RailsApps Project

Startup Prelaunch Application

Ruby on Rails tutorial for a startup prelaunch site with a signup page.

Contents

Chapter 1

Introduction

Ruby on Rails tutorial showing how to create a “launching soon” application for a startup prelaunch site with a sign-up page.

The initial website for a typical web startup announces the founders’ plans and encourages visitors to enter an email address for future notification of the site’s launch.

With this application, you’ll be able to:

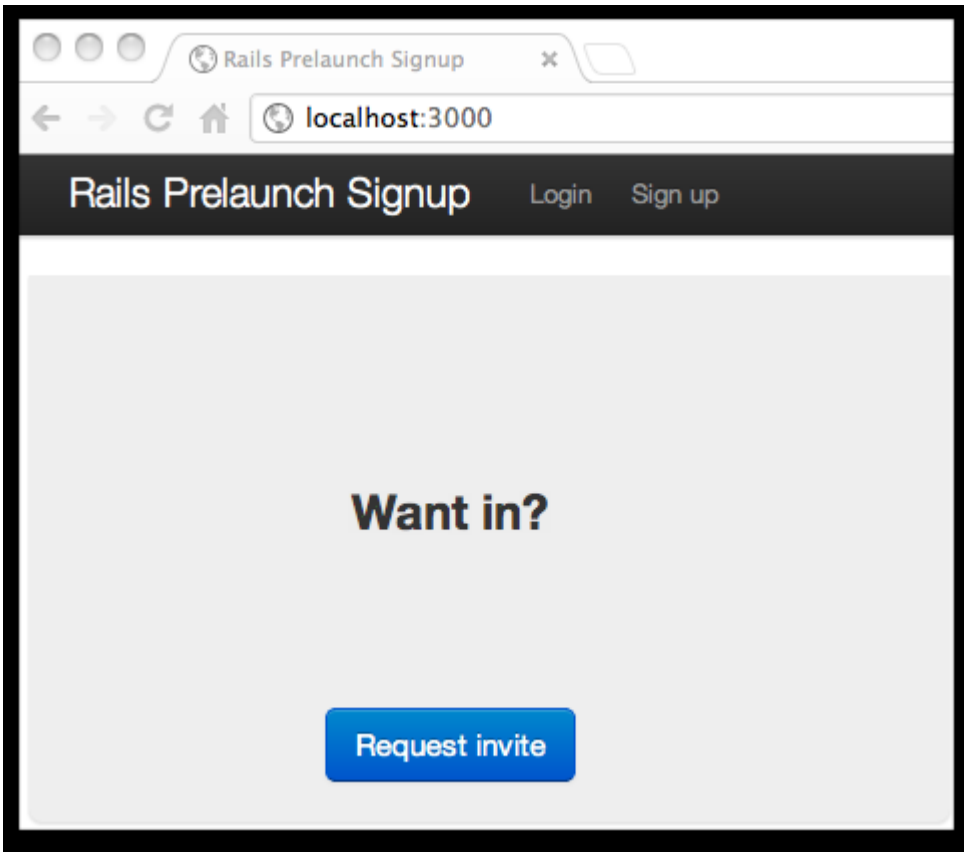
- sign up visitors
- encourage visitors to share the site on social networks
- add email addresses to a MailChimp mailing list
- send invitations (individually or in bulk) to create new user accounts

Unlike a service such as LaunchRock, KickoffLabs and Unbounce, you’ll have your own application you can customize as you wish. Just as important, when your visitors sign up, they are creating real user accounts in a user management system you can use after you launch. When you launch, you can send invitations that prompt each user to complete their registration process and begin using the new site.

We use these popular gems:

- Devise gives you ready-made authentication and user management.
- CanCan provides authorization which restricts the resources a user is allowed to access.
- Twitter Bootstrap is a front-end framework for CSS styling.

Screenshot



Is It for You?

It's not difficult to build an application such as this in Rails. But why build it yourself if others have already done so? This project 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.

By using code from this project, you'll be able to:

- direct your attention to the design and product offer for your prelaunch site;
- get started faster building the ultimate application for your business.

The tutorial can help founders who are new to Rails to learn how to build a real-world Ruby on Rails web application.

You'll find the complete rails-prelaunch-signup 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.

This 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 a list of top resources for Ruby and Rails. The article What is Ruby? And Rails? is a good place to get a basic introduction to Rails.

This is one in a series of Rails example apps and tutorials from the RailsApps Project.

Support the Project

The RailsApps project provides example applications that developers use as starter apps. Hundreds of developers use the apps, report problems as they arise, and propose solutions. Rails changes frequently; each application is known to work and serves as your personal "reference implementation" so you can stay up to date. Each is accompanied by a tutorial so there is no mystery code. Maintenance and development of the RailsApps applications is supported by subscriptions to the RailsApps tutorials. Subscription sales support the project so we can keep the applications current so you'll always have an up-to-date reference implementation.

Chapter 2

Concepts

The application exists to collect email addresses of visitors who are interested in using your website when it is ready for launch.

Unlike a service such as LaunchRock, KickoffLabs and Unbounce, you'll have your own application you can customize as you wish. Just as important, when your visitors sign up, they are creating real user accounts in a user management system you can use after you launch. When you launch, you can send invitations that prompt each user to complete their registration process and begin using the new site.

Why We Use Devise

We've chosen to use the Devise gem for our authentication and user management.

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. By using Devise for the prelaunch sign-up application, you can use the same user database for your post-launch product. You'll also have the benefit of receiving help from a large community of developers using Devise, should you need help in troubleshooting or customizing the implementation.

You can look for help on the Devise wiki and Stack Overflow.

How We Configure Devise

Devise offers many features and configuration choices; in fact, exploring its options takes more time than actually implementing authentication from scratch. This app will save you time; we've selected the configuration that best accommodates a typical web startup sign-up site.

Devise handles signing up users through a registration feature. We use the Devise registration process to collect email addresses and create user accounts. Instead of confirming an email address immediately, we email a "thank you for your request" acknowledgment and postpone the email confirmation step. When you are ready to invite users to the site, go to the app's administrative interface. Then select users to receive an invitation that instructs them to confirm their email address and set a password.

Using Git Branches

The git version control system allows you to maintain two separate branches (for example, “master” and “work in progress”).

When you complete the tutorial, you will have a ready-to-run application in the master branch that you can deploy to the production server.

The prelaunch application can be up and running on your production server while you work on implementing your “real” site.

As visitors sign up on the prelaunch site, continue working on the code in the “work in progress” branch. If you wish, you can deploy the unfinished “work in progress” branch to a staging or testing server.

