



*RailsApps Project*

# Subscription Site with Recurly

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

# Contents

1.	Introduction .....	3
2.	Architecture and Implementation .....	9
3.	Accounts You May Need .....	12
4.	Getting Started .....	17
5.	Create the Application .....	18
6.	Test-Driven Development .....	24
7.	Configuration .....	26
8.	Layout and Stylesheets .....	30
9.	Authentication .....	31
10.	Authorization .....	32
11.	User Management .....	33
12.	Initial Data .....	36
13.	Home Page .....	40
14.	Administrative Page .....	44
15.	Content Pages .....	45
16.	Limit Access to Content .....	49
17.	Registration Page .....	53
18.	Redirect After Sign Up or Log In .....	60
19.	Recurly Integration .....	62
20.	Account Changes .....	86
21.	Recurly Push Notifications .....	101
22.	Customize, Test, and Deploy .....	113
23.	Comments .....	117

## Chapter 1

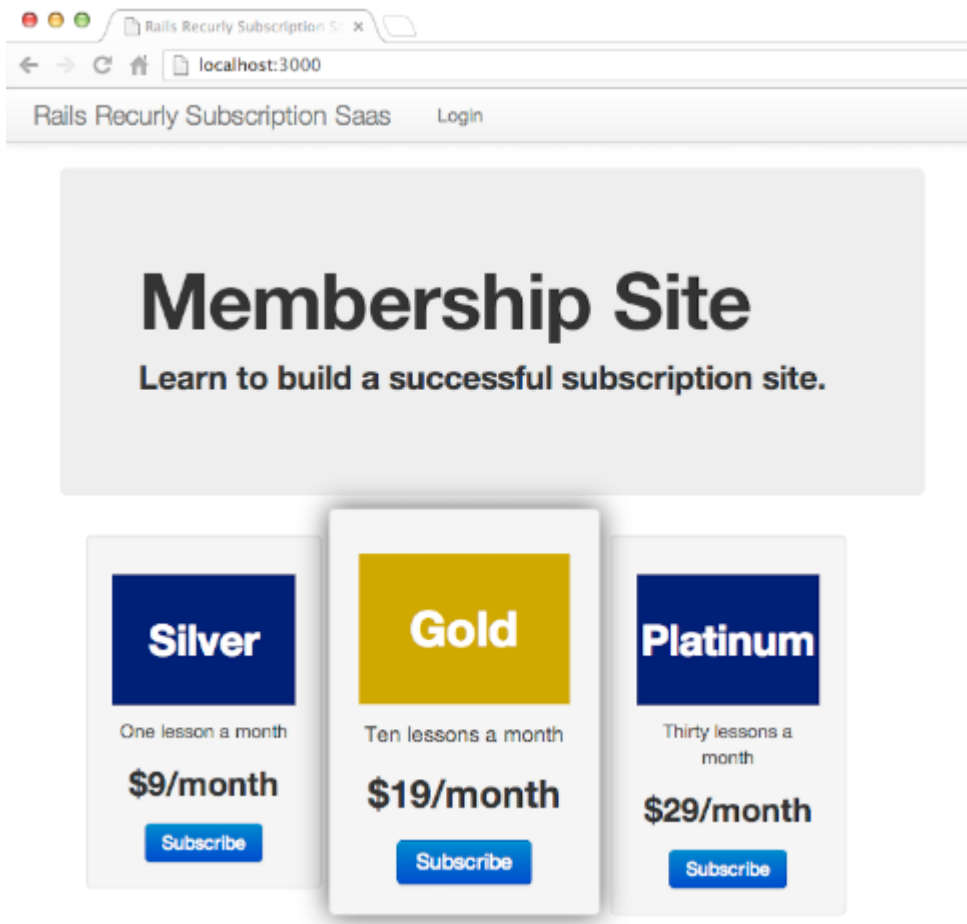
# Introduction

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

- [Recurly](#) 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 Recurly
- uses Recurly for no local credit card storage
- Recurly accepts credit card payments from customers in any country or currency
- PCI compliance using the Recurly JavaScript library
- Recurly 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
- [Recurly](#) 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.

## Recurly

Recurly 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; for example, [Stripe](#) is popular and may be the least expensive and most popular for low-volume startups. (See the [rails-stripe-membership-saas](#) application from the RailsApps project if you want to use Stripe.) 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. (Are there other reasons to use Recurly versus Stripe? [Leave a comment.](#))

## 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 Recurly
- 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 Recurly with user management provided by Devise for our account management system.

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

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

We'll provide options for the user to upgrade or downgrade subscription plans and cancel a subscription as an extension of the user management system provided by Devise. We'll provide a link to Recurly's subscription interface for the user to change credit cards.

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 Recurly when a user's account comes due. With this approach, you would use Recurly only for processing credit card transactions. But there's no reason to implement recurring billing yourself. Recurly provides a complete, well-tested, and hosted mechanism for recurring billing. We'll use Recurly'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 Recurly, 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 Recurly encounters a declined transaction. Recurly provides "webhooks" to set the status of a subscription. When Recurly 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 Recurly webhooks, we'd have to either query the Recurly API on each login or run a repeating cron job to check for subscription expiration. The application will be notified immediately by Recurly so there is no need for the overhead of checking on each login. The Recurly 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 Recurly, your server will never receive sensitive credit card details. Instead you'll use the Recurly JavaScript library on your subscription payment form which sends the credit card details directly to the Recurly 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 Recurly 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 Recurly and set a unique customer id that we'll use to identify the subscriber in both our application and the Recurly billing system.

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*. You should also consider how you will implement SSL for security.

## 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 competitive, 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. Unlike Stripe, for example, Recurly's dunning management feature sends emails to the customer when the customer's credit card expires to encourage continuing the subscription.

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

This tutorial shows to set up recurring billing using Recurly. Before you start, go to the [Recurly website](#) and set up an account.

## 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 Stripe, Recurly does not include a merchant account as part of the service. If your business is established and already taking credit card payments, you will already have a merchant account. If not, you will have to obtain a merchant account. If you have a U.S.-based business, Recurly will refer you to their partner, [TSYS Merchant Solutions](#), one of the largest credit card processors in the U.S., and a salesperson will contact you to send you an application form. It takes a few days to get approval so get the process started early.

