.

We set a constant `STRIPE_PUBLIC_KEY` in the file **config/initializers/stripe.rb**. For ease in testing, we use a constant rather than using the environment variable directly.

Now we must make the public key available to the JavaScript code we'll use to submit credit card data to the Stripe server.

We have several options. We could hardcode the Stripe public key in the JavaScript code we'll write. But it is easier to use the constant we've already set, so we're using the same value in development, testing, and production. We could include the constant as a Ruby variable in the JavaScript code by giving the JavaScript file an **.erb** extension. But we'd like to avoid mixing Ruby variables into JavaScript. So instead we'll use an advanced technique to set the Stripe public key constant as a metatag that can be accessed using conventional JavaScript.

We only want the metatag to appear on pages that contain forms that submit credit card data to Stripe. We'll use a combination of a `content_for?` view helper with `yield` to inject the metatag into the application layout from the **app/views/devise/registrations/new.html.erb** and **app/views/devise/registrations/edit.html.erb** files. Our application layout already includes `<%= yield(:head) %>` so we can easily add additional tags to the page HEAD element.

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

```
<% content_for :head do %>
  <%= tag :meta, :name => "stripe-key", :content => STRIPE_PUBLIC_KEY %>
<% end %>
<h2>Sign up</h2>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name), :html => {:class => 'card_form form-vertical' }) do
|f| %>
  <h3><%= params[:plan].titleize if params[:plan] %> Subscription Plan</h3>
  <%= hidden_field_tag 'plan', params[:plan] %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= f.input :name, :autofocus => true %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :required => true %>
  <%= f.input :password_confirmation, :required => true %>
  <%= f.button :submit, 'Sign up', :class => 'btn-primary' %>
<% end %>
```

The `<% content_for :head do %>` block injects the `STRIPE_PUBLIC_KEY` in the application layout as a metatag named "stripe-key."

Do the same for the file **app/views/devise/registrations/edit.html.erb**:

```
<% content_for :head do %>
  <%= tag :meta, :name => "stripe-key", :content => STRIPE_PUBLIC_KEY %>
<% end %>
<h2>Edit <%= resource_name.to_s.humanize %></h2>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name), :html => { :method => :put, :class => 'form-vertical'
}) do |f| %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= f.input :name, :autofocus => true %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :autocomplete => "off", :hint => "leave it blank if you don't
want to change it", :required => false %>
  <%= f.input :password_confirmation, :required => false %>
  <%= f.input :current_password, :hint => "we need your current password to confirm your
changes", :required => true %>
  <%= f.button :submit, 'Update', :class => 'btn-primary' %>
<% end %>
<h3>Cancel my account</h3>
<p>Unhappy? <%= link_to "Cancel my account", registration_path(resource_name), :data =>
{ :confirm => "Are you sure?" }, :method => :delete %>.</p>
<%= link_to "Back", :back %>
```

Any JavaScript used on these two pages will have access to the Stripe public key through a jQuery selector. It's more complicated than hardcoding the Stripe public key or using a Ruby variable in the JavaScript but the layers of indirection make the implementation more robust.

With the Stripe public key set as a metatag, we will add fields for credit card data to the form.

Add Credit Card Data to the Registration Form

We'll need fields for a credit card number, security code, and expiration date.

We'll also add a placeholder `div` for any error messages returned by the Stripe server.

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

```
<% content_for :head do %>
  <%= tag :meta, :name => "stripe-key", :content => STRIPE_PUBLIC_KEY %>
<% end %>
<h2>Sign up</h2>
<div id="stripe_error" class="alert alert-error" style="display:none" >
</div>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name), :html => {:class => 'card_form form-vertical' }) do
|f| %>
  <h3><%= params[:plan].titleize if params[:plan] %> Subscription Plan</h3>
  <%= hidden_field_tag 'plan', params[:plan] %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= f.input :name, :autofocus => true %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :required => true %>
  <%= f.input :password_confirmation, :required => true %>
  <% if @user.stripe_token %>
    <p>Credit card acceptance is pending.</p>
  <% else %>
    <div class="field">
      <%= label_tag :card_number, "Credit Card Number" %>
      <%= text_field_tag :card_number, nil, name: nil %>
    </div>
    <div class="field">
      <%= label_tag :card_code, "Card Security Code (CVV)" %>
      <%= text_field_tag :card_code, nil, name: nil %>
    </div>
    <div class="field">
      <%= label_tag :card_month, "Card Expiration" %>
      <%= select_month nil, {add_month_numbers: true}, {name: nil, id: "card_month"}%>
      <%= select_year nil, {start_year: Date.today.year, end_year: Date.today.year+10},
{name: nil, id: "card_year"}%>
    </div>
    <div class="field">
      <%= f.input :coupon, :label => 'Promotional Coupon (if any)' %>
    </div>
  <% end %>
  <%= f.hidden_field :stripe_token %>
  <%= f.button :submit, 'Sign up', :class => 'btn-primary' %>
<% end %>
```

Take a look at `div id="stripe_error"`. We give it the “alert alert-error” style from Twitter Bootstrap. It remains hidden with a `style="display:none"` unless we use JavaScript to show it when we receive an error message from the Stripe server. We’ll add the necessary JavaScript soon.

If you examine the HTML source, you will see:

```
<form accept-charset="UTF-8" action="/users" class="card_form form-vertical"
id="new_user" method="post">
```

The Rails form builder creates a form with the id `new_user`. The id is generated automatically from the controller and action names "user" and "new". We assign the class `card_form` to the form. Our JavaScript code will interact with the form. We could select the form with the "new_user" id but instead we'll use the "card_form" class in our jQuery selector. This gives us flexibility to use the same JavaScript code to interact with either a "new registration" or "edit registration" form.

We've added fields for credit card data but we've made inclusion of the fields conditional on the absence of the `@user.stripe_token` attribute. This accommodates a situation where correct credit card data is provided but the form fails to pass a validation test (such as a blank email address). If Stripe accepts the credit card data and provides a `stripe_token` but there's no email address, we'll save the Stripe token in the hidden field and prompt the user to correct the email address.

Notice the differences between the credit card data fields and the previous data fields.

The name, 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.text_field` create form controls 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. For the expiration `select_month` and `select_year` we also set an element ID to replace the ID that the Rails form helper generates. Thus the fields will be available to our JavaScript code but will not be submitted to the server.

We add a hidden field to the form named `stripe_token`. We'll soon add JavaScript to submit credit card data from the form and obtain a Stripe token from the Stripe server. The JavaScript code will add the Stripe token to the hidden field before submitting the form to our server.

Now we'll add the JavaScript to process the form.

Adding JavaScript from the Stripe Server

There are several approaches to accessing external JavaScript libraries and setting up scripts to only run on a specific page. For a comprehensive introduction to using external JavaScript libraries and page-specific JavaScript, see our article Adding JavaScript to Rails.