If you want to invite selected users as beta testers, use the staging server or set up a separate “beta” server with the “work in progress” branch.

When you ready to replace the prelaunch site with your finished application, rename the master branch as “prelaunch.” Then rename the “work in progress” branch as “master” and deploy to the production server.

Prelaunch and Post-Launch Database

You can use the same user database before and after you launch.

During your prelaunch phase, your visitors will add themselves to the application database. An email address is saved to the database (along with administrative details such as creation date). The new user will receive an acknowledgment.

As you implement the “real” application, you’ll be changing the database schema by adding new Model classes and adding attributes to the existing User model. Rails manages the database changes with “migrations.” Each time you add a model or new attributes, you’ll run a migration on your development database. As long as you don’t delete the User email attribute, you can add anything to the database schema without worrying that you’ll lose your user data.

When you ready to replace the prelaunch site with your finished application, you’ll deploy the new master branch and run pending migrations on the production database. All the user accounts will remain intact and you’ll add the new models and attributes required by the “real” application.

Since you are developing with Rails and using git for version control, this approach is not unusual. In the real world, any application wil have multiple branches; typically, a “stable” branch is deployed to production and “edge” branches accommodate bug fixes or new features that are periodically rolled into the stable branch and deployed to production. You’ll

gain experience with this approach by building your “real” application on top of your prelaunch application.

Chapter 3

Product Planning

A robust software development process includes product planning, project management, and testing. Arguably, for a simple application like this, you don't need a lot of "ceremony." However, whether you're a solo operator or part of a team, product planning can be helpful. Consider some of the benefits:

- It helps you discover the functionality you need to implement.
- It helps you describe and discuss features with your business partners.
- It serves as a "to-do list" to help you track progress.
- It is the basis for acceptance testing or integration testing.

The article Rails and Product Planning goes into further detail about how to include product planning in your software development process.

User stories (definition: user stories) are a way to discuss and describe the requirements for a software application. The process of writing user stories will help you identify all the features that are needed for your application. Breaking down the application's functionality into discrete user stories will help you organize your work and track progress toward completion.

User stories are generally expressed in the following format:

```
As a <role>, I want <goal> so that <benefit>
```

As an example, here are user stories we will implement for this application.

```
*Request Invitation*
As a visitor to the website
I want to request an invitation
so I can be notified when the site is launched

*See Invitation Requests*
As the owner of the site
I want to view a list of visitors who have requested invitations
so I can know if my offer is popular

*Send Invitations*
As the owner of the site
I want to send invitations to visitors who have requested invitations
so users can try the site

*Collect Email Addresses*
As the owner of the site
I want to collect email addresses for a mailing list
so I can send announcements before I launch the site

*Social Network Sharing After Sign Up*
As a user
I want an option to post to a social network after I sign up
so my followers will learn about the site
```

We'll use this list of user stories as our to-do list in implementing the application.

Chapter 4

Project Management

Building a web application is a complex project; building a business is even more complicated. If you're working solo, you'll want to track progress toward your goals. If you've put together a team, you'll need a way to keep track of tasks so you can communicate and coordinate.

Once you've defined your tasks through a product planning process, you'll need to organize and track your tasks. A simple to-do list may be all you need for a small project. Keep track of your tasks on paper or with a web or mobile app. I also recommend looking at the *kanban* approach that is often used in software development.

See article Rails and Project Management for suggestions of some tools and approaches you can use for task management.

If you've given some thought to product planning and project management, let's consider what you'll need to accomplish before you begin building your application.

Chapter 5

Accounts You May Need

Before you start, you will need accounts for *domain registration*, *hosting*, *ssl*, *email*, and a *source control repository*.

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.

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

As a general practice, it is wise to host any webapp that requires login over an SSL connection. SSL, the Secure Sockets Layer, allows web browsers to connect to web servers without risk of eavesdropping and tampering.

You can set up an SSL connection inexpensively using the CloudFlare CDN (content delivery network) service. Or you can purchase an SSL certificate and set up SSL directly with Heroku. If you're not deploying to Heroku, do your research early to find out how to set up SSL and apply for an SSL certificate if necessary.

CloudFlare for SSL

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, your website visitors will connect to Cloudfare with their web browsers, and Cloudfare will connect to Heroku securely, providing a complete SSL connection through Cloudfare to Heroku with your custom domain. Not only do you get SSL for no more than you'd pay at Heroku to use an SSL certificate, but you get the Cloudfare CDN services as part of the bargain.

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

Heroku for SSL

If you don't want to use CloudFlare, you can set up an SSL certificate for a custom domain on 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).

Email Service Providers

You'll need infrastructure for three types of email:

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

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.

Mailing List

This tutorial shows how to add invitation requesters to a MailChimp list. MailChimp allows you to send up to 12,000 emails/month to list of 2000 or fewer subscribers for free. After you sign up for a MailChimp account, get your API key. Look under "Account" for "API Keys and Authorized Apps."

After you have set up a MailChimp account, create a new mailing list where you can collect email addresses of visitors who have requested invitations. The MailChimp "Lists" page has

a button for "Create List." The list name and other details are up to you. You'll need to obtain the list ID for use later in the tutorial when we add code to subscribe visitors to the MailChimp list. To find the list ID, on the MailChimp "Lists" page, look for the dropdown "gear" menu and click "List Settings and Unique ID." At the bottom of the List Settings page, you'll find the unique ID for the mailing list.

You should create a welcome message for the mailing list. When our application adds a visitor to the mailing list, the visitor will receive a welcome email. This will be the message that acknowledges they've been added to the waiting list to receive an invitation to the new site when it launches. The message should tell them they are on the invitation list, remind them why your new site will be great, and let them know if they should expect newsletter or announcement emails. It's a bit difficult to find the MailChimp option to create a welcome message. Look for the button "Design Signup Forms" or the "Forms" menu item under the settings "gear" associated with each list. You might not think a welcome email is a form, but that's what MailChimp names it. In the dropdown list of "Forms & Response Emails" it is named "Final 'Welcome' Email".

Transactional Email

This tutorial provides instructions for Mandrill by MailChimp. The Mandrill transactional email service integrates well with the MailChimp email list manager service. Plus, you can send up to 12,000 emails/month from the service for free.

Sign up for a MailChimp account to get started. After you've created your MailChimp account, see the instructions How do I use Mandrill if I already have a MailChimp account?.

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 6

Getting Started

Before You Start

Most of the tutorials from the RailsApps project take about two hours to complete. This one is more complex; allow around six hours.