Pricing and contract terms for merchant accounts can be complicated, confusing, and often misleading. [CardPaymentOptions.com](#) offers an ebook and reviews of credit card processors to help you compare offers. I was pleased to see TSYS Merchant Solutions offers favorable terms for Recurly customers. Here are the terms I received from TSYS Merchant Solutions for the RailsApps merchant account:

- no cancellation fee
- month-to-month contract

- setup fee \$0
- monthly minimum \$0
- monthly maintenance fee \$0
- annual fee \$0
- monthly regulatory and compliance fee \$5
- Visa/MC/Discover Interchange Plus rate .30%
- authorization fee \$0.08
- batch fee \$0
- monthly PCI compliance \$6.50
- Amex rate 3.50%

Add the average [interchange fee](#) charged by banks:

- interchange fee 1.79%

Add Recurly's fees:

- 1.25% + \$0.10/transaction
- \$69/monthly minimum

And you can see a total for the merchant account and Recurly:

- 3.34% + \$0.18/transaction
- \$11.50/month fees
- \$69/monthly minimum

Compare to Stripe:

- 2.9% + \$0.30/transaction
- no monthly minimum
- no merchant account needed

Again, you should carefully analyze costs based on your projected revenue before choosing either Stripe or Recurly. Stripe's low percentage fee is offset by a high per-transaction fee and Recurly's high percentage fee is offset by a low per-transaction fee.

## 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 Recurly's servers when they sign up for a subscription. The Recurly JavaScript library will open an SSL connection to Recurly'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 Recurly's servers. However, your security-conscious visitors will be uneasy if they see that the web URL for your registration page begins with `http://` and not `https://` (indicating an SSL connection). For their peace of mind (and the higher conversion rate that comes with trust), you should host your website with an SSL connection. Additionally, as a general practice, it is wise to host any 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 Cloudflare with their web browsers, providing a complete SSL connection through Cloudflare to Heroku with your custom domain. Not only do you get SSL for no more than you'd pay at Heroku to use an SSL certificate, but you get the Cloudflare CDN services as part of the bargain.

If you're deploying on Heroku, you can wait until you've deployed to sign up for a Cloudflare account.

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-recurly-subscription-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-recurly-subscription-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-recurlly-subscription-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-recurlly-subscription-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-recurlly-subscription-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-recurlly-subscription-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 "cancanc", ">= 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-recurly-subscription-saas application:

```
gem 'recurly', '>= 2.1.5'
gem 'nokogiri', '>= 1.5.5'
gem 'countries', '>= 0.8.4'
gem 'httpi', '>= 1.1.1'
gem 'httpclient', '>= 2.3.0.1'
```

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 Rails default information page.

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-recurly-subscription-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_FIRST_NAME: First
ADMIN_LAST_NAME: User
ADMIN_EMAIL: user@example.com
ADMIN_PASSWORD: changeme
ROLES: [admin, silver, gold, platinum]
RECURLY_API_KEY: = recurly_api_key
RECURLY_JS_PRIVATE_KEY: = recurly_js_private_key
RECURLY_SUBDOMAIN: myapp
```

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 Recurly gem requires an API key to operate. We'll also need to supply a private key so the Recurly gem can generate a cryptographic signature. You can find both keys on your Recurly account profile under "API Credentials".

We'll also set a `RECURLY_SUBDOMAIN` environment variable that we'll use in the JavaScript code that initiates the Recurly transaction. Recurly creates a customer-facing website and API interface with a subdomain on the Recurly website. The URL will look like this:

`https://myapp.recurly.com/` where "myapp" is the name of your company or service. You'll use the subdomain "myapp" to form the URL that makes requests to the Recurly API interface. The Recurly subdomain is listed on your Recurly account profile.

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.

## Recurly initializer

You've set the Recurly API key and private 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 data.

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

```
Recurly.api_key = ENV['RECURLY_API_KEY']
Recurly.js.private_key = ENV['RECURLY_JS_PRIVATE_KEY']
RECURLY_SUBDOMAIN = ENV['RECURLY_SUBDOMAIN']
# Recurly.default_currency = 'USD'
```

If you're not operating with US dollars as your currency, you can change the default in the initializer file.

We are using local environment variables from the **config/application.yml** file. The `Recurly.api_key` and `Recurly.js.private_key` values should be kept secret so no other developer can use your Recurly account. The `RECURLY_SUBDOMAIN` 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

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 adds navigation links, includes flash messages for errors and notifications, and applies CSS styling using Twitter Bootstrap.

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

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

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](#).

## 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 the 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 Recurly

Recurly customer accounts require both a first name and last name.

We’ll also need a customer id that will be used to match our application users with Recurly customer accounts.

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 ChangesForRecurlyToUsers
```

Modify the resulting file **db/migrate/xxxxxxx\_\_changes\_for\_recurly\_to\_users.rb**:

```
class ChangesForRecurlyToUsers < ActiveRecord::Migration
  def change
    change_table :users do |t|
      t.rename :name, :first_name
      t.string :last_name
      t.string :customer_id
    end
  end
end
```

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

```
$ rake db:migrate
```

Now we'll modify the User model.

## Modify the User Model

We'll add new attributes to the User model to accommodate Recurly 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 :first_name, :last_name, :email, :password, :password_confirmation,
                 :remember_me, :customer_id

  def name
    name = "#{first_name.capitalize} #{last_name.capitalize}"
  end
end
```

First 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.

We add `first_name`, `last_name`, and `customer_id` to the `attr_accessible` method so we can set these attributes from our registration form.

We've also defined a method that concatenates the `first_name` and `last_name` to return a full name.

With these changes 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 file **db/seeds.rb** already contains:

```
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
```

Replace **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 :first_name => ENV['ADMIN_FIRST_NAME'].dup,
:last_name => ENV['ADMIN_LAST_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 :first_name => 'Silver', :last_name => 'User',
:email => 'user2@example.com', :password => 'changeme', :password_confirmation =>
'changeme'
user2.add_role :silver
user3 = User.find_or_create_by_email :first_name => 'Gold', :last_name => 'User',
:email => 'user3@example.com', :password => 'changeme', :password_confirmation =>
'changeme'
user3.add_role :gold
user4 = User.find_or_create_by_email :first_name => 'Platinum', :last_name => '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.

Next we add an administrator and three sample users. Our User model now has separate first and last names to accommodate Recurly's customer schema.

We've added a user with an administrator role using values 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.

## 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 subscription 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** 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 familiar `index`, `show`, etc. methods of a RESTful controller. This is a case where a RESTful controller is not needed or appropriate. By default, the controller will render a view corresponding to each action.

# Check the Content Views

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

## **app/views/content/silver.html.erb**

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

## **app/views/content/gold.html.erb**

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

## **app/views/content/platinum.html.erb**

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

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

# 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 Recurly 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 Recurly Subscription 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” access. 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>
    <%= f.error_notification %>
    <%= display_base_errors resource %>
    <%= hidden_field_tag 'plan', params[:plan] %>
    <%= f.hidden_field :customer_id %>
    <%= f.input :first_name, :autofocus => true %>
    <%= f.input :last_name %>
    <%= 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 %>
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 also include the `customer_id` as a hidden field.

We add input fields for Recurly’s required `first_name` and `last_name`.

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 Recurly 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

# Recurly 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 Recurly 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, we can implement an architecture that cuts risk and makes it much easier to meet PCI compliance requirements.

Recurly offers the [Recurly.js JavaScript library](#) which generates a subscription form dynamically. The JavaScript library submits the credit card data directly to the Recurly servers, bypassing your server completely. Unfortunately, the Recurly.js JavaScript library is not well-suited for integration with a Rails application. It is standalone code that only provides what is needed to create a Recurly subscription. To be used with our application, we'd have to implement a complicated two-step registration process that uses a typical Rails form to create an account (with user name and password) and a separate JavaScript-based form for subscription payment. As Rails developers, we'd like to use the native Rails form helpers to build a single form that registers a new user and collects subscription payment information. To do so, we'll borrow some code from the Recurly.js JavaScript library and create our own `recurly.js` script so we can send billing information to Recurly securely while processing a new user registration on our own server.

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 Recurly's servers
- Recurly's servers create a customer record and process a credit card transaction
- a JavaScript callback receives a Recurly token or an error message
- the JavaScript function submits the form to our server
- a User model `before_create` method checks that a Recurly customer account exists
- a Registrations controller `create` action creates a new user

This hybrid approach makes implementation more complex but the result is robust and secure. Recurly takes responsibility for the security of the credit card data and it never touches our server.

This approach is not original. It is very similar to the implementation used in the [rails-stripe-membership-saas](#) example application which, in turn, was inspired by Ryan Bates's implementation from his RailsCast [Billing with Stripe](#). Thank you, Ryan!

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

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

## Prepare Your Recurly Account

Before we can submit a billing request to Recurly, we have to set up our subscription plans in Recurly.

We'll tell Recurly 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, Recurly will do all the work of notifying the user, initiating monthly billing, and contacting the user when a credit card is declined or expires.

Go to your Recurly account profile to create a subscription plan. Look for "Configuration/ Subscription Plans." Recurly offers [documentation about creating subscriptions](#) and [additional detail about subscriptions](#).

Create three different plans with the following values:

Plan Name	Plan Code	Pricing	Interval
-----------	-----------	---------	----------

Silver	silver	9.00	monthly
Gold	gold	19.00	monthly
Platinum	platinum	29.00	monthly

“Plan Name” is displayed on invoices and in the Recurly web interface. “Plan Code” is a unique string of your choice that is used to identify the plan when subscribing a customer. The “Plan Code” should correspond to a role we’ve created to manage access. “Pricing” is the subscription price. You’ll specify the billing frequency. Optionally, you can specify a free trial period. 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 Recurly knows about our subscription plans, we’ll add attributes to our User model.

## Add a Virtual Attribute 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 Recurly. A successful transaction will return a Recurly token. The Recurly 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](#).

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 :first_name, :last_name, :email, :password, :password_confirmation,
:remember_me, :customer_id, :card_token
  attr_accessor :card_token

  def name
    name = "#{first_name.capitalize} #{last_name.capitalize}"
  end
end

```

We add the virtual attribute with `attr_accessor` which will automatically create getter and setter methods and store the `card_token` value as an instance variable. Also we add the new `card_token` field to the `attr_accessible` list so that it can be set through mass assignment.

With the virtual attribute in place, we will modify the Registrations controller and add fields for credit card data to the form.

## Modify the Registrations Controller

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"
      @signature = Recurly.js.sign :subscription => { :plan_code => @plan }
      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
    resource.customer_id ||= SecureRandom.uuid
  end
end

```

We add a statement to each method.

For the `new` method, we add:

```
@signature = Recurly.js.sign :subscription => { :plan_code => @plan }
```

For the `new` method, if the *plan* parameter is present, we obtain a Recurly cryptographic signature and then call the `super` method to inherit the original controller `new` action.

Recurly requires a [cryptographic signature](#) to verify information sent to Recurly from the visitor's browser. The signature validates that parameters submitted from the form have not been tampered with before being received by Recurly. The Recurly gem provides a method `Recurly.js.sign` to generate the cryptographic signature. The `Recurly.js.sign` method [takes various parameters](#) such as subscription, account, or transaction; in this case, we are creating a subscription so we pass the *plan* parameter to the `Recurly.js.sign` method. The resulting cryptographic signature contains an authentication code combined with the encoded *plan* parameter. When the Recurly servers parse the form submission, the authentication code guarantees the authenticity of the message and the encoded plan parameter should match the *plan* parameter submitted via the form.

For the `build_resource` method, we add:

```
resource.customer_id ||= SecureRandom.uuid
```

We set a customer id for the User object. The customer id is essential, as it allows us to maintain concurrency between Recurly's customer records and our own. Recurly wants us to assign a unique identifier when we create a customer record. As a Rails ActiveRecord object, our User model will get a unique object id when the database record is created. But we are creating a Recurly customer record before we create the ActiveRecord object so the ActiveRecord object id is not yet available. We surmount this obstacle by using the Ruby [SecureRandom.uuid](#) method to generate a unique identifier. This will be our customer id for both Recurly's customer records and our own.

## 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 Recurly server.

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

```

<h2>Sign up</h2>
<div id="card_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',
  'data-subdomain' => RECURLY_SUBDOMAIN, 'data-signature' => @signature, 'data-ip_address'
=> request.remote_ip}) do |f| %>
  <h3><%= params[:plan].titleize if params[:plan] %> Subscription Plan</h3>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= hidden_field_tag 'plan', params[:plan] %>
  <%= f.hidden_field :customer_id %>
  <%= f.input :first_name, :autofocus => true %>
  <%= f.input :last_name %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :required => true %>
  <%= f.input :password_confirmation, :required => true %>
  <% if @user.card_token %>
    <p>Credit card acceptance is pending.</p>
  <% else %>
    <div class="field">
      <%= label_tag :country, "Country" %>
      <%= select_tag 'country', country_options_for_select(selected = 'US',
priority_countries = 'US') %>
    </div>
    <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">
      <%= label_tag :coupon, "Promotional Coupon (if any)" %>
      <%= text_field_tag :coupon, nil, name: nil %>
    </div>
  <% end %>
  <%= f.hidden_field :card_token %>
  <%= f.button :submit, 'Sign up', :class => 'btn-primary' %>
<% end %>

```

Take a look at `div id="card_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 Recurly 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"
data-subdomain="..." data-signature="..." data-ip_address="..." 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 name “user” and action name “new”.

We use an HTML5 technique for embedding data within the page that will be used by our JavaScript code. We set form tag attributes `data-subdomain`, `data-signature`, and `data-ip_address`. The `data-subdomain` value is set with the `RECURLY_SUBDOMAIN` constant that we set in our **config/initializers/recurly.rb** file (which was originally set in the **config/application.yml** file). Recurly needs the user's IP address (for credit card fraud prevention) and a cryptographic signature (to guarantee the form data is tamperproof). The IP address is obtained from the Rails `request` object and the cryptographic signature is generated in the Registrations controller `new` action.

We also pass the `customer_id` attribute as a hidden field. This will be used by the JavaScript library when the credit card data is submitted to Recurly to create a customer record. It will also be used to set the User model `customer_id` attribute when our application saves the record in our database.

We've added fields for credit card data but we've made inclusion of the fields conditional on the absence of the `@user.card_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 Recurly accepts the credit card data and provides a `card_token` but there's no email address, we'll save the Recurly token in a 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 `first_name`, `last_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.

It's not strictly necessary to provide the customer's IP address and country to Recurly but it improves credit card fraud detection. We use a `country_select` form helper from Josh Robinson's [countries gem](#) to provide a list of countries that returns an ISO two digit country code. "US" is set as the selected country but you can change this as necessary.

Now we'll add JavaScript to submit credit card data from the form to the Recurly server.

## Add the Recurly JavaScript Library

You can learn more about the [Recurly JavaScript library](#) (with examples), see the [Recurly JavaScript library documentation](#), or download the code from the [Recurly.js GitHub repository](#). The Recurly JavaScript library is intended to be used to generate a form dynamically using JavaScript. We've hacked the code to create our own *recurly.js* script that allows us to submit a billing request to the Recurly servers from our own Rails-based form. The hacked code is not elegant. You may wish to rewrite it using CoffeeScript (if you do, please contribute it to the project).

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