Developers often copy external JavaScript libraries to the **vendor/assets/javascripts** folder and let the Rails asset pipeline combine all the JavaScript, making it available throughout the application. That's not how we want to handle the Stripe JavaScript library. We don't want to store it in our application where a trespasser could modify it to intercept credit card data. It is best to load it from Stripe's server. And we want to execute it only on the registration page.

We'll use two techniques described in the article Adding JavaScript to Rails. We'll add a script named *jquery.readyselector.js* (from John Firebaugh) that allows us to execute JavaScript conditionally on a page. A second script, *jquery.externalscript.js*, extends jQuery to add a function that allows us to download (or load from the cache) a JavaScript file from a remote server.

Script for Conditional Execution of Page-Specific JavaScript

First, make sure that the application layout file is set to provide parameters needed for the *jquery.readyselector.js* script. The default RailsApps application layout file **app/views/layouts/application.html.erb** should already contain a `<body>` tag that looks like this:

```
<body class="<%= controller_name %> <%= action_name %>">
```

If it doesn't, make sure to add it now.

We'll add a script that will help us execute JavaScript conditionally on a page.

Create a file **app/assets/javascripts/jquery.readyselector.js**:

```
(function ($) {
  var ready = $.fn.ready;
  $.fn.ready = function (fn) {
    if (this.context === undefined) {
      // The $().ready(fn) case.
      ready(fn);
    } else if (this.selector) {
      ready($.proxy(function(){
        $(this.selector, this.context).each(fn);
      }, this));
    } else {
      ready($.proxy(function(){
        $(this).each(fn);
      }, this));
    }
  }
})(jQuery);
```

If you prefer CoffeeScript, you can convert JavaScript to CoffeeScript here or here.

If you haven't changed the default manifest file, the *jquery.readyselector.js* script will be automatically loaded by the `//= require_tree .` directive in the **app/assets/javascripts/application.js** manifest file. If you've removed the `//= require_tree .` directive, specify the script in the **app/assets/javascripts/application.js** manifest file:

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

Script for Dynamic Loading of Remote JavaScript Files

Here's how we can download (or load from the cache) a JavaScript file from a remote server.

Other approaches to downloading a JavaScript file from a remote server require adding additional `<script>` tags to the page. For the optimal performance, it is better to use the following technique. See our article Adding JavaScript to Rails for details.

Create a file **app/assets/javascripts/jquery.externalscript.js**:

```
jQuery.externalScript = function(url, options) {
  options = $.extend(options || {}, {
    dataType: "script",
    cache: true,
    url: url
  });
  return jQuery.ajax(options);
};
```

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
```

Next we'll combine functions from these two scripts to load the Stripe JavaScript library.

Create a Registrations JavaScript File

We'll create a new file for our page-specific JavaScript and give the file the same name as the controller. For now, we'll only create some test code.

Create a file **app/assets/javascripts/registrations.js**:

```
$('.home.index').ready(function() {
  console.log("Page-specific JavaScript on the home.index page.");
});

$('.registrations.new').ready(function() {
  console.log("Page-specific JavaScript on the registrations.new page.");
});

$('.registrations').ready(function() {
  $.externalScript('https://js.stripe.com/v1/').done(function(script, textStatus) {
    console.log('Script loading: ' + textStatus );
    if (typeof Stripe != 'undefined') {
      console.log('Okay. Stripe file loaded.');
    }
    else
    {
      console.log('Problem. Stripe file not loaded.');
    }
  });
});
```

This code will test if the Stripe JavaScript library has been successfully downloaded.

Later we will add the code that collects the credit card data from the subscription form, submits it to the Stripe server, and obtains the Stripe token.

Let's test the code to make sure the Stripe JavaScript library is loading from the remote server.

Start the server and visit the home page. Check the JavaScript console to observe the debugging messages.

Here's how to use the JavaScript console. Every web browser provides access to the JavaScript console for debugging. In Chrome, choose the menu item *View/Developer/JavaScript Console*. Reload the page and you will see console messages. We've added console messages to trace the program flow as each function is executed. Use this technique for debugging if you have problems.

When you visit the home page, you should see a debugging message in the JavaScript console:

```
Page-specific JavaScript on the home.index page.
```

Click a "Subscribe" button. You should see the registration page. You'll see more debugging messages in the JavaScript console:

```
Page-specific JavaScript on the registrations.new page.
Script loading: success
Okay. Stripe file loaded.
```

If you see these messages, you're ready to continue.

Add JavaScript for Form Processing

We need code that runs in the browser to submit the credit card data to Stripe and obtain a Stripe token before the form is sent to our server.

Replace the contents of **app/assets/javascripts/registrations.js** with this:

```
$('.registrations').ready(function() {
  $.externalScript('https://js.stripe.com/v1/').done(function(script, textStatus) {
    if (typeof Stripe != 'undefined') {
      console.log('Stripe JavaScript file loaded.');
    }
    else
    {
      console.log('Problem: Stripe JavaScript file not loaded.');
    }
    Stripe.setPublishableKey($('meta[name="stripe-key"]').attr('content'));
    console.log('set Stripe public key: ' +
$('meta[name="stripe-key"]').attr('content'));
    var subscription = {
      setupForm: function() {
        console.log('function: setupForm')
        return $('.card_form').submit(function() {
          console.log('setupForm: form submitted')
          $('input[type=submit]').prop('disabled', true);
          subscription.processCard();
          return false;
        });
      },
      processCard: function() {
        console.log('function: processCard');
        var card;
        card = {
          name: $('#user_name').val(),
          number: $('#card_number').val(),
          cvc: $('#card_code').val(),
          expMonth: $('#card_month').val(),
          expYear: $('#card_year').val()
        };
        return Stripe.createToken(card, subscription.handleStripeResponse);
      },
      handleStripeResponse: function(status, response) {
        console.log('function: handleStripeResponse');
        if (status === 200) {
          return alert('Stripe response: ' + response.id);
        } else {
          return alert('Stripe response: ' + response.error.message);
        }
      }
    };
    return subscription.setupForm();
  });
});
```

This is not the final version of the JavaScript code; it contains debugging messages so you can verify the code is working.

We add the functionality we need to process a form and submit the credit card data to the Stripe server. We'll look closely at the code in the next section. For now, let's test the new code.

Start the server and visit the page. Check the JavaScript console to observe the debugging messages. Fill out the form, using a fake credit card number.

Stripe provides a set of fake credit card numbers which will force various responses. See the Stripe Testing documentation. Here are some useful numbers:

- a fake Visa card: *4242424242424242*
- incorrect number: use a number that fails the checksum, e.g. *4242424242424241*
- invalid expiry month: use an invalid month, e.g. *13*
- invalid expiry year: use a year in the past, e.g. *1970*
- invalid cvc: use a two digit number, e.g. *99*

Submit the form. You should see an alert displaying the response from the Stripe server.

It's fun (for a few minutes) to enter various numbers and observe the response from the Stripe server.

Here's what you should see in the JavaScript console:

```
Stripe JavaScript file loaded.
set Stripe public key: ...
function: setupForm
setupForm: form submitted
function: processCard
function: handleStripeResponse
```

Yea! You're communicating with the Stripe server.

Now that you've confirmed you can send credit card data to the Stripe server, remove the debugging code from the script. While we're at it, we'll add the code to handle the response from the Stripe server.

Replace the contents of **app/assets/javascripts/registrations.js** with this:

```
$('.registrations').ready(function() {
  $.externalScript('https://js.stripe.com/v1/').done(function(script, textStatus) {
      Stripe.setPublishableKey($('meta[name="stripe-key"]').attr('content'));
      var subscription = {
        setupForm: function() {
          return $('.card_form').submit(function() {
            $('input[type=submit]').prop('disabled', true);
            if ($('#card_number').length) {
              subscription.processCard();
              return false;
            } else {
              return true;
            }
          });
        },
        processCard: function() {
          var card;
          card = {
            name: $('#user_name').val(),
            number: $('#card_number').val(),
            cvc: $('#card_code').val(),
            expMonth: $('#card_month').val(),
            expYear: $('#card_year').val()
          };
          return Stripe.createToken(card, subscription.handleStripeResponse);
        },
        handleStripeResponse: function(status, response) {
          if (status === 200) {
            $('#user_stripe_token').val(response.id)
            $('.card_form')[0].submit()
          } else {
            $('#stripe_error').text(response.error.message).show();
            return $('input[type=submit]').prop('disabled', false);
          }
        }
      };
      return subscription.setupForm();
  });
});
```

Here's an explanation of the JavaScript.

We restrict execution of the code to a page that has a class "registrations." In this case, the code will run on the Registrations#New page. It will also run on the Registrations#Edit page which we will set up later.

We use the `externalScript` function to obtain the Stripe JavaScript library from the Stripe server (or a local cache).

We obtain the Stripe public key. You'll recall it is initially set as an environment variable in the **config/application.yml** file, set as a constant in the **config/initializers/stripe.rb** file, and injected as a metatag in the application layout by the **app/views/devise/registrations/new.html.erb** view. It is finally obtained from the metatag with a jQuery selector.

We create a `subscription` object with several functions: `setupForm`, `processCard`, and `handleStripeResponse`.

We call the `setupForm` function which listens for the form submission event. When the visitor submits the form, the submit button is disabled (so the user can't press it repeatedly) and the `processCard` function is called. Returning `false` makes sure the form is not submitted to our server at this stage. Notice that we don't call the `processCard` function if the credit card number is blank; instead we return "true" which submits the form. This will be true if the credit card data was previously accepted (and a Stripe token is included in the hidden field) but there was an error such as a missing email address.

The `processCard` function parses the form and obtains the credit card data. Then it calls a function from the Stripe library named `createToken` which submits the card data to the Stripe server. When the Stripe server returns a response, we call the `handleStripeResponse` function.

The `handleStripeResponse` function checks the HTTP status code. If it is okay (status code 200), the function sets the `stripe_token` hidden field in the form (it has the HTML element ID "user_stripe_token") and calls `submit` on the element with the "card_form" class (which is the form). If the HTTP status code is not okay, the function sets the error message from the Stripe server in the `stripe_error` div and re-enables the submit button.

In effect, we modify the form's submit button so that the credit card data is first sent to the Stripe server, the Stripe token is obtained and set in a hidden field, and then the form is submitted to our application for processing by our Registrations controller.

You can start the server and create a subscription. Use the fake credit card number *4242424242424242*. You'll see an error message on the page if the credit card data is incomplete. If the credit card data is accepted, a new User account will be created and you'll be redirected to an appropriate content page where you'll see a message "Welcome! You have signed up successfully."

But wait. We've created a new User account but we haven't charged the user for the subscription. To do so, we'll modify the User model to create a new Stripe customer and initiate a credit card transaction when a new user is created.

Prepare Your Stripe Account

Before we can modify the User model to create a new Stripe customer, we have to prepare our Stripe account to recognize the parameters we'll be saving. We have to set up our subscription plans in Stripe.

This step is necessary if you're going to use Stripe to automatically handle recurring billing. We'll tell Stripe that we have 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 to create a subscription plan. Stripe offers documentation about creating a plan and additional detail about 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 should have an ID that corresponds to the roles we've created 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.

Now that Stripe knows about our subscription plans, we'll modify the User model to create a new Stripe customer.

Modify the User Model

We want to make a request to Stripe to create a customer whenever a new user is created. There are several ways we could do this:

- modify the Registrations controller `create` action to initiate a request to Stripe
- add a new method such as `save_new_customer` to the User model and swap it for the `save` method used in the Registrations controller `create` action
- add a `save_new_customer` method to the User model and call it with `before_create` in the User model
- add an `update_stripe` method to the User model and call it with `before_save` in the User model

The first option, modifying the Registrations controller, results in the much-abhorred “fat controller” anti-pattern. We’ll avoid that. The second option moves the Stripe request to the model but there is no need to modify the Registrations controller to use an alternative to the `save` method. The third option requires no changes to the Registrations controller but only contacts Stripe when a new user is created. The fourth option is optimal. We’ll add an `update_stripe` method that is called whenever the user instance is saved. This approach allows us to either create a new Stripe customer on `create` or update the Stripe customer record on `update`.

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

```
class User < ActiveRecord::Base
  rolify
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # Setup accessible (or protected) attributes for your model
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me,
:stripe_token, :coupon
  attr_accessor :stripe_token, :coupon
  before_save :update_stripe

  def update_stripe
    return if email.include?(ENV['ADMIN_EMAIL'])
    return if email.include?('@example.com') and not Rails.env.production?
    if customer_id.nil?
      if !stripe_token.present?
        raise "Stripe token not present. Can't create account."
      end
      if coupon.blank?
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name
        )
      else
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name,
          :coupon => coupon
        )
      end
    else
      customer = Stripe::Customer.retrieve(customer_id)
      if stripe_token.present?
        customer.card = stripe_token
      end
      customer.email = email
      customer.description = name
      customer.save
    end
    self.last_4_digits = customer.cards.data.first["last4"]
    self.customer_id = customer.id
    self.stripe_token = nil
  rescue Stripe::StripeError => e
```