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-prelaunch-signup 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

Text Editor

You'll need a text editor for writing code and editing files. I recommend Sublime Text for Mac OS X, Windows, or Linux.

Terminal

You'll need a Terminal application to run programs from your computer's command line. "The Command Line Crash Course" explains how to launch a terminal application.

Ruby and Rails

Check that you have appropriate versions of Ruby and Rails installed in your development environment. You'll need:

- The Ruby language (version 2.0.0)
- The Rails gem (version 3.2)

Here's how to check your versions of Ruby and Rails:

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

If older versions are installed, you must install newer versions to avoid unexpected problems.

The Installing Rails article will show you how to install Ruby 2.0 and Rails 3.2. Be sure to review the article to make sure your development environment is set up properly. A Rails development environment includes several important gems in addition to Rails. The article shows how to update key components needed for Rails to run successfully.

RVM

The tutorial assumes you are using RVM, the Ruby Version Manager. The Installing Rails article will show you how to install Rails using RVM. RVM makes it easy to switch between different versions of Ruby and Rails, as well as create project-specific gemsets that can help you isolate problems when you upgrade gems.

Chapter 7

Create the Application

Take a moment to think about how you're going to create the application.

Keep in mind that all the applications from the RailsApps project are available on GitHub. That means you can download the entire working application. It's a good idea to download (clone) the entire application if you run into problems and want to try a "known-to-work" version of the application. You can use a file compare tool to see if there are differences between your version and the "reference implementation." You'll need git on your machine. See Rails and Git. To clone:

```
$ git clone git://github.com/RailsApps/rails-prelaunch-signup.git
```

If you wish to skip the tutorial and build the application immediately, you can download the application from GitHub and search-and-replace the project name throughout the application. But there's a better option! You can use the Rails Composer tool to generate the complete working application. You'll be able to give it your own project name when you generate the app. Generating the application gives you additional options. You'll find instructions on the rails-prelaunch-signup README page.

Copy and Paste from the Tutorial

The most obvious way to get the code into your text editor is to copy and paste from this tutorial, assuming you are reading this on your computer (not printed pages or a tablet). It's a bit tedious and error-prone but you'll have a good opportunity to examine the code closely.

Some students like to type in the code, character by character. If you have patience, it's a worthwhile approach because you'll become more familiar with the code than by copying and pasting.

Building from Scratch

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

We'll use the rails3-bootstrap-devise-cancan application as a starter app. Using the starter app saves many steps. Devise will be installed for authentication and 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.

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, using RVM to create a project-specific gemset at the same time:

```
$ mkdir rails-prelaunch-signup
$ cd rails-prelaunch-signup
$ rvm use ruby-2.0.0@rails-prelaunch-signup --ruby-version --create
$ gem install rails
$ rails new . -m https://raw.github.com/RailsApps/rails-composer/master/composer.rb -T
```

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

We create a new project directory. You can use a different name if you wish.

The `rvm ...` command creates a new gemset and saves a **.ruby-version** file to the directory (see Project Gemsets with RVM). Then we install the newest version of Rails.

Note the period after the `rails new` command. The `rails new .` command creates a new Rails application named after the directory you are in. In this case, your application will be named `rails-prelaunch-signup` (unless you've chosen a different name for your project directory).

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

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 "rails-prelaunch-signup" (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
question  Continuous testing?
      1)  None
      2)  Guard
  extras  Set a robots.txt file to ban spiders? (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

For "Continuous testing?" choose "None" for this tutorial. Choose "Guard" if you are doing test-driven development and want to run tests automatically when you make changes to your application.

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

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

It takes a few minutes for Rails Composer to generate the starter application.

Troubleshooting

If you get an error like this:

```
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 (…)".

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

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. If you like, add an acknowledgment and a link to the RailsApps project.

Set Up Source Control (Git)

When you generate the starter app, Rails Composer sets up a source control repository and makes an initial commit of the code. See the article Rails with Git and GitHub for more information.

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

Quick Test

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

For a "smoke test" to see if the starter app runs, display a list of Rake tasks:

```
$ rake -T
```

Set Up Gems

The Rails Composer program sets up your Gemfile.

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", :group => [:development, :test]
gem "database_cleaner", :group => :test
gem "email_spec", :group => :test
gem "cucumber-rails", :group => :test, :require => false
gem "launchy", :group => :test
gem "capybara", :group => :test
gem "factory_girl_rails", :group => [:development, :test]
gem "bootstrap-sass"
gem "devise"
gem "cancan"
gem "rolify"
gem "simple_form"
gem "quiet_assets", :group => :development
gem "figaro"
gem "better_errors", :group => :development
gem "binding_of_caller", :group => :development, :platforms => [:mri_19, :rbx]
```

Add the Gibbon gem which will be used to access the MailChimp API:

```
gem "gibbon"
```

This tutorial requires Rails version 3.2.13.

Note: The RailsApps examples are generated with application templates created by the Rails Apps Composer Gem. For that reason, groups such as `:development` or `:test` are specified inline. You can reformat the Gemfiles 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

Install the required gems:

```
$ bundle install
```

You can check which gems are installed 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.

Commit to Git

Commit your changes to git:

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

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'll see the rails3-bootstrap-devise-cancan application.

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, 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 8

Configuration

Configuration File

The application uses the figaro gem to set environment variables. See the article Rails Environment Variables for more information.

Rails Composer sets up the figaro gem and generates a **config/application.yml** file and lists it in your **.gitignore** file. You'll also have a **config/application.example.yml** file which you can check into your GitHub repository. Don't put any secrets in your **config/application.example.yml** 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.

The Rails Composer tool generates a **config/application.yml** file that looks like this:

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

We'll edit this file to add credentials for our application.

First we'll add the credentials we need to send email. Instead of Gmail for transactional email, we'll use Mandrill to increase deliverability for email messages from the application. Replace `GMAIL_USERNAME` with `MANDRILL_USERNAME`. And `GMAIL_PASSWORD` with `MANDRILL_API_KEY`. Add the Mandrill user name and API key.

When visitors sign up to be notified of our launch, we'll add them to a MailChimp list. Add an environment variable for the MailChimp API key: `MAILCHIMP_API_KEY`. You can find the MailChimp API key under the tab for "Account" on the MailChimp website after you log in. Look for the "Api Keys and Authorized Apps" menu item.

We'll add `MAILCHIMP_LIST_ID` for the ID of the mailing list we'll set up in MailChimp. To find the list ID, log in to MailChimp and on the MailChimp "Lists" page, look for the dropdown "gear" menu for the mailing list you've created and click "List Settings and Unique ID." At the bottom of the List Settings page, you'll find the unique ID for the mailing list.