```

// Recurly.js
// JavaScript library for the Recurly API
// adapted for the rails-recurly-subscription-saas example application
// from https://github.com/recurly/recurly-js

function createObject(o) {
  function F() {}
  F.prototype = o || this;
  return new F();
};

var Recurly = {};

$('.registrations.new').ready(function() {

  Recurly.settings = {
    enableGeoIP: true
  , acceptedCards: ['american_express', 'discover', 'mastercard', 'visa']
  , oneErrorPerField: true
  , baseURL: 'https://api.recurly.com/jsonp/' + $('#new_user').data('subdomain') + '/'
  };

  Recurly.version = '2.1.3';

  Recurly.ajax = function(options) {
    options.data = $.extend({js_version: Recurly.version}, options.data);
    return $.ajax(options);
  };

  Recurly.flattenErrors = function(obj, attr) {
    var arr = [];
    var attr = attr || '';
    if( typeof obj == 'string'
    || typeof obj == 'number'
    || typeof obj == 'boolean') {
      if (attr == 'base') {
        return [obj];
      }
      return ['' + attr + ' ' + obj];
    }
    for(var k in obj) {
      if(obj.hasOwnProperty(k)) {
        // Inherit parent attribute names when property key
        // is a numeric string; how we deal with arrays
        attr = (parseInt(k).toString() == k) ? attr : k;
        var children = Recurly.flattenErrors(obj[k], attr);
        for(var i=0, l=children.length; i < l; ++i) {
          arr.push(children[i]);
        }
      }
    }
  }
}

```

```

    return arr;
};

Recurly.Account = {
  create: createObject
, toJSON: function(card) {
    return {
      first_name: card.first_name
    , last_name: card.last_name
    , account_code: card.customer_id
    , email: card.email
    };
  }
};

Recurly.BillingInfo = {
  create: createObject
, toJSON: function(card) {
    return {
      first_name: card.first_name
    , last_name: card.last_name
    , month: card.expMonth
    , year: card.expYear
    , number: card.number
    , verification_value: card.cvc
    , country: card.country
    , ip_address: card.ip_address
    };
  }
};

Recurly.Subscription = {
  create: createObject
, plan: Recurly.Plan
, addOns: []

, calculateTotals: function() {
    var totals = {
      stages: {}
    };

    // PLAN
    totals.plan = this.plan.cost.mult(this.plan.quantity);

    // ADD-ONS
    totals.allAddOns = new Recurly.Cost(0);
    totals.addOns = {};
    for(var l=this.addOns.length, i=0; i < l; ++i) {
      var a = this.addOns[i],
          c = a.cost.mult(a.quantity);
    }
  }
};

```



```

    totals.addOns[a.code] = c;
    totals.allAddOns = totals.allAddOns.add(c);
}

totals.stages.recurring = totals.plan.add(totals.allAddOns);

totals.stages.now = totals.plan.add(totals.allAddOns);

// FREE TRIAL
if(this.plan.trial) {
    totals.stages.now = Recurly.Cost.FREE;
}

// COUPON
if(this.coupon) {
    var beforeDiscount = totals.stages.now;
    var afterDiscount = totals.stages.now.discount(this.coupon);
    totals.coupon = afterDiscount.sub(beforeDiscount);
    totals.stages.now = afterDiscount;
}

// SETUP FEE
if(this.plan.setupFee) {
    totals.stages.now = totals.stages.now.add(this.plan.setupFee);
}

// VAT
if(this.billingInfo &&
Recurly.isVATChargeApplicable(this.billingInfo.country, this.billingInfo.vatNumber)) {
    totals.vat = totals.stages.now.mult( (Recurly.settings.VATPercent/100) );
    totals.stages.now = totals.stages.now.add(totals.vat);
}

return totals;
}

, redeemAddOn: function(addOn) {
var redemption = addOn.createRedemption();
this.addOns.push(redemption);
return redemption;
}

, removeAddOn: function(code) {
for(var a=this.addOns, l=a.length, i=0; i < l; ++i) {
    if(a[i].code == code) {
        return a.splice(i,1);
    }
}
}

, findAddOnByCode: function(code) {
    for(var l=this.addOns.length, i=0; i < l; ++i) {

```

```

        if(this.addOns[i].code == code) {
            return this.addOns[i];
        }
    }
    return false;
}

, toJSON: function(plan, coupon) {
    var json = {
        plan_code: plan.plan_code
        , quantity: plan.quantity ? plan.quantity : 1
        , currency: plan.currency ? plan.currency : 'USD'
        , coupon_code: coupon.coupon_code ? coupon.coupon_code : undefined
        , add_ons: []
    };

    for(var i=0, l=this.addOns.length, a=json.add_ons, b=this.addOns; i < l; ++i) {
        a.push({
            add_on_code: b[i].code
            , quantity: b[i].quantity
        });
    }

    return json;
}

, save: function(signature, plan, coupon, card, callback) {
    var json = {
        subscription: this.toJSON(plan, coupon)
        , account: Recurly.Account.toJSON(card)
        , billing_info: Recurly.BillingInfo.toJSON(card)
        , signature: signature
    };

    Recurly.ajax({
        url: Recurly.settings.baseURL+'subscribe',
        data: json,
        dataType: "jsonp",
        jsonp: "callback",
        timeout: 60000,
        success: function(response){
            callback(response)
        },
        error: function() {
            console.log(['Unknown error processing transaction. Please try again later.']);
        }
    });
}

```

```
};
});
```

The code is idiosyncratic and can be improved. The best that can be said is that it works. Please feel free to improve it and contribute back to the project.

## Script for Conditional Execution of Page-Specific JavaScript

We'll use a technique described in the article [Adding JavaScript to Rails](#) to only execute our JavaScript library on the registration page. We'll add a script named *jquery.readystatechange.js* (from [John Firebaugh](#)) that allows us to execute JavaScript conditionally on a page.

First, make sure that the application layout file is set to provide parameters needed for the *jquery.readystatechange.js* script. The default RailsApps application layout file 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.readystatechange.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.readys selector.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.readys selector
```