```
    logger.error "Stripe Error: " + e.message
    errors.add :base, "#{e.message}."
    self.stripe_token = nil
    false
  end

end
```

We add `before_save :update_stripe` to call the `update_stripe` method when the user is saved or updated.

The `before_save` callback is carried out after validation so there will be no attempt to create a Stripe customer if an email address is missing or if there are other validation errors.

The `update_stripe` method makes our request to the Stripe service. We skip the process if the email address contains our administrator's email address or includes "example.com". This allows us to run `rake db:seed` or `rake db:reset` to set up our administrative user and default sample accounts for this tutorial.

The `update_stripe` method checks if the user instance already has a `customer_id`. If it does not, and a `stripe_token` is present, we can create a new Stripe customer. To create a new Stripe customer, we make a request using the Stripe API call `Stripe::Customer.create`.

We will create a new subscription when we create a new customer. We provide the customer's email address and name. We supply the `stripe_token` which tells Stripe how to charge the customer. We provide a `plan` which tells Stripe how much to charge. We're using Rolify roles to designate subscription plans so we retrieve the role name and use it to designate the subscription plan.

If we have a coupon, we include a `coupon` identifier so Stripe will apply a discount.

We don't transmit any credit card data. Stripe already has the credit card data keyed to the stripe token; we transmit the token instead of providing credit card data.

If `customer_id` is already available as an attribute of the user instance, we know that we're updating a record rather than creating a new customer. We use the Stripe API call `Stripe::Customer.retrieve(customer_id)` to obtain a customer object and then reset the email address and name. If a `stripe_token` is present, we tell Stripe to use a new credit card. We'll implement this function when we modify the application's "Edit Account" feature.

After create or update, we obtain the last four digits of the user's credit card number. We finish by setting the `customer_id` to match the Stripe customer ID and we set the stripe token to nil.

The `update_stripe` method sets user attributes `customer_id` and `last_4_digits` which are saved with the attributes passed from the form when the Registration controller completes the `user.save` call.

We handle errors with a `rescue` condition. Errors in credit card data will be caught by the JavaScript functions before the form is submitted to the server. If another type of error prevents the Stripe customer from being created, the `rescue` condition will set an error message and return false which will prevent the user account from being saved.

Test Stripe Integration

We have all the code needed for visitors to sign up for new subscriptions. Let's test the application.

Reset the database:

```
$ rake db:reset
```

Start the web server:

```
$ rails server
```

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

Click on the "Subscribe" button for any subscription plan. You'll see the registration page. Fill in the form using the fake credit card number *4242424242424242*. Submit the form.

You'll be redirected to an appropriate page and see a message "Welcome! You have signed up successfully." You'll be logged in as a new user. If you've selected the Silver Plan for the new user, you should be on the page http://localhost:3000/content/silver.html. You should see an error message if you attempt to visit http://localhost:3000/content/gold.html.

Visit your Stripe dashboard at https://manage.stripe.com/#test/customers and see the new customer you've created. If you check the customer details, you should see that the fake credit card was billed and the subscription status is "active."

Pat yourself on the back! You've reached a milestone. You now can take credit card payments for subscriptions.

There's more to do, however:

- modify "edit account" so users can change credit cards, change plans, and unsubscribe
- implement Stripe "webhooks" so expired subscribers will be denied access to the site

Let's update our Git repository before we continue.

Git Workflow

Commit your changes to git:

```
$ git add -A
$ git commit -m "add stripe"
$ git checkout master
$ git merge --squash billing
$ git commit -m "add stripe"
$ git branch -D billing
```

Chapter 20

Account Changes

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 use a different credit card for payments? We could require them to cancel their subscription and create a new one. It would be better if they could edit their account to change their credit card.

We'll also make it easy for a subscriber to upgrade or downgrade a subscription plan and cancel a subscription.

Get started by creating a new git branch for this feature:

```
$ git checkout -b change-acct
```

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 two additional forms that are hidden on the page and are revealed as modal windows when a "Change plan" or "Change card" button is clicked.

We've got a lot of code to add, so we'll do it in stages.

We'll modify the file **app/views/devise/registrations/edit.html.erb**.

There's no need to replace the existing form. It is useful for changing the user's name, email address, or password.

Add this code before the `<h3>Cancel my account</h3>` statement:

```
<h3>Subscription Plan</h3>
<p>
  <%= @user.roles.first.name.titleize %>
  <a data-toggle="modal" href="#plan-options" class="btn btn-mini" type="button">Change
plan</a>
</p>

<h3>Card</h3>
<p>
  Using card ending with <%= @user.last_4_digits %>
  <a data-toggle="modal" href="#card-data" class="btn btn-mini" type="button">Change
card</a>
</p>
```

We display the name of the current subscription plan, which is derived from the roles object associated with the user. Rolify lets us set multiple roles (a feature we will not use in this application). We pick the first role (the only role), obtain its name, and "titleize" it to display it in titlecase. We add a link that is styled as a Twitter button. The link will open a Twitter Bootstrap modal window labeled "plan-options", which we will add soon.

We do something similar for the credit card. We show the last four digits of the card that was used to purchase the subscription. Then we add a link that will open a Twitter Bootstrap modal window labeled "card-data". The link will be styled as a miniature button.

For stylistic consistency, replace the next statement with the following:

```
<p>Unhappy? <%= link_to "Cancel my account", registration_path(resource_name), :data =>
{ :confirm => "Are you sure?" }, :method => :delete %>.</p>
```

```
<p>Unhappy? <%= link_to "Cancel my account", registration_path(resource_name), :data =>
{ :confirm => "Are you sure?" }, :method => :delete, :class => 'btn btn-mini' %></p>
```

We've added a Twitter Bootstrap class that styles the link as a miniature button.

You can also remove the line:

```
<%= link_to "Back", :back %>
```

Now we'll add the first of two modal windows containing a form. It doesn't matter where the forms are placed in the file as they will be hidden when the page is initially displayed and will appear as an overlay when revealed by a click on the appropriate link. I suggest you append the code at the end of the file.

The first form is identifed as "plan-options":

```
<div id="plan-options" class="modal" style="display: none;">
  <%= simple_form_for resource, :as => resource_name, :url => update_plan_path, :html =>
{:method => :put, :class => 'form-horizontal' } do |f| %>
    <div class="modal-header">
      <a class="close" data-dismiss="modal">&#215;</a>
      <h3>Change Plan</h3>
    </div>
    <div class="modal-body">
      <%= f.input :role_ids, :collection => Role.all.delete_if {|i| i.name == 'admin'},
:as => :radio_buttons, :label_method => lambda {|t| t.name.titleize}, :label => false,
:item_wrapper_class => 'inline' %>
    </div>
    <div class="modal-footer">
      <%= f.submit "Change Plan", :class => "btn btn-primary" %>
      <a class="btn" data-dismiss="modal" href="#">Close</a>
    </div>
  <% end %>
</div>
```

We include this form in a div designated with the Twitter Bootstrap “modal” class. It will be displayed as an overlay when the user clicks the “Change plan” button.

We’re using helper methods provided by the simple_form gem. It looks as if we bind the 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.

The form has a section named “modal-header” that contains a link to close the modal window. We follow Twitter Bootstrap’s example and use HTML entity #215 (an “x” character) for the link.

The next section is named “modal-body” and it contains some of the most complex code we’ll use in this application.

We want to display a set of radio buttons that display a collection of all the subscription plans and, when selected, set a role id. A user’s subscription plan is encoded as a role id. Role ids are nested attributes of a User object and can be set as `user.role_ids`. So `:role_ids` is the first parameter we pass to the input field helper.

The input field helper wants a collection as the second parameter. We supply `Role.all` but we have to massage the list to remove the “admin” role. The Hash `delete_if` method is useful here. The block we supply to the `delete_if` method removes any role named “admin”.

We tell the input field helper we want to display the selection field as radio buttons. The `:item_wrapper_class => 'inline'` will display the radio buttons and labels horizontally.

The simple_form `label_method` parameter allows us to set a label for each radio button. If we didn't set the `label_method`, the radio buttons would be labeled with an integer. We need to obtain the name attribute of each role and then apply the `titleize` method to display titlecase. The `label_method` parameter does not take a block but we can use a programming construct called an anonymous function (a Ruby language "lambda") to manipulate the Role instance, obtaining the name attribute and making it titlecase, before passing it to the `label_method` parameter. This is a bit of advanced Ruby magic that is particularly useful here.

Confusingly, we set the `label` parameter to false. If we didn't do this, the entire field would be automatically labelled "Roles."

Finally, the form has a section named "modal-footer" that contains a submit button and another link to close the modal window.

Now we'll add another modal window containing a form for changing the credit card.