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.

Specify roles in the configuration file. You will need an "admin" role and "user" role. Remove the "VIP" role as we won't use it.

We'll add `EMAIL_ADDRESS` to provide the default sender email address we use when sending email from the application.

Finally, we'll add `DOMAIN` which is also used when sending email.

Your modified **config/application.yml** file will look like this:

```
MANDRILL_USERNAME: Your_Username
MANDRILL_API_KEY: Your_Mandrill_API_Key
MAILCHIMP_API_KEY: Your_MailChimp_API_Key
MAILCHIMP_LIST_ID: Your_List_ID
ADMIN_NAME: First User
ADMIN_EMAIL: user@example.com
ADMIN_PASSWORD: changeme
ROLES: [admin, user]
EMAIL_ADDRESS: user@example.com
DOMAIN: example.com
```

All configuration values in the **config/application.yml** file are available anywhere in the application as environment variables. For example, `ENV["MANDRILL_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. Use `$ env` to list all the Unix environment variables that are set for your shell. See the article Rails Environment Variables for more information.

Configure Email

You must configure your application to send email messages, for example, to acknowledge invitation requests or send welcome messages. See the article Send Email with Rails for details.

Rails Composer sets up a default email configuration. You must add details about your email account.

When the application sends email messages in development mode, the email messages will be visible in the log file. If you want actual messages to be sent in development mode, configure the **config/environments/development.rb** file to match the **config/environments/production.rb** file.

To deliver email in production, you must 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"
```

Rails Composer sets up the **config/environments/production.rb** file for a connection to Gmail:

```
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"]
}
```

In production, you should use an email service provider such as Mandrill.

Replace the Gmail settings with Mandrill settings in the **config/environments/production.rb** file:

```
config.action_mailer.smtp_settings = {
  :address   => "smtp.mandrillapp.com",
  :port      => 25,
  :user_name => ENV["MANDRILL_USERNAME"],
  :password  => ENV["MANDRILL_API_KEY"]
}
```

Note that the password will be your Mandrill API key. Mandrill does not require settings for `domain`, `authentication`, or `enable_starttls_auto`. You can use `port` 25 or 587 (some ISPs restrict connections on port 25).

You can replace the `ENV["MANDRILL_USERNAME"]` and `ENV["MANDRILL_API_KEY"]` environment variables with your Mandrill username and API key ("hardcoding the credentials"). However, committing the file to a public GitHub repository will expose your secret API key. It is better to obtain these environment variables from the **config/application.yml** file. See the article Rails Environment Variables for details.

Configure Devise for Email

Complete your email configuration by modifying the **config/initializers/devise.rb** file and setting the `config.mailer_sender` option for the return email address for messages that Devise sends from the application.

```
# ==> Mailer Configuration
# Configure the e-mail address which will be shown in Devise::Mailer,
# note that it will be overwritten if you use your own mailer class with default "from"
parameter.
config.mailer_sender = ENV["EMAIL_ADDRESS"]
```

Commit to Git

If you will be sharing your git repository publicly on GitHub, before you commit to git, check the file **.gitignore** and make sure the file **/config/application.yml** is listed so your private configuration settings are not stored in your git repository.

Commit your changes to git:

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

Chapter 9

Test-Driven Development

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

Testing is at the center of any robust software development process. Unit tests confirm that small, discrete portions of the application continue working as you add features and refactor code. RSpec is a popular choice for unit testing. Integration tests determine whether the application's features work as expected, testing the application from the point of view of the user. Cucumber is a popular choice for integration testing and behavior driven development.

The rails3-devise-rspec-cucumber tutorial:

- shows how to set up RSpec and provides example specs for use with Devise
- shows how to set up Cucumber and provides example scenarios for use with Devise

To learn more about writing tests, see the books:

- The RSpec Book
- The Cucumber Book

You won't learn the finer points of writing tests in this tutorial but I'll give you examples you can use to see how good tests work.

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.

Installing Tests from the Example Application

The starter app only installs tests designed for the features of the rails3-bootstrap-devise-cancan example application. A full suite of tests for the rails-prelaunch-signup example application are not yet available.

Commit to Git

If you've downloaded test files from the example application, commit your changes to git:

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

Chapter 10

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:

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

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

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

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

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 11

Authentication

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

The starter app script sets up Devise:

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

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

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

Chapter 12

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 13

User Management

By default, Devise uses an email address to identify users. The starter application adds a "name" attribute as well. We won't ask visitors for a name when they request an invitation but we'll leave the name attribute in the User model in case you want to use it eventually.

Devise provides all the functionality for a user to log in and view and edit their profile. 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

Modify the User Model

Devise offers an optional Confirmable module which sends an email to a new user to request confirmation of an email address when creating a new account. We'll modify the User model to add the Confirmable module.

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

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

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

end
```

We're adding the Confirmable module to an existing application so we'll need to modify the database schema with a migration.

Add Migration for the Confirmable Module

The Confirmable module requires addditional attributes in the User model. Adding `devise :confirmable` to the User model adds the attributes but we'll need matching fields in the User table. We'll change our database schema accordingly.

Create a database migration with this command:

```
$ rails generate migration AddConfirmableToUsers confirmation_token:string confirmed_at:datetime
confirmation_sent_at:datetime unconfirmed_email:string
```

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

```
$ rake db:migrate
```

Next we need to accomodate the Confirmable module in our tests.

Modify the User Factory

Rails Composer sets up the factory_girl gem to create default model objects for tests. For testing the User model, Rails Composer sets up a factory to create a User model object.

We need to modify the User model factory to accommodate the Confirmable module.

Modify the file **spec/factories/users.rb** by uncommenting this line: `confirmed_at Time.now`:

```
FactoryGirl.define do
  factory :user do
    name 'Test User'
    email 'example@example.com'
    password 'changeme'
    password_confirmation 'changeme'
    # required if the Devise Confirmable module is used
    confirmed_at Time.now
  end
end
```

Git

Commit your changes to git:

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

Chapter 14

Initial Data

Set Up a Database Seed File

You'll want to set up default users so you can test the application.

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

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

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

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. Notice we have to supply the argument `:without_protection => true` to override the mass-assignment protection configured in the Role class.

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

After we add the administrator, we need to apply the `user.skip_confirmation!` method and save the user. We will be using the Devise Confirmable module to send a confirmation request to any new user's email address. We skip the confirmation step for our 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.