## 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.");
});
```

This code will test if JavaScript is executing conditionally on a page.

Later we will add the code that collects the credit card data from the subscription form and submits it to the Recurly 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.

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 Recurly and obtain a Recurly token before the form is sent to our server.

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

```

$('.registrations.new').ready(function() {
  var signature = $('#new_user').data('signature')
  var ip_address = $('#new_user').data('ip_address')
  var subscription = {
    setupForm: function() {
      console.log('function: setupForm')
      return $('#card_form').submit(function() {
        console.log('setupForm: form submitted')
        $('#input[type=submit]').prop('disabled', true);
        if ($('#card_number').length) {
          subscription.processCard();
          return false;
        } else {
          return true;
        }
      });
    },
    processCard: function() {
      console.log('function: processCard');
      var plan;
      plan = {
        plan_code: $('#plan').val(),
      };
      var coupon;
      coupon = {
        coupon_code: $('#coupon').val(),
      };
      var card;
      card = {
        customer_id: $('#user_customer_id').val(),
        email: $('#user_email').val(),
        first_name: $('#user_first_name').val(),
        last_name: $('#user_last_name').val(),
        number: $('#card_number').val(),
        cvc: $('#card_code').val(),
        expMonth: $('#card_month').val(),
        expYear: $('#card_year').val(),
        country: $('#country').val(),
        ip_address: ip_address
      };
      return Recurly.Subscription.save(signature, plan, coupon, card,
subscription.handleResponse);
    },
    handleResponse: function(response) {
      console.log('function: handleResponse');
      if(response.success) {
        return alert('Recurly response: ' + response.success.token);
      }
      else if(response.errors) {

```

```

        return alert('Recurly response: ' + Recurly.flattenErrors(response.errors));
    }
}
};
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 Recurly 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.

Recurly provides a set of fake credit card numbers which will force various responses. See the Recurly [Test Credit Card Numbers](#) documentation. Here are some useful numbers:

- a fake credit card: *4111-1111-1111-1111*
- 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 Recurly server.

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

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

```

function: setupForm
setupForm: form submitted
function: processCard
function: handleResponse

```

Yea! You're communicating with the Recurly server.

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

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



```

$('.registrations.new').ready(function() {
  var signature = $('#new_user').data('signature')
  var ip_address = $('#new_user').data('ip_address')
  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 plan;
      plan = {
        plan_code: $('#plan').val(),
      };
      var coupon;
      coupon = {
        coupon_code: $('#coupon').val(),
      };
      var card;
      card = {
        customer_id: $('#user_customer_id').val(),
        email: $('#user_email').val(),
        first_name: $('#user_first_name').val(),
        last_name: $('#user_last_name').val(),
        number: $('#card_number').val(),
        cvc: $('#card_code').val(),
        expMonth: $('#card_month').val(),
        expYear: $('#card_year').val(),
        country: $('#country').val(),
        ip_address: ip_address
      };
      return Recurly.Subscription.save(signature, plan, coupon, card,
subscription.handleResponse);
    },
    handleResponse: function(response) {
      if(response.success) {
        $('#user_card_token').val(response.success.token)
        $('#card_form')[0].submit()
      }
      else if(response.errors) {
        $('#card_error').text(Recurly.flattenErrors(response.errors)).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.new" In this case, the code will run on the Registrations#New page.

We set variables for `signature` and `ip_address` which we obtain from the HTML5 data attributes of the tag with the `#new_user` id.

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

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 Recurly 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 Recurly library named `Subscription.save` which submits the card data to the Recurly server. When the Recurly server returns a response, we call the `handleResponse` function.

The `handleResponse` function checks the `response` object for a success boolean. If the success code is true, the function sets the `card_token` hidden field in the form (it has the HTML element ID "user\_card\_token") and calls `submit` on the element with the "card\_form" class (which is the form). If the status code is false, the function sets the error message from the Recurly server in the `card_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 Recurly server, the response 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 4111-1111-1111-1111. 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 should check with Recurly to make sure a new customer was created before we save the User record.

# Modify the User Model

We want to make a request to Recurly to check that a customer record was created before a new user is saved. There are several ways we could do this:

- modify the Registrations controller `create` action to initiate a request to Recurly
- 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 `check_recurly` method to the User model and call it with `before_create` 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 Recurly 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 is optimal. We’ll add a `check_recurly` method that is called before the user instance is created.

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 :first_name, :last_name, :email, :password, :password_confirmation,
  :remember_me, :card_token, :customer_id
  attr_accessor :card_token
  before_create :check_recurly

  def name
    name = "#{first_name.capitalize} #{last_name.capitalize}"
  end

  def check_recurly
    customer = Recurly::Account.find(customer_id) unless customer_id.nil?
    rescue Recurly::Resource::NotFound => e
      logger.error e.message
      errors.add :base, "Unable to create your subscription. #{e.message}"
      false
    end
  end
end
```

We add `before_create :check_recurly` to call the `check_recurly` method when the user is created.

The `check_recurly` method makes our request to the Recurly service. We skip the process if `customer_id` is nil. 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 (we don't set a `customer_id` for the admin and sample accounts).

If the user instance has a `customer_id`, we make a request using the Recurly API call `Recurly::Account.find` and look up the customer record using the `customer_id`. If the API call fails or the customer is not found, we don't create a User record and we throw an error. If the customer is found, we obtain and set the last four digits of the user's credit card number and the User record is created.

## Test Recurly Integration

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

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 `4111-1111-1111-1111`. 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 Recurly account profile and see the new customer you've created. Look for "Customer Service/Subscriptions." 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 Recurly “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 Recurly"
$ git checkout master
$ git merge --squash billing
$ git commit -m "add Recurly"
$ git branch -D billing
```

## Chapter 20

# Account Changes

We've completed integration with Recurly so we can sign up new subscribers. Recurly will automatically bill their credit cards every month.

Our subscribers may need to make changes to their accounts. They may wish to change an email address or password, upgrade or downgrade a subscription plan, or change a credit card number.

Our "Edit account" page is very similar to the "New account" registration page. In its rudimentary form, it contains a simple form to change the user's name, email address, or password. The **app/views/devise/registrations/edit.html.erb** file implements this feature using the Devise registrations controller. There's no need to replace the existing form. We can use the existing form for changes to user name, email address, or password. We'll add to this page to implement Recurly account changes.

Recurly provides a user-facing website where subscribers can change their credit card numbers. To make it easy for our subscribers to change a credit card number, we'll add a link to the Recurly account page. We're not storing credit card numbers in our application so there's nothing more we need to do.

Changing subscription plans requires updating both our application User records and the Recurly account records. We'll add a form to our "Edit Account" page and make changes to the registrations controller and User model to implement this functionality.

Get started by creating a new git branch for the account changes implementation:

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

## Modify "Edit account" Page for "First Name" and "Last Name"

Recurly customer accounts require both a first name and last name. We'll modify the "Edit account" Page to add "First Name" and "Last Name".

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

Remove:

```
<%= f.input :name, autofocus: true %>
```

and substitute:

```
<%= f.input :first_name, autofocus: true %>
<%= f.input :last_name %>
```

Now the form matches the User model.

## Change the Style for the “Cancel My Account” Link on the “Edit account” Page

We’re going to add small buttons using Twitter Bootstrap styles for our “Change Card” and “Change Plan” features.

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

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 “Cancel my account” link as a miniature button.

You can also remove the line:

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

## Add a Link on the “Edit account” Page to “Change Card”

For a “Change Card” feature, we’ll add a link to the user’s Recurly account page.

Modify the file **app/views/devise/registrations/edit.html.erb**. Add this code above the `<h3>Cancel my account</h3>` statement:

```
<h3>Card</h3>
<p>
  Need to use a different credit card?
  <%= link_to "Change Card", "https://#{RECURLY_SUBDOMAIN}.recurly.com/", class: "btn
  btn-mini" %>
</p>
```

This will add a link to the Recurly website. We apply a Twitter Bootstrap style to display the link as a button.

When the user visits the link, the user will see a page identified as “Subscription billing powered by Recurly.” They will enter their email address to receive a password to access their Recurly customer account. There are two limitations to this approach. First, the password they use for their Recurly customer account is different from the password they use to access your site. Second, the design of the Recurly customer account page will not be the same as your own site. If these limitations trouble you, you can implement your own “Change Card” functionality using the Recurly API. However, given the complexity of the additional code required to implement the alternative, and the fact that users will seldom need to change their credit cards, it is easier to simply provide a link to the Recurly customer account page.

## Add a Form to the “Edit account” Page to “Change Plan”

Your site will likely have more than one subscription plan. Effective marketing often requires tiered plans at different price points. If so, you’ll want to provide an option for a user to upgrade plans. And they may want to downgrade plans, as well. When a subscriber changes subscription plans, we’ll need to inform Recurly as well as update our own user records.

Here’s where you’ll experience the benefits of using a billing service. Recurly manages all the complexity of prorating charges and issuing a credit or collecting additional fees (see details about [Recurly Upgrades and Downgrades](#)). If you were not using a billing service, you’d have to implement this complexity yourself.

To implement the “Change Plan” feature, we’ll add an additional form to the “Edit account” page. The form will be revealed as a modal window when a “Change plan” button is clicked.

First we’ll add a button that reveals the modal window containing the form.

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>

```

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 next.

Now we’ll add the code for the modal window containing a form. It doesn’t matter where the form is placed in the file as it will be hidden when the page is initially displayed and will appear as an overlay when revealed by a click on the appropriate link. For convenience, we’ll add it at the end of the file.

The form is identified as “plan-options”:

```

<div id="plan-options" class="modal" style="display: none;">
  <%= simple_form_for resource, :as => resource_name, :url =>
registration_path(resource_name) , :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 {|l| l.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.

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

```

<h2>Account</h2>
<div id="card_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 :first_name, autofocus: true %>
  <%= f.input :last_name %>
  <%= 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>Card</h3>
<p>
  Need to use a different credit card?
  <%= link_to "Change Card", "https://#{RECURLY_SUBDOMAIN}.recurly.com/", class: "btn
btn-mini" %>
</p>

<h3>Subscription Plan</h3>
<p>
  <%= @user.roles.first.name.titleize %>
  <%= link_to "Change plan", "#plan-options", class: "btn btn-mini", data: {toggle:
"modal"} %>
</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">
      <%= link_to "&#215;" .html_safe, "#", class: "close", data: {dismiss: "modal"} %>
      <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', checked: resource.role_ids.first %>
    </div>
    <div class="modal-footer">
      <%= f.submit "Change Plan", class: "btn btn-primary" %>
    </div>
  </div>

```

```
<%= link_to "Close", "#", class: "btn", data: {dismiss: "modal"} %>
</div>
<% end %>
</div>
```

There's a lot of new code in the Devise user edit view. To accommodate the "Change Plan" 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 Recurly 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 :first_name, :last_name, :email, :password, :password_confirmation,
:remember_me, :card_token, :customer_id
  attr_accessor :card_token
  before_create :check_recurly
  before_destroy :cancel_subscription

  def name
    name = "#{first_name.capitalize} #{last_name.capitalize}"
  end

  def check_recurly
    customer = Recurly::Account.find(customer_id) unless customer_id.nil?
  rescue Recurly::Resource::NotFound => e
    logger.error e.message
    errors.add :base, "Unable to create your subscription. #{e.message}"
    false
  end

  def cancel_subscription
    unless customer_id.nil?
      customer = Recurly::Account.find(customer_id)
      subscription = customer.subscriptions.first unless customer.nil?
      if (!subscription.nil?) && (subscription.state == 'active')
        subscription.cancel
      end
    end
  rescue Recurly::Resource::NotFound => e
    logger.error e.message
    errors.add :base, "Unable to cancel your subscription. #{e.message}"
    false
  end
end
end

```

We've added a `cancel_subscription` method that is activated by the `before_destroy` callback. The method uses the Recurly API to obtain a Recurly customer object and then initiates a subscription cancellation call. We return false to cancel the `destroy` action if Recurly returns an error.

The `cancel_subscription` method only takes action if the user has a Recurly id. You may want to delete the example users created by the `db:seed` process. The example users don't have a

Recurly id and the `destroy` action would fail for them unless we make the `cancel_subscription` method conditional. We also make sure the Recurly customer exists and has an active subscription before attempting to cancel the subscription.

## 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 Recurly when a subscription plan changes. Recurly will prorate 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 :first_name, :last_name, :email, :password, :password_confirmation,
:remember_me, :card_token, :customer_id
  attr_accessor :card_token
  before_create :check_recurly
  before_destroy :cancel_subscription

  def name
    name = "#{first_name.capitalize} #{last_name.capitalize}"
  end

  def check_recurly
    customer = Recurly::Account.find(customer_id) unless customer_id.nil?
  rescue Recurly::Resource::NotFound => e
    logger.error e.message
    errors.add :base, "Unable to create your subscription. #{e.message}"
    false
  end

  def update_plan(role)
    self.role_ids = []
    self.add_role(role.name)
    customer = Recurly::Account.find(customer_id) unless customer_id.nil?
    unless customer.nil?
      subscription = customer.subscriptions.first
      subscription.update_attributes! :timeframe => 'now', :plan_code => role.name
    end
  rescue Recurly::Resource::Invalid => e
    logger.error e.message
    errors.add :base, "Unable to update your subscription. #{e.message}"
    false
  end

  def cancel_subscription
    unless customer_id.nil?
      customer = Recurly::Account.find(customer_id)
      subscription = customer.subscriptions.first unless customer.nil?
      if (!subscription.nil?) && (subscription.state == 'active')
        subscription.cancel
      end
    end
  end
  rescue Recurly::Resource::NotFound => e

```

```

    logger.error e.message
    errors.add :base, "Unable to cancel your subscription. #{e.message}"
    false
  end
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 obtain a customer object with a call to Recurly, make sure the user has a Recurly customer account, and call the Recurly `subscription.update_attributes!` method.