```
<div id="card-data" class="modal" style="display: none;">
  <%= simple_form_for resource, :as => resource_name, :url => update_card_path, :html =>
{:method => :put, :class => 'form-horizontal card_form' } do |f| %>
    <div class="modal-header">
      <a class="close" data-dismiss="modal">&#215;</a>
      <h3>Change Credit Card</h3>
    </div>
    <div class="modal-body">
      <div class="field">
        <%= label_tag :card_number, "Credit Card Number" %>
        <%= text_field_tag :card_number, nil, name: nil %>
      </div>
      <div class="field">
        <%= label_tag :card_code, "Card Security Code (CVV)" %>
        <%= text_field_tag :card_code, nil, name: nil %>
      </div>
      <div class="field">
        <%= label_tag :card_month, "Card Expiration" %>
        <%= select_month nil, {add_month_numbers: true}, {name: nil, id: "card_month"}%>
        <%= select_year nil, {start_year: Date.today.year, end_year:
Date.today.year+10}, {name: nil, id: "card_year"}%>
      </div>
      <%= f.hidden_field :name %>
      <%= f.hidden_field :email %>
      <%= f.hidden_field :stripe_token %>
    </div>
    <div class="modal-footer">
      <%= f.submit "Change Credit Card", :class => "btn btn-primary" %>
      <a class="btn" data-dismiss="modal" href="#">Close</a>
    </div>
  <% end %>
</div>
```

This form is very similar to the form on the registration page that is used when a new account is created.

Our JavaScript code will intercept the submission of this form, access the Stripe server to set a new credit card, and obtain a Stripe token before submitting the form to our server.

Here's the complete version of the file **app/views/devise/registrations/edit.html.erb**:

```
<% content_for :head do %>
  <%= tag :meta, :name => "stripe-key", :content => STRIPE_PUBLIC_KEY %>
<% end %>
<h2>Account</h2>
<div id="stripe_error" class="alert alert-error" style="display:none" ></div>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name), :html => { :method => :put, :class => 'form-vertical'
}) do |f| %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= f.input :name, :autofocus => true %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :autocomplete => "off", :hint => "leave it blank if you don't
want to change it", :required => false %>
  <%= f.input :password_confirmation, :required => false %>
  <%= f.input :current_password, :hint => "we need your current password to confirm your
changes", :required => true %>
  <%= f.button :submit, 'Update', :class => 'btn-primary' %>
<% end %>

<h3>Subscription Plan</h3>
<p>
  <%= @user.roles.first.name.titleize %>
  <a data-toggle="modal" href="#plan-options" class="btn btn-mini" type="button">Change
plan</a>
</p>

<h3>Card</h3>
<p>
  Using card ending with <%= @user.last_4_digits %>
  <a data-toggle="modal" href="#card-data" class="btn btn-mini" type="button">Change
card</a>
</p>

<h3>Cancel my account</h3>

<p>Unhappy? <%= link_to "Cancel my account", registration_path(resource_name), :data =>
{ :confirm => "Are you sure?" }, :method => :delete, :class => 'btn btn-mini' %></p>

<div id="plan-options" class="modal" style="display: none;">
  <%= simple_form_for resource, :as => resource_name, :url => update_plan_path, :html =>
{:method => :put, :class => 'form-horizontal' } do |f| %>
    <div class="modal-header">
      <a class="close" data-dismiss="modal">&#215;</a>
      <h3>Change Plan</h3>
    </div>
    <div class="modal-body">
      <%= f.input :role_ids, :collection => Role.all.delete_if {|i| i.name == 'admin'},
:as => :radio_buttons, :label_method => lambda {|t| t.name.titleize}, :label => false,
:item_wrapper_class => 'inline' %>
```

```
    </div>
    <div class="modal-footer">
      <%= f.submit "Change Plan", :class => "btn btn-primary" %>
      <a class="btn" data-dismiss="modal" href="#">Close</a>
    </div>
  <% end %>
</div>

<div id="card-data" class="modal" style="display: none;">
  <%= simple_form_for resource, :as => resource_name, :url => update_card_path, :html =>
{:method => :put, :class => 'form-horizontal card_form' } do |f| %>
    <div class="modal-header">
      <a class="close" data-dismiss="modal">&#215;</a>
      <h3>Change Credit Card</h3>
    </div>
    <div class="modal-body">
      <div class="field">
        <%= label_tag :card_number, "Credit Card Number" %>
        <%= text_field_tag :card_number, nil, name: nil %>
      </div>
      <div class="field">
        <%= label_tag :card_code, "Card Security Code (CVV)" %>
        <%= text_field_tag :card_code, nil, name: nil %>
      </div>
      <div class="field">
        <%= label_tag :card_month, "Card Expiration" %>
        <%= select_month nil, {add_month_numbers: true}, {name: nil, id: "card_month"}%>
        <%= select_year nil, {start_year: Date.today.year, end_year:
Date.today.year+10}, {name: nil, id: "card_year"}%>
      </div>
      <%= f.hidden_field :name %>
      <%= f.hidden_field :email %>
      <%= f.hidden_field :stripe_token %>
    </div>
    <div class="modal-footer">
      <%= f.submit "Change Credit Card", :class => "btn btn-primary" %>
      <a class="btn" data-dismiss="modal" href="#">Close</a>
    </div>
  <% end %>
</div>
```

There's a lot of new code in the Devise user edit view. To accommodate the "Change Plans" feature we will need to improve the registrations controller, the users controller, and the User model. First we will modify the User model to accommodate the "Cancel my account" feature.

Modify the User Model for Subscription Cancellations

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_destroy` callback so we can notify Stripe to cancel subscription billing.

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