Seed the Database

We'll reset the database because we've made changes:

```
$ 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 application still looks like 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" (though you won't see a button to delete the admin user).

Stop the server with Control-C.

Chapter 15

Update the Home Page

The starter app displays a list of users on the home page. That's not suitable for a prelaunch sign-up app.

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

```
<h3>Home</h3>
```

You can embellish the page as you wish.

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 16

Request Invitation

We're ready to implement the first user story.

Here's the user story we'll specify and implement:

```
*Request Invitation*
As a visitor to the website
I want to request an invitation
so I'll be notified when the site is launched
```

Before we write any code we'll set up our 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 request-invitation
```

The command creates a new branch named "request-invitation" 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).

Implement "Request Invitation" Form

The application's home page doesn't contain a "Request Invitation" form. We have some choices. Should we add a sign-up form to the home page? We already have a sign-up form provided by Devise, our authentication gem. It'll be easier to adapt the existing Devise mechanism.

We'll modify the Devise form to use the SimpleForm gem. The SimpleForm gem lets us create forms that include tags that apply attractive Twitter Bootstrap form styles.

We'll modify the file **app/views/devise/registrations/new.html.erb** to make it a "Request Invitation" form.

- We'll use SimpleForm.
- We'll change the heading from "Sign up" to "Request Invitation."
- We'll remove the password fields.
- We'll remove the user name field (you could keep it if you wish).
- We'll change the submit button text from "Sign up" to "Request Invitation."
- We'll remove the Devise navigation links.

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

```
<h2>Request Invitation</h2>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name)) do |f| %>
  <%= devise_error_messages! %>
  <%= f.input :email, :placeholder => 'user@example.com' %>
  <%= f.submit "Request Invitation" %>
<% end %>
```

Update the User Model

Devise won't let us create a new user without a password when we use the default `:validatable` module. We want Devise to validate the email address but ignore the missing password, so we'll override Devise's `password_required?` method.

Modify the file **app/models/user.rb** to override methods supplied by Devise:

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

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

  # override Devise method
  # no password is required when the account is created; validate password when the
user sets one
  validates_confirmation_of :password
  def password_required?
    if !persisted?
      !(password != "")
    else
      !password.nil? || !password_confirmation.nil?
    end
  end

end
```

Use Devise Registrations Page for the Home Page

Examine the **config/routes.rb** file to learn how to use the Devise registrations page as our home page.

Rails Composer sets up the **config/routes.rb** file to look like this:

```
RailsPrelaunchSignup::Application.routes.draw do
  authenticated :user do
    root :to => 'home#index'
  end
  root :to => "home#index"
  devise_for :users
  resources :users
end
```

You'll see a route to the home page for authenticated users (those who have an account and are logged in) and the same route for all other users (those who have no account or are not logged in).

Notice that the name of the application is hardcoded in the routes configuration file. If you've named the application "RailsPrelaunchSignup" as we suggested, you can copy the example code below. If not, avoid changing the application name in your file.

To make the Devise registrations page serve as the home page for users who have no account (or are not logged in), replace the second `root :to => "home#index"` as follows:

config/routes.rb

```
RailsPrelaunchSignup::Application.routes.draw do
  authenticated :user do
    root :to => 'home#index'
  end
  devise_scope :user do
    root :to => "devise/registrations#new"
  end
  devise_for :users
  resources :users
end
```

With this change, casual visitors will see a "Request Invitation" form on the home page. And users who log in (presumably only an administrator or invited guests) will see the application "home#index" page.

Create a "Thank You" Page

For a simple "thank you" page, create a file for a static web page. Use your file browser to create the file or use the Unix `touch` command:

```
$ touch public/thankyou.html
```

And modify the file **public/thankyou.html**:

```
<h1>Thank You</h1>
```

Obviously, you can embellish this page as needed.

Later, this tutorial will show you how to eliminate the "thank you" page and use Ajax to update the sign up page with a thank you message. For now, it's helpful to see a simple implementation without Ajax.

Redirect to “Thank You” Page After Successful Sign Up

We want the visitor to see the “thank you” page after they request an invitation.

Override the Devise::RegistrationsController with a new controller. Create a file:

app/controllers/registrations_controller.rb

```
class RegistrationsController < Devise::RegistrationsController
  protected

  def after_inactive_sign_up_path_for(resource)
    # the page prelaunch visitors will see after they request an invitation
    '/thankyou.html'
  end

  def after_sign_up_path_for(resource)
    # the page new users will see after sign up (after launch, when no invitation is
needed)
    redirect_to root_path
  end

end
```

When a visitor creates an account by requesting an invitation, Devise will call the `after_inactive_sign_up_path_for` method to redirect the visitor to the thank you page. In the future, after you launch the application and allow visitors to become active as soon as they create an account, you may use the `after_sign_up_path_for` to redirect the new user to a welcome page.

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

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

Notice that the plural `:controllers` is required (Devise allows more than one controller to be replaced). For more information, the Devise wiki explains How to Redirect to a Specific Page on Successful Sign Up.

Postpone Confirmation of New Accounts

We want a visitor to create a new account when they request an invitation but we don’t want to confirm the email address and activate the account until we send an invitation.

There are several ways to implement this functionality. One approach is to add an “active” attribute to the User model and designate the account as “inactive” when it is created. Another approach is to simply revise the confirmation email to make it a simple “welcome” without the confirmation request but this would require re-implementing the confirmation request process later. The simplest approach is to postpone sending the user a request to confirm their email address, leaving the account unconfirmed until after we send the user an invitation.

We’ll modify the file **app/models/user.rb** to override methods supplied by Devise. We’ll add three new methods:

- `confirmation_required?`
- `active_for_authentication?`
- `send_reset_password_instructions`

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

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

  # override Devise method
  # no password is required when the account is created; validate password when the
user sets one
  validates_confirmation_of :password
  def password_required?
    if !persisted?
      !(password != "")
    else
      !password.nil? || !password_confirmation.nil?
    end
  end

  # override Devise method
  def confirmation_required?
    false
  end

  # override Devise method
  def active_for_authentication?
    confirmed? || confirmation_period_valid?
  end

  def send_reset_password_instructions
    if self.confirmed?
      super
    else
      errors.add :base, "You must receive an invitation before you set your password."
    end
  end

end
```

Devise uses a conditional "after_create" callback to generate a confirmation token and send the confimation request email. It is only called if `confirmation_required?` returns true. By indicating that "confirmation is not required," no confimation email will be sent when the account is created.

When we tell Devise that "confirmation is not required," Devise will assume that any new account is "active_for_authentication." We don't want that, so we override the `active_for_authentication?` method so that unconfirmed accounts are not active.