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

## Modify the Registrations Controller for Subscription Plan Changes

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

We will add an `update_plan` action to the registrations controller to accommodate the “Change Plans” 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 two different forms (changing the plan 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 plan, so we’ll leave the `update` method untouched (for changing name/email/password) and add a new `update_plan` method.

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"
      @signature = Recurly.js.sign :subscription => { :plan_code => @plan }
      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

  private
  def build_resource(*args)
    super
    if params[:plan]
      resource.add_role(params[:plan])
    end
    resource.customer_id ||= SecureRandom.uuid
  end
end

```

Not every request will require a change of 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 Recurly to update the customer account. Recurly 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 Recurly 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.

# Add Routes for Account Changes

We've added a new action to the registrations controller to accommodate the "Change Plans" feature. We need to add a route to invoke the new action.

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

```
RailsRecurllySubscriptionSaas::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'
  end
  resources :users
end
```

The `devise_scope` method ([see API](#)) applies [Devise.mappings](#) so the new routes are associated with other routes specified by `devise_for` ([see API](#)). 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` route to respond to an HTTP "put" request (a form submission). This generates the route that we use in our "Change Plans" form:

- `update_plan_path`

Our "Change Plans" feature is 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 Recurly'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 Recurly customer account.

Now an administrator can change subscription plans and Recurly 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

# Recurly Push Notifications

What happens when a credit card expires or a monthly transaction is declined? Recurly will automatically retry a recurring payment after it fails and send notification emails to the customer. After a number of attempts (set in your Recurly “Dunning Management” settings), Recurly will cancel the subscription. But how will your application know to deny access for a subscriber with an expired account? Recurly provides [webhooks](#) (push notifications) to communicate events to you (for details, see the [Recurly Push Notifications documentation](#) and [Recurly Push Notifications API](#)).

A Recurly push notification is an HTTP request from Recurly’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 Recurly servers. The HTTP request contains XML data that provides data about the event, including a customer account code that can be used to retrieve the data from the Recurly server. It is best to ignore the event data (because it could be falsified) and query the Recurly server to obtain the subscription status.

There are a few ways we could handle push notifications. We could use the existing Registrations or Users controller, adding a new non-RESTful action to process the push notifications. Instead we will create a new controller with a new action to process push notifications.

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

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

## Create a Recurly Controller

You could use the `rails generate` command to create a controller but we don’t need views so it is easiest to just create a new file for the controller.

Create a file **app/controllers/recurly\_controller.rb**:

```

class RecurlyController < ApplicationController
  protect_from_forgery :except => :push

  def push
    notification = Hash.from_xml(request.body())
    render :text => "Request accepted."
    if notification.has_key?('expired_subscription_notification')
      account_code =
notification['expired_subscription_notification']['account']['account_code']
      logger.info "Recurly push notification: expired_subscription_notification for
account #{account_code}"
      customer = Recurly::Account.find(account_code)
      subscription = customer.subscriptions.first unless customer.nil?
      if (!subscription.nil?) && (subscription.state == 'expired')
        user = User.find_by_customer_id(account_code)
        user.expire unless user.nil?
      end
    end
  rescue Recurly::Resource::NotFound => e
    logger.error "Recurly: #{e.message}"
  rescue ActiveRecord::RecordNotFound => e
    logger.error "Customer record not found: #{e.message}"
  end
end

```

There's no need to make this a RESTful controller (with actions such as `index`, `update`, etc.) as we are just responding to one type of request. We can give any name to the action; we'll call it `push`.

The action will respond to an HTTP POST request, equivalent to receiving a form submission. Ordinarily, all Rails controllers expect to find an `authenticity_token` parameter in a POST request that is automatically generated as a hidden field in any form by the Rails form helpers. The Rails controller `protect_from_forgery` method adds a `before_filter` called `verify_authenticity_token` to all actions. It will compare the `authenticity_token` parameter against an ID stored in the session variable. If they differ, the action will not be executed. This provides protection from Cross-Site Request Forgery (CSRF) attacks. Recurly can't provide an authenticity token so we will disable CSRF protection for the `push` action with the statement `protect_from_forgery :except => :push`.

Recurly provides event data in the form of an XML document. We really don't want to write code to parse the XML data (that's a 1990s exercise); fortunately, Ruby's Hash class give us a `from_xml` method that will create a hash from the HTTP request body. We let the Recurly server know that we've received the request by rendering a simple text message (all Recurly actually needs is an HTTP 200 status code). If we don't give a response, Recurly will retry ten times before giving up.

Recurly will send push notifications for every customer event. We only care about an “expired\_subscription\_notification” event; we’ll ignore the rest. We check if the hash contains a “expired\_subscription\_notification” key and look for an account code. With the account code, we can make a request to Recurly for a customer object and look for a subscription. The `subscription.state` method should indicate the subscription is expired. If it is, we’ll find the user and call the `user.expire` method.

The `user.expire` method doesn’t yet exist. We’ll add it after we add a route for the new action.

## Add a Route for the Push Notifications

We’ve added a new actions to the Recurly controller to accommodate push notifications. We need to add a route to invoke the new action.

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

```
RailsRecurlySubscriptionSaas::Application.routes.draw do
  post "recurly/push"
  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'
  end
  resources :users
end
```

The route `post "recurly/push"` will invoke the Recurly controller’s `push` action to respond to requests to the URL:

- `http://www.example.com/recurly/push`

You’ll need to set this URL in your Recurly “Push Notifications” settings.

## Add an Expire Method to the User Model

Now that our application responds to push notifications from Recurly, 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 :first_name, :last_name, :email, :password, :password_confirmation,
:remember_me, :card_token, :customer_id
  attr_accessor :card_token
  before_create :check_recurly
  before_destroy :cancel_subscription

  def name
    name = "#{first_name.capitalize} #{last_name.capitalize}"
  end

  def check_recurly
    customer = Recurly::Account.find(customer_id) unless customer_id.nil?
  rescue Recurly::Resource::NotFound => e
    logger.error e.message
    errors.add :base, "Unable to create your subscription. #{e.message}"
    false
  end

  def update_plan(role)
    self.role_ids = []
    self.add_role(role.name)
    customer = Recurly::Account.find(customer_id) unless customer_id.nil?
    unless customer.nil?
      subscription = customer.subscriptions.first
      subscription.update_attributes! :timeframe => 'now', :plan_code => role.name
    end
    true
  rescue Recurly::Resource::Invalid => e
    logger.error e.message
    errors.add :base, "Unable to update your subscription. #{e.message}"
    false
  end

  def update_recurly
    customer = Recurly::Account.find(customer_id) unless customer_id.nil?
    unless customer.nil?
      customer.email = email
      customer.first_name = first_name
      customer.last_name = last_name
      customer.save!
    end
  end
end