```
class User < ActiveRecord::Base
  rolify
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # Setup accessible (or protected) attributes for your model
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me,
:stripe_token, :coupon
  attr_accessor :stripe_token, :coupon
  before_save :update_stripe
  before_destroy :cancel_subscription

  def update_stripe
    return if email.include?(ENV['ADMIN_EMAIL'])
    return if email.include?('@example.com') and not Rails.env.production?
    if customer_id.nil?
      if !stripe_token.present?
        raise "Stripe token not present. Can't create account."
      end
      if coupon.blank?
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name
        )
      else
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name,
          :coupon => coupon
        )
      end
    else
      customer = Stripe::Customer.retrieve(customer_id)
      if stripe_token.present?
        customer.card = stripe_token
      end
      customer.email = email
      customer.description = name
      customer.save
    end
    self.last_4_digits = customer.cards.data.first["last4"]
    self.customer_id = customer.id
    self.stripe_token = nil
```

```
  rescue Stripe::StripeError => e
    logger.error "Stripe Error: " + e.message
    errors.add :base, "#{e.message}."
    self.stripe_token = nil
    false
  end

  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
  rescue Stripe::StripeError => e
    logger.error "Stripe Error: " + e.message
    errors.add :base, "Unable to cancel your subscription. #{e.message}."
    false
  end

end
```

We've added a `cancel_subscription` method that is activated by the `before_destroy` callback. The `cancel_subscription` method only takes action if the user has a customer id. Example users created by the `db:seed` process don't have a customer id and the `destroy` action would fail for them which is why we make the `cancel_subscription` method conditional.

The method uses the Stripe API to obtain a Stripe customer object and then initiates a subscription cancellation call. We make sure the Stripe customer exists and has an active subscription before attempting to cancel the subscription. We return false to cancel the `destroy` action if Stripe returns an error.

Modify the User Model for Subscription Plan Changes

As implemented, either the user or the administrator can upgrade or downgrade a user's subscription plan and the change will be recorded in the application's roles datatable. However, we need to inform Stripe when a subscription plan changes. Stripe will pro-rate the plan cost and refund or bill the price difference (or optionally, wait until the next billing cycle to change the billed rate).

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

```
class User < ActiveRecord::Base
  rolify
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # Setup accessible (or protected) attributes for your model
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me,
:stripe_token, :coupon
  attr_accessor :stripe_token, :coupon
  before_save :update_stripe
  before_destroy :cancel_subscription

  def update_plan(role)
    self.role_ids = []
    self.add_role(role.name)
    unless customer_id.nil?
      customer = Stripe::Customer.retrieve(customer_id)
      customer.update_subscription(:plan => role.name)
    end
    true
  rescue Stripe::StripeError => e
    logger.error "Stripe Error: " + e.message
    errors.add :base, "Unable to update your subscription. #{e.message}."
    false
  end

  def update_stripe
    return if email.include?(ENV['ADMIN_EMAIL'])
    return if email.include?('@example.com') and not Rails.env.production?
    if customer_id.nil?
      if !stripe_token.present?
        raise "Stripe token not present. Can't create account."
      end
      if coupon.blank?
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name
        )
      else
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name,
          :coupon => coupon
```

```
      )
    end
  else
    customer = Stripe::Customer.retrieve(customer_id)
    if stripe_token.present?
      customer.card = stripe_token
    end
    customer.email = email
    customer.description = name
    customer.save
  end
  self.last_4_digits = customer.cards.data.first["last4"]
  self.customer_id = customer.id
  self.stripe_token = nil
rescue Stripe::StripeError => e
  logger.error "Stripe Error: " + e.message
  errors.add :base, "#{e.message}."
  self.stripe_token = nil
  false
end

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
rescue Stripe::StripeError => e
  logger.error "Stripe Error: " + e.message
  errors.add :base, "Unable to cancel your subscription. #{e.message}."
  false
end

end
```

We add the `update_plan` method for resetting the role. Rolify supports multiple roles (though we only use a single role in this application) and we must remove an existing role before adding a new role. Then we make sure the user has a Stripe customer account, obtain a customer object with a call to Stripe, and call the Stripe `update_subscription` method.

With these changes, the User model can update the Stripe customer account when a user or administrator changes a subscription plan.

Modify the Registrations Controller for Account Changes

The registrations controller handles requests when the user submits the edit form.

We will add `update_plan` and `update_card` actions to the registrations controller to accommodate the "Change Plans" and "Change Card" features.

It would be nice to keep the controller completely RESTful and not add any new actions, replacing the `update` method to handle submissions of three different forms (changing the plan, changing the card, and updating name/email/password). However, the `update` method supplied by Devise is quite complex, with logic that forces the user to enter the current password to authorize any changes. We don't want to ask the user to enter a password to change the card or plan, so we'll leave the `update` method untouched (for changing name/email/password) and add two new methods.

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

```
class RegistrationsController < Devise::RegistrationsController

  def new
    @plan = params[:plan]
    if @plan && ENV["ROLES"].include?(@plan) && @plan != "admin"
      super
    else
      redirect_to root_path, :notice => 'Please select a subscription plan below.'
    end
  end

  def update_plan
    @user = current_user
    role = Role.find(params[:user][:role_ids]) unless params[:user][:role_ids].nil?
    if @user.update_plan(role)
      redirect_to edit_user_registration_path, :notice => 'Updated plan.'
    else
      flash.alert = 'Unable to update plan.'
      render :edit
    end
  end

  def update_card
    @user = current_user
    @user.stripe_token = params[:user][:stripe_token]
    if @user.save
      redirect_to edit_user_registration_path, :notice => 'Updated card.'
    else
      flash.alert = 'Unable to update card.'
      render :edit
    end
  end

  private
  def build_resource(*args)
    super
    if params[:plan]
      resource.add_role(params[:plan])
    end
  end
end
```

Not every request will require a change of card or subscription plan. Sometimes the user will be submitting a form to change name, email address, or password. In that case, the inherited `update` method from the Devise registrations controller will be invoked.

The `update_plan` method applies changes to the `current_user` (note that changes by an administrator are accommodated elsewhere by the User controller). We identify the new role that is specified in the form and call the User `update_plan(role)` method. That method will make a request to Stripe to update the customer account. Stripe will automatically apply

refunds or additional charges as necessary. The User `update_plan(role)` method returns `true` on a successful update or throws an error if an attempt to reach the Stripe server fails. On a successful update we redirect to the edit page with a status message. On a failure we render the edit page, showing any errors with the `display_base_errors` view helper.

The `update_card` method relies on the JavaScript code in **app/assets/javascripts/registrations.js** to intercept the card data and submit it to the Stripe server to obtain the Stripe token. The process is very similar to creating a new subscription. Once we have obtained the Stripe token, we call `@user.save`. Just like when a subscription is created, the User model `update_stripe` method calls the Stripe API and makes a request that associates the new card with the customer record for future billing. On a successful update we redirect to the edit page with a status message. On a failure we render the edit page, showing any errors with the `display_base_errors` view helper (this is where a "Credit card declined" message will appear).

Add Routes for Account Changes

We've added two new actions to the registrations controller to accommodate the "Change Plans" and "Change Card" features. We need to add routes to invoke the new actions.

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