We eliminate a potential vulnerability by overriding the Devise `send_reset_password_instructions` method. If a visitor requests an invitation but hasn't received an invitation or set up an account, the visitor can circumvent the invitation process by attemmpting to login and clicking the "Forgot your password?" link. We override the Devise `send_reset_password_instructions` method to make sure the user has a confirmed account before emailing password reset instructions.

Send a Welcome Email

Ordinarily, when the Devise Confirmable module is configured and a new user requests an invitation, Devise will send an email with instructions to confirm the account. We've changed this behavior so the user doesn't get a confirmation request. However, we still want to send a welcome email.

We didn't revise the confirmation email to make it a welcome message. That might seem simpler but it would require us to re-implement the confirmation request process later. Instead we'll send an email welcome message using a new ActionMailer method when the account is created.

You can skip this step if you plan on using MailChimp to manage a mailing list of visitors who have requested invitations. Later in this tutorial, we show how to set up MailChimp to collect email addresses. MailChimp offers an option to send a welcome email when a new user is subscribed to the mailing list. MailChimp makes it easy to design an attractive welcome email so we recommend using MailChimp's welcome message. For now, we suggest following along so you can see how we send email from Rails.

Generate a mailer with views and a spec:

```
$ rails generate mailer UserMailer
```

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

```
class UserMailer < ActionMailer::Base
  default :from => ENV["EMAIL_ADDRESS"]

  def welcome_email(user)
    mail(:to => user.email, :subject => "Invitation Request Received")
    headers['X-MC-GoogleAnalytics'] = ENV["DOMAIN"]
    headers['X-MC-Tags'] = "welcome"
  end
end
```

If you're not using Mandrill for an SMTP relay service, you can leave out these statements:

```
headers['X-MC-GoogleAnalytics'] = ENV["DOMAIN"]
headers['X-MC-Tags'] = "welcome"
```

The `X-MC-GoogleAnalytics` header sets up tracking with your Google Analytics account (your domain name is set here). The `X-MC-Tags` header identifies each email as a welcome message for analyzing delivery success.

Create a mailer view by creating a file **app/views/user_mailer/welcome_email.html.erb**. Copy this code to used as the template for the email message, formatted in HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
  </head>
  <body>
    <h1>Welcome</h1>
    <p>
      We have received your request for an invitation to <%= link_to ENV["DOMAIN"], root_url %>.
    </p>
    <p>
      We'll contact you when we launch.
    </p>
  </body>
</html>
```

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

```
Welcome!

We have received your request for an invitation to <%= link_to ENV["DOMAIN"], root_url %>.

We'll contact you when we launch.
```

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 service, you can skip this step if you configure Mandrill to automatically generate a plain-text version of all emails.

Now we'll wire up the User model to send the welcome message when an account is created.

Modify the file **app/models/user.rb** to add the `send_welcome_email` method and a filter for `after_create`:

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

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

  after_create :send_welcome_email

  # override Devise method
  # no password is required when the account is created; validate password when the
user sets one
  validates_confirmation_of :password
  def password_required?
    if !persisted?
      !(password != "")
    else
      !password.nil? || !password_confirmation.nil?
    end
  end

  # override Devise method
  def confirmation_required?
    false
  end

  # override Devise method
  def active_for_authentication?
    confirmed? || confirmation_period_valid?
  end

  def send_reset_password_instructions
    if self.confirmed?
      super
    else
      errors.add :base, "You must receive an invitation before you set your password."
    end
  end

  private

  def send_welcome_email
    return if email.include?(ENV['ADMIN_EMAIL'])
    UserMailer.welcome_email(self).deliver
  end

end
```

We add an `after_create` filter that calls the `send_welcome_email` method.

Now the visitor will get a welcome email when they request an invitation.

There's an unintended side effect of adding the `send_welcome_email` method. When you reset the database with `rake db:reset` you'll send welcome messages to the administrator's address listed in the **db/seeds.rb** file. To eliminate unwanted mail, we short-circuit the method with a return statement so we don't send a welcome email to the administrator's address.

Tweak the User Interface

The required functionality is largely complete. But there are a few small changes to make.

If the user visits the sign-in page immediately after requesting an invitation, they may see this flash message:

"A message with a confirmation link has been sent to your email address. Please open the link to activate your account."

In the file **config/locales/devise.en.yml**, find this message:

```
signed_up_but_unconfirmed: 'A message with a confirmation link has been sent to your
email address. Please open the link to activate your account.'
```

Replace it with this:

```
signed_up_but_unconfirmed: 'Your invitation request has been received. You will receive
an invitation when we launch.'
```

Be careful not to use an apostrophe or single quote in the message unless you surround the text with double quotes.

If the visitor attempts to sign in, they will see this message:

"You have to confirm your account before continuing."

In the file **config/locales/devise.en.yml**, find this message:

```
unconfirmed: 'You have to confirm your account before continuing.'
```

Replace it with this:

```
unconfirmed: 'Your account is not active.'
```

The user will also see links on the sign-in page: “Didn’t receive confirmation instructions?” and “Forgot your password?”. Before we launch, we don’t want visitors to see these links.

The sign-in page is provided from a view template in the Devise gem. We don’t have to modify the sign-in page. Instead we’ll modify the Devise “links” partial.

Modify the file **app/views/devise/shared/_links.html.erb**:

```
<%- if devise_mapping.recoverable? && controller_name != 'passwords' %>
  <%= link_to "Forgot your password?", new_password_path(resource_name) %><br />
<% end -%>
```

Evaluating only the functionality, the feature is complete. In the next section, we’ll improve the look and feel of the feature.

Test the Application

You can check that the application 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 “Request Invitation” form. You can enter an email address and click the button. You should see the “Thank You” page. If you check the console, you should see an indication that an email message was dispatched. No email is actually sent unless you’ve configured the **config/environments/development.rb** file to include `config.action_mailer.perform_deliveries = true`.

After you’ve requested an invitation, log in as the administrator and click on the “Admin” navigation link and you’ll see a list of users. We’ll improve the administrative dashboard later to show the status of each user. Be sure to log out or you’ll still be logged in as the administrator when you next test the application.