```

```

rescue Recurly::Resource::NotFound => e
  logger.error e.message
  errors.add :base, "Unable to update your subscription. #{e.message}"
  false
end

def cancel_subscription
  unless customer_id.nil?
    customer = Recurly::Account.find(customer_id)
    subscription = customer.subscriptions.first unless customer.nil?
    if (!subscription.nil?) && (subscription.state == 'active')
      subscription.cancel
    end
  end
end

rescue Recurly::Resource::NotFound => e
  logger.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 Recurly push notification 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 Recurly push notification notifies the application of a cancelled subscription.

# Set Your Push Notification URL in Your Recurly Account Settings

After you've implemented handling of Recurly push notifications, you need to set your URL in your Recurly account settings.

Visit your Recurly dashboard and enter the URL in your "Push Notifications" settings:

- `http://www.example.com/recurly/push`

There's no easy way to test the push notifications directly from Recurly until after you deploy, since the Recurly servers won't find your application at <http://localhost:3000/recurly/push>. But we can create a test harness you can use to test push notifications locally.

## Faking a Push Notification in Development

The code we've provided in the tutorial should work as implemented (if it doesn't, check the GitHub issues for this application). If you're having problems, or you'd like to extend the application with responses to other push notifications, it'd be nice to have some code that would invoke the controller's `push` action.

We can do that by adding a new `test` action to the Recurly controller in the **`app/controllers/recurly_controller.rb`** file:

```

class RecurlyController < ApplicationController
  protect_from_forgery :except => :push

  def push
    notification = Hash.from_xml(request.body())
    render :text => "Request accepted."
    if notification.has_key?('expired_subscription_notification')
      account_code =
notification['expired_subscription_notification']['account']['account_code']
      logger.info "Recurly push notification: expired_subscription_notification for
account #{account_code}"
      customer = Recurly::Account.find(account_code)
      subscription = customer.subscriptions.first unless customer.nil?
      if (!subscription.nil?) && (subscription.state == 'expired')
        user = User.find_by_customer_id(account_code)
        user.expire unless user.nil?
      end
    end
  rescue Recurly::Resource::NotFound => e
    logger.error "Recurly: #{e.message}"
  rescue ActiveRecord::RecordNotFound => e
    logger.error "Customer record not found: #{e.message}"
  end

  def test
    xml = <<XML
<expired_subscription_notification>
  <account>
    <account_code>1</account_code>
    <username nil="true"></username>
    <email>verena@example.com</email>
    <first_name>Verena</first_name>
    <last_name>Example</last_name>
    <company_name nil="true"></company_name>
  </account>
  <subscription>
    <plan>
      <plan_code>1dpt</plan_code>
      <name>Subscription One</name>
    </plan>
    <uuid>d1b6d359a01ded71caed78eaa0fedf8e</uuid>
    <state>expired</state>
    <quantity type="integer">1</quantity>
    <total_amount_in_cents type="integer">200</total_amount_in_cents>
    <activated_at type="datetime">2010-09-23T22:05:03Z</activated_at>
    <canceled_at type="datetime">2010-09-23T22:05:43Z</canceled_at>
    <expires_at type="datetime">2010-09-24T22:05:03Z</expires_at>
    <current_period_started_at
type="datetime">2010-09-23T22:05:03Z</current_period_started_at>
    <current_period_ends_at

```

```

type="datetime">2010-09-24T22:05:03Z</current_period_ends_at>
  <trial_started_at nil="true" type="datetime">
    </trial_started_at><trial_ends_at nil="true" type="datetime"></trial_ends_at>
  </subscription>
</expired_subscription_notification>
XML
test_request = HTTParty::Request.new(recurly_push_url)
test_request.open_timeout = 1 # seconds
test_request.read_timeout = 1
test_request.headers = { "Content-Type" => "text/xml" }
test_request.body = xml
test_response = HTTParty.post(test_request)
render :text => "Check server log for result of request to #{recurly_push_url}"
rescue HTTPClient::ReceiveTimeoutError
  logger.info "Testing push notification listener: sent XML to #{recurly_push_url}"
  render :text => "Check server log for result of request to #{recurly_push_url}"
end
end

```

The new action creates an XML document that matches a Recurly “expired\_subscription\_notification” push notification (see the [Recurly Push Notifications API](#)). Then we construct an HTTP request, specifying the “Content-Type” as “text/xml” and using the XML document as the request body. We use the HTTPClient library to send the request to the URL formed by the `recurly_push_url` route.

This fully meets the definition of a hack. As soon as you add a route to the **config/routes.rb** file, you can fake a Recurly push notification.

## Add a Route to Test Push Notifications

We’ve added an action to the Recurly controller to test push notifications. We need to add a route to invoke the new action.

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

```

RailsRecurlySubscriptionSaas::Application.routes.draw do
  get "recurly/test"
  post "recurly/push"
  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'
  end
  resources :users
end

```

The route `get "recurly/test"` will invoke the Recurly controller's `push` action when you enter this URL in the browser:

- <http://localhost:3000/recurly/test>

Use it for simple testing of push notifications in development. You won't see a response in the browser but you can dig into the server log to see the application's response to the request.

## Testing a Recurly Push Notification in Production

It's not easy to test a Recurly push notification in production.

The most expedient test is to deploy the application on a production web server and trigger an actual event 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 debugging statements in the **app/controllers/recurly\_controller.rb** file:

```

def push
  logger.info request.body()
  notification = Hash.from_xml(request.body())
  .
  .
  .
end

```

The debug code should show you the results of any push notification request in your log file.

You can use the [RequestBin](#) service to see what Recurly 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 actual push notifications from Recurly 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 push notifications from the Recurly servers.

## Git Workflow

Commit your changes to git:

```
$ git add -A
$ git commit -m "add Recurly webhooks"
$ git checkout master
$ git merge --squash webhooks
$ git commit -m "add Recurly 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 Recurly for payment processing, allows users to change credit cards or subscription plans, and uses Recurly webhooks to delete users who have expired or declined credit cards.

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 Recurly. Use a different email address with a fake credit card number to test subscribing using Recurly. 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 Recurly dashboard.

Stop the server with Control-C.

## Testing

If you've copied the RSpec unit tests and Cucumber integration tests from the [rails-recurly-subscription-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_FIRST_NAME='First' ADMIN_LAST_NAME='User'
ADMIN_EMAIL='user@example.com' ADMIN_PASSWORD='changeme'
$ heroku config:add RECURLY_API_KEY=secret RECURLY_JS_PRIVATE_KEY=secret
RECURLY_SUBDOMAIN=mydomain
```

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.

Subscription Site with Recurly  
Copyright (C) 2013 Daniel Kehoe. All rights reserved.