```
RailsStripeMembershipSaas::Application.routes.draw do
  get "content/gold"
  get "content/silver"
  get "content/platinum"
  authenticated :user do
    root :to => 'home#index'
  end
  root :to => "home#index"
  devise_for :users, :controllers => { :registrations => 'registrations' }
  devise_scope :user do
    put 'update_plan', :to => 'registrations#update_plan'
    put 'update_card', :to => 'registrations#update_card'
  end
  resources :users
end
```

The `devise_scope` method applies Devise.mappings so the new routes are associated with other routes specified by `devise_for`. 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 `update_plan` and `update_card` routes to respond to an HTTP "put" request (a form submission). This generates the routes that we use in our "Change Plans" and "Change Card" forms:

- update_plan_path
- update_card_path

Our "Change Plans" and "Change Card" features are now ready for the user.

We'll add one more feature. We want to allow the administrator to change a subscription plan for any user.

Modify the User Controller for Subscription Plan Changes

The users controller handles requests when an administrator changes the user's subscription plan.

We need to modify the `update` method in the users controller to make sure Stripe's customer records are updated when an administrator makes a change.

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

```
class UsersController < ApplicationController
  before_filter :authenticate_user!

  def index
    authorize! :index, @user, :message => 'Not authorized as an administrator.'
    @users = User.all
  end

  def show
    @user = User.find(params[:id])
  end

  def update
    authorize! :update, @user, :message => 'Not authorized as an administrator.'
    @user = User.find(params[:id])
    role = Role.find(params[:user][:role_ids]) unless params[:user][:role_ids].nil?
    params[:user] = params[:user].except(:role_ids)
    if @user.update_attributes(params[:user])
      @user.update_plan(role) unless role.nil?
      redirect_to users_path, :notice => "User updated."
    else
      redirect_to users_path, :alert => "Unable to update user."
    end
  end

  def destroy
    authorize! :destroy, @user, :message => 'Not authorized as an administrator.'
    user = User.find(params[:id])
    unless user == current_user
      user.destroy
      redirect_to users_path, :notice => "User deleted."
    else
      redirect_to users_path, :notice => "Can't delete yourself."
    end
  end
end
```

The `update` action is more complex than the actions we added to the registrations controller because we are updating all attributes with a single action. The `authorize!` uses CanCan to make sure only the administrator can invoke the update. We find the name of the requested role if a `params[:user][:role_ids]` parameter exists. We strip away the `params[:user][:role_ids]` parameter to avoid mass assignment errors. Then we call `update_attributes`. If the User object successfully saves the new attributes, we call the `User.update_plan` method to change the subscription plan in both the application role table and the remote Stripe customer account.

Now an administrator can change subscription plans and Stripe will update the customer account and apply refunds or additional charges as necessary.

Git Workflow

Commit your changes to git:

```
$ git add -A
$ git commit -m "accommodate changes to account"
$ git checkout master
$ git merge --squash change-acct
$ git commit -m "accommodate changes to account"
$ git branch -D change-acct
```

Chapter 21

Stripe Push Notifications

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 (push notifications) to communicate events to you (for details, see the Stripe webhooks documentation).

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 retrive the data from the Stripe server. It is best to 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. Both these approaches have merit. Instead, we'll take a different approach. We'll use Danny Whalen's stripe_event gem. With this gem, we don't need to add a new controller or action.

Get started by creating a new git branch for this feature:

```
$ git checkout -b webhooks
```

Stripe Event Gem

We already added the stripe_event gem when we set up our Gemfile. If not, add the gem to the Gemfile:

```
gem "stripe_event"
```

and run the `bundle install` command to install the required gem on your computer.

Mount the Engine

The stripe_event gem is a Rails engine, which is a miniature Rails application that can be added to an application. As a miniature Rails application, it has its own controller and routes hidden in the gem. The engine's functionality becomes available when you mount it in your routes file.

Modify **config/routes.rb** to mount the engine:

```
RailsStripeMembershipSaas::Application.routes.draw do
  mount StripeEvent::Engine => '/stripe'
  get "content/gold"
  get "content/silver"
  get "content/platinum"
  authenticated :user do
    root :to => 'home#index'
  end
  root :to => "home#index"
  devise_for :users, :controllers => { :registrations => 'registrations' }
  devise_scope :user do
    put 'update_plan', :to => 'registrations#update_plan'
    put 'update_card', :to => 'registrations#update_card'
  end
  resources :users
end
```

We've chosen to mount the engine so it responds to requests to http://localhost:3000/stripe. You can specify a different path. You'll supply this address when you set the webhooks URL in your Stripe account settings.

Note that the application name is used in the routes file. If you've changed the name of the application from "rails-stripe-membership-saas" you'll need to change the first line of the routes file.

Modify the Stripe Initializer

We need to specify what happens when the stripe_event engine receives a webhook request. Conveniently, we can specify this in the Stripe initializer file.

Modify the **config/initializers/stripe.rb** file:

```
Stripe.api_key = ENV["STRIPE_API_KEY"]
STRIPE_PUBLIC_KEY = ENV["STRIPE_PUBLIC_KEY"]

StripeEvent.setup do
  subscribe 'customer.subscription.deleted' do |event|
    user = User.find_by_customer_id(event.data.object.customer)
    user.expire
  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 stripe_event engine to respond when the application receives a webhook with a `customer.subscription.deleted` event. We use the Stripe event data to find the appropriate User instance. Then we call the `User.expire` method. We'll add the `expire` method to the User class in the next step.

The stripe_event gem readme handles other possibilities, including multiple event types. Refer to the Stripe API documentation 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.

Add an Expire Method to the User Model

Now that our application responds to webhook requests from Stripe, we must consider how to process an expired customer subscription.

There are several possible approaches to handling an expired subscription. One option is to keep the customer record and change the subscription plan to a role named "expired" or something similar. If we did that, we'd have to provide a form to allow an expired subscriber to log in, change a credit card, and update the subscription. It's easier to simply delete the account and expect the user to create a new account to re-subscribe.

When a user visits the website and cancels a subscription, we delete the customer record. Similarly, we could simply delete the customer record when a subscription expires. Instead we can encourage the user to resubscribe by notifying the user with an email message when the subscription has ended due to a payment failure. We'll add a custom `expire` method to our User class that will send an email message before deleting the user.