Git Workflow

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

```
$ git add -A
$ git commit -m "implement 'Request Invitation' feature"
```

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 request-invitation
$ git commit -m "implement 'Request Invitation' feature"
```

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

```
$ git branch -D request-invitation
```

Chapter 17

Improved Home Page

The functionality of the "Request Invitation" feature is complete. Let's improve the look and feel of the home page.

We can use the JavaScript and CSS provided by Twitter Bootstrap to pop up a form when the "Request invite" button is clicked. Clicking the button triggers a lightbox effect: The form appears in a bright overlay with the page darkened behind it. Because no further action can be taken until the form is submitted or cancelled, we say the form appears in a "modal window." A modal window is a user interface element in desktop applications and on the web, commonly used to block action until the user responds to the message in the window.

Git Workflow

Create a new git branch for the changes you'll make to the page design:

```
$ git checkout -b modal
```

The command creates a new branch named "modal" and switches to it.

Add a Modal Window

It'd be nice for the "request invitation" page to include some "romance copy" to convince the visitor of the value of the site. The "call to action" could be a big button that says, "Request Invitation," which opens a modal window and invites the visitor to enter an email address and click a submit button.

Twitter Bootstrap gives us everything we need to implement this in a few lines of code.

Open the file **app/views/devise/registrations/new.html.erb** and replace the contents with this new code:

```
<div id="request-invite" class="modal" style="display: none;">
  <%= simple_form_for resource, :as => resource_name, :url =>
registration_path(resource_name) , :html => {:class => 'form-horizontal' } do |f| %>
    <div class="modal-header">
      <a class="close" data-dismiss="modal">&#215;</a>
      <h3>Request Invitation</h3>
    </div>
    <div class="modal-body">
      <%= f.error_notification %>
      <%= f.input :email, :placeholder => 'user@example.com' %>
    </div>
    <div class="modal-footer">
      <%= f.submit "Request Invitation", :class => "btn btn-primary", :id =>
'invitation_button' %>
      <a class="btn" data-dismiss="modal" href="#">Close</a>
    </div>
  <% end %>
</div>
<div id="romance-copy" style="text-align: center; margin-top: 80px">
  <h2>Want in?</h2>
</div>
<div id="call-to-action" style="text-align: center; margin-top: 80px">
  <a class="btn btn-primary btn-large" data-toggle="modal"
href="#request-invite">Request invite</a>
</div>
```

The revised view code includes CSS classes from Twitter Bootstrap that implement the modal window and apply style to the buttons. You'll note that our code includes some simple style rules to position the romance copy and the call-to-action button. These styles could be moved to an external stylesheet; for now, it's easier to include them in the view file.

Display Errors

Displaying the "request invitation" form inside a modal window is a nice touch but it creates a problem: Form validation error messages are hidden. See for yourself by running the application. Open the modal window, leave the email field blank and click the submit button. You'll see the home page without any indication that there was a problem. Click the home page "Request Invite" button again and you'll see there is an error message in the modal window.

We want to force the form to be displayed when errors are present. We can do this by modifying the first line of the file **app/views/devise/registrations/new.html.erb**:

```
<div id="request-invite" class="modal" style="display: <%= @user.errors.any? ? 'block' :
'none';%>">
```

When you test this by submitting a form without an email address, you'll see the form appears with error messages.

Adding Ajax

Thank you to Andrea Pavoni for the Ajax implementation.

We've improved the appearance of the "invitation request" page by adding a modal window using Twitter Bootstrap. We can make further improvements.

In the simplest web applications, clicking a link or a button opens a new page. Since 2005 or so, users have become accustomed to buttons that refresh a page with new data from a web server without requiring a full page reload. This kind of interaction requires asynchronous JavaScript (Ajax). It is very common now but once was considered an innovative Web 2.0 technology and identified with the newest and most sophisticated websites. As implemented, the page refreshes with a roundtrip to the server when we submit the form successfully (or get an an error). Our app will look more sophisticated if we use Ajax techniques to update the page without a page refresh.

We don't need changes to the "invitation request" page. We'll add a partial to provide a "thank you" message. Add a file **app/views/devise/registrations/_thankyou.html.erb**:

```
<h1>Thank you</h1>
<div id="request-invite" class="modal" style="display: 'block';">
  <div class="modal-header">
    <a class="close" data-dismiss="modal">&#215;</a>
    <h3>Thank you!</h3>
  </div>
  <div class="modal-body" style="margin-bottom: 120px; overflow: visible">
    <p>We have received your request for an invitation to example.com.</p>
    <p>We'll contact you when we launch.</p>
  </div>
</div>
```

Next, add some JavaScript to the file **app/assets/javascripts/application.js**. This will trigger an Ajax submission action when the "request invitation" button is clicked and update the `#request-invite` div on completion.

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require_tree .
$('document').ready(function() {

  // display validation errors for the "request invitation" form
  if ($('.alert-error').length > 0) {
    $("#request-invite").modal('toggle');
  }

  // use Ajax to submit the "request invitation" form
  $('#invitation_button').on('click', function() {
    var email = $('form #user_email').val();
    var dataString = 'user[email]='+ email;
    $.ajax({
      type: "POST",
      url: "/users",
      data: dataString,
      success: function(data) {
        $('#request-invite').html(data);
      }
    });
    return false;
  });

})
```

This JavaScript code will run on every page of the application. Since our application is simple, this is not an issue. As you begin to customize your application, you might want to refactor this as page-specific JavaScript. The Paloma gem offers an easy way to organize JavaScript files using the Rails asset pipeline. It provides a capability to execute page-specific JavaScript. See the article Rails and JavaScript.

Finally, we need to override the `create` method in the Devise registration controller to render the “thank you” partial. Modify the file **app/controllers/registrations_controller.rb**:

```
class RegistrationsController < Devise::RegistrationsController

  # override #create to respond to Ajax with a partial
  def create
    build_resource

    if resource.save
      if resource.active_for_authentication?
        sign_in(resource_name, resource)
        (render(:partial => 'thankyou', :layout => false) && return)  if request.xhr?
        respond_with resource, :location => after_sign_up_path_for(resource)
      else
        expire_session_data_after_sign_in!
        (render(:partial => 'thankyou', :layout => false) && return)  if request.xhr?
        respond_with resource, :location => after_inactive_sign_up_path_for(resource)
      end
    else
      clean_up_passwords resource
      render :action => :new, :layout => !request.xhr?
    end
  end

  protected

  def after_inactive_sign_up_path_for(resource)
    # the page prelaunch visitors will see after they request an invitation
    # unless Ajax is used to return a partial
    '/thankyou.html'
  end

  def after_sign_up_path_for(resource)
    # the page new users will see after sign up (after launch, when no invitation is
needed)
    redirect_to root_path
  end

end
```

Now we can say we have built a “Web 2.0” application. When the visitor submits the form, the modal window changes to display a “thank you” message (or an error message) without a page refresh.