Modify the file **app/models/user.rb** to add an `expire` method:

```
class User < ActiveRecord::Base
  rolify
  # Include default devise modules. Others available are:
  # :token_authenticatable, :confirmable,
  # :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # Setup accessible (or protected) attributes for your model
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me,
:stripe_token, :coupon
  attr_accessor :stripe_token, :coupon
  before_save :update_stripe
  before_destroy :cancel_subscription

  def update_plan(role)
    self.role_ids = []
    self.add_role(role.name)
    unless customer_id.nil?
      customer = Stripe::Customer.retrieve(customer_id)
      customer.update_subscription(:plan => role.name)
    end
    true
  rescue Stripe::StripeError => e
    logger.error "Stripe Error: " + e.message
    errors.add :base, "Unable to update your subscription. #{e.message}."
    false
  end

  def update_stripe
    return if email.include?(ENV['ADMIN_EMAIL'])
    return if email.include?('@example.com') and not Rails.env.production?
    if customer_id.nil?
      if !stripe_token.present?
        raise "Stripe token not present. Can't create account."
      end
      if coupon.blank?
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name
        )
      else
        customer = Stripe::Customer.create(
          :email => email,
          :description => name,
          :card => stripe_token,
          :plan => roles.first.name,
          :coupon => coupon
```

```
      )
    end
  else
    customer = Stripe::Customer.retrieve(customer_id)
    if stripe_token.present?
      customer.card = stripe_token
    end
    customer.email = email
    customer.description = name
    customer.save
  end
  self.last_4_digits = customer.cards.data.first["last4"]
  self.customer_id = customer.id
  self.stripe_token = nil
rescue Stripe::StripeError => e
  logger.error "Stripe Error: " + e.message
  errors.add :base, "#{e.message}."
  self.stripe_token = nil
  false
end

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
rescue Stripe::StripeError => e
  logger.error "Stripe Error: " + e.message
  errors.add :base, "Unable to cancel your subscription. #{e.message}."
  false
end

def expire
  UserMailer.expire_email(self).deliver
  destroy
end

end
```

Our new `expire` method is simple. We call a method on an ActionMailer method to send an email. Then we destroy the user.

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 => "notifications@example.com"

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

Replace the "notifications@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 the User model is equipped to send an email message when a Stripe webhook notifies the application of a cancelled subscription.

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 and add a URL such as https://example.com/stripe.

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/stripe. You should see:

```
Started GET "/stripe" 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 can deploy the application so it is running on a production web server and then use the "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/stripe.rb** file:

```
subscribe do |event|
  Rails.logger.info event
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 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 (localtunnel on GitHub) which will expose your local web server to the Internet so you can receive webhook requests from the Stripe servers.

Git Workflow

Commit your changes to git:

```
$ git add -A
$ git commit -m "add Stripe webhooks"
$ git checkout master
$ git merge --squash webhooks
$ git commit -m "add Stripe webhooks"
$ git branch -D webhooks
```

Chapter 22

Customize, Test, and Deploy

Additional Features

You've created a fully functional membership site that's ready to take credit card payments and serve access to restricted content.

It uses Stripe for payment processing, allows users to change credit cards or subscription plans, and uses Stripe webhooks to delete users who have expired or declined credit cards.

You might consider a few enhancements. For example, you might want the application to respond to a Stripe webhook event when a credit card payment is successful by sending a "paid" invoice. Or you might respond to Stripe webhook events when credit card payments are unsuccessful by sending a friendly note encouraging the user to check for an expired credit card before the subscription is cancelled.

If you have suggestions for additional features, please create an issue on GitHub.

Cleanup

Several unneeded files are generated in the process of creating a new Rails application.

Additionally, you may want to prevent search engines from indexing your website if you've deployed it publicly while still in development.

See instructions for cleaning up unneeded files in Rails and banning spiders.

Test the App

You can check that your app runs properly by entering the command:

```
$ rails server
```

To see your application in action, open a browser window and navigate to http://localhost:3000/.

Sign in as the first user (the administrator) using:

- email: user@example.com
- password: changeme

You'll see a navigation link for Admin. Clicking the link will display a page with a list of users at
http://localhost:3000/users.

To sign in as the second user, use

- email: user2@example.com
- password: changeme

The second user will not see the Admin navigation link and will not be able to access the page at
http://localhost:3000/users.

You should be able to create additional users. If you use "example.com" in an email address in development or testing, the application will not connect with Stripe. Use a different email address with a fake credit card number to test subscribing using Stripe. You'll see the new users listed when you log in as an administrator. And you'll see the new users listed as customers when you visit your Stripe dashboard.

Stop the server with Control-C.

Testing

If you've copied the RSpec unit tests and Cucumber integration tests from the rails-stripe-membership-saas example application, you can run `rake -T` to check that rake tasks for RSpec and Cucumber are available.

Run `rake spec` to run all RSpec tests.

Run `rake cucumber` (or more simply, `cucumber`) to run all Cucumber scenarios.

Deploy to Heroku

For your convenience, here is a Tutorial for Rails on Heroku. Heroku provides low cost, easily configured Rails application hosting.

Be sure to set up SSL before you make your application available in production. See the Heroku documentation on SSL.

Prior to deployment, change your **db/seeds.rb** file. Remove the "example.com" sample users.

```
puts 'ROLES'
YAML.load(ENV['ROLES']).each do |role|
  Role.find_or_create_by_name({ :name => role }, :without_protection => true)
  puts 'role: ' << role
end
puts 'DEFAULT USERS'
user = User.find_or_create_by_email :name => ENV['ADMIN_NAME'].dup, :email =>
ENV['ADMIN_EMAIL'].dup, :password => ENV['ADMIN_PASSWORD'].dup, :password_confirmation
=> ENV['ADMIN_PASSWORD'].dup
puts 'user: ' << user.name
user.add_role :admin
```

You'll need to set the configuration values from the **config/application.yml** file as Heroku environment variables. See the article Rails Environment Variables for more information.

With the figaro gem, just run:

```
$ rake figaro:heroku
```

Alternatively, you can set Heroku environment variables directly with `heroku config:add`.

```
$ heroku config:add GMAIL_USERNAME='myname@gmail.com' GMAIL_PASSWORD='secret'
$ heroku config:add 'ROLES=[admin, silver, gold, platinum]'
$ heroku config:add ADMIN_NAME='First User' ADMIN_EMAIL='user@example.com'
ADMIN_PASSWORD='changeme'
$ heroku config:add STRIPE_API_KEY=secret STRIPE_PUBLIC_KEY=secret
```

If you don't remove the "example.com" sample users, `rake db:seed` will fail with errors if you attempt to run it on Heroku after deployment, since you are not supplying a credit card for Stripe for these users.

Prepare your application assets for Heroku.

Add this configuration parameter to the **config/application.rb** file:

```
# Heroku requires this to be false
config.assets.initialize_on_precompile=false
```

Then precompile assets, commit to git, and push to Heroku:

```
$ rake assets:precompile
$ git add -A
$ git commit -m "assets compiled for Heroku"
$ git push heroku master
```

Complete Heroku deployment with:

```
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

See the Tutorial for Rails on Heroku for details.

Chapter 23

Comments

Credits

Daniel Kehoe implemented the application and wrote the tutorial.

Did You Like the Tutorial?

Was this useful to you? Follow rails_apps on Twitter and tweet some praise. I'd love to know you were helped out by the tutorial.

Get some link juice! Add your website to the list of Rails Applications Built from the Examples. I love to see what people have built with these examples.

Any issues? Please create an issue on GitHub. Reporting (and patching!) issues helps everyone.

Membership Site with Stripe