Test the Application

You can check that the application 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 "Request Invitation" form in a modal window. Submitting an invitation request should display the "thank you" message in the modal window.

Git Workflow

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

```
$ git add -A
$ git commit -m "improve design for 'Request Invitation' feature"
```

Merge the working branch to "master":

```
$ git checkout master
$ git merge --squash modal
$ git commit -m "improve design for 'Request Invitation' feature"
$ git branch -D modal
```

Chapter 18

Admin Page

As the owner of the site, you'll want to see how many visitors have requested invitations. You likely don't want anyone else to view that information, so we'll need some form of authorization to restrict access to yourself or approved administrators.

Fortunately, the rails3-bootstrap-devise-cancan starter application already has created an administrative page that is access controlled. We'll take a look at how it is implemented here but we don't need to add any functionality.

User Story

Here's the user story for the administrative feature:

```
*View Progress*
As the owner of the site
I want to know how many visitors have requested invitations
so I can know if my offer is popular
```

The Administrative Page

We have a page that shows a list of users.

Take a look at the file **app/views/users/index.html.erb**:

```
<h3>Users</h3>
<div class="span8">
<table class="table table-condensed">
  <thead>
    <tr>
      <th>Username</th>
      <th>Email</th>
      <th>Registered</th>
      <th>Role</th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
  <% @users.each do |user| %>
    <tr>
      <td><%= link_to user.name, user %></td>
      <td><%= user.email %></td>
      <td><%= user.created_at.to_date %></td>
      <td><%= user.roles.first.name.titleize unless user.roles.first.nil? %></td>
      <td>
        <a data-toggle="modal" href="#role-options-<%= user.id %>" class="btn btn-mini"
type="button">Change role</a>
        <%= render user %>
      </td>
      <td><%= link_to("Delete user", user_path(user), :data => { :confirm => "Are you
sure?" }, :method => :delete, :class => 'btn btn-mini') unless user == current_user
%></td>
    </tr>
  <% end %>
  </tbody>
</table>
</div>
```

We can use this page as our "administrative dashboard." It lists each user with an email address, the date the user registered, the user's role (either Admin or User), and a button to change the user's role. The page is provided with the rails3-bootstrap-devise-cancan starter application.

Restricting Access to the Administrative Page

Take a look at the controller 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])
    if @user.update_attributes(params[:user], :as => :admin)
      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
```

Notice the index method contains a statement which limits access to administrators only:

```
authorize! :index, @user, :message => 'Not authorized as an administrator.'
```

When we generated our starter app, the application template set up everything we need for limiting access to administrative pages. You can take a look at the tutorial for the rails3-bootstrap-devise-cancan application to see how it is done. In a nutshell, CanCan provides an `Ability` class with an `authorize!` method. We add the `authorize!` method to any controller action that should be restricted to administrators only. The `authorize!` method makes a call to the `Ability` class to determine which users are authorized for access.

If you look at the file **app/models/ability.rb**, you'll see this:

```
class Ability
  include CanCan::Ability

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

end
```

Very simply, if we add the `authorize!` method to any controller action, it will check the `Ability` class and allow access for users in the "admin" role. By default, other users will not be allowed. Of course, if the `authorize!` method is not included in a controller action, any user will have access.

We haven't made any changes to the administrative page provided by the starter application, so there's nothing to commit to git.

Chapter 19

Send Invitations

As implemented, our application collects invitation requests from visitors. Our administrative dashboard displays a list of visitors' email addresses. This is all you need while you are promoting and building your application. But when you ready for users to begin trying your site, you'll want to select users for your beta test, send each an invitation, and ask each to confirm their email address and set an account password. We'll call this functionality the "Send Invitations" feature.

Git Workflow

Create a new git branch for this feature:

```
$ git checkout -b invitations
```

User Story

Here's the user story we'll specify and implement:

```
*Send Invitations*
As the owner of the site
I want to send invitations to visitors who have requested invitations
so users can try the site
```

Implementation

We'll add a "send invitation" link to each user listed on the administrative dashboard. We'll need a corresponding `invite` action in the user controller and a matching route. We also want to make sure only an administrator is authorized to initiate the `invite` action.

Devise already knows how to send an account confirmation email to a user, with its `user.send_confirmation_instructions` method, which will generate a confirmation token and send an email message to the user. We'll call the Devise `user.send_confirmation_instructions` method from the `invite` action in the user controller.

Add a User Controller Action

So far, our application architecture has been very simple. We've used a few RESTful actions supplied by Devise and improved on a simple `index` method to prepare our administrative dashboard page. Now we'll need a custom `invite` action in the user controller.

Modify the controller file **app/controllers/users_controller.rb** by adding an `invite` action:

```
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])
    if @user.update_attributes(params[:user], :as => :admin)
      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

  def invite
    authorize! :invite, @user, :message => 'Not authorized as an administrator.'
    @user = User.find(params[:id])
    @user.send_confirmation_instructions
    redirect_to :back, :only_path => true, :notice => "Sent invitation to
#{@user.email}."
  end

end
```

CanCan's `authorize!` method ensures that only an administrator is authorized to initiate the `invite` action.

The `invite` action will use Devise's supplied `send_confirmation_instructions` method to generate a confirmation token and send an email to the selected user. Then it will redirect the administrator back to a previous page. The redirect method takes an extra argument

`:only_path => true` to avoid a potential security vulnerability. Without this precaution, the Brakeman security scanner warns of unvalidated redirects.

Add a Route

Modify **config/routes.rb** to add the new action. Replace `resources :users` with:

```
resources :users do
  get 'invite', :on => :member
end
```

The Rails Guide, Routing from the Outside In, shows how we add additional routes to a RESTful resource.

Sending Invitations from the Administrative Page

We will modify the file **app/views/users/index.html.erb** extensively in this step. We'll add a link to the `invite` action for each user. While we're at it, we'll add columns to show when the visitor joined (that is, confirmed their account), the number of times they've logged in, and the date of the most recent login. We'll change the table headings and widen the table to `class="span12"` to accommodate the extra columns.

Replace the contents of **app/views/users/index.html.erb** with this code:

```
<h3>Users</h3>
<div class="span12">
  <table class="table table-condensed">
    <thead>
      <tr>
        <th>Username</th>
        <th>Email</th>
        <th>Requested</th>
        <th>Invitation</th>
        <th>Joined</th>
        <th>Visits</th>
        <th>Most Recent</th>
        <th>Role</th>
        <th></th>
        <th></th>
      </tr>
    </thead>
    <tbody>
    <% @users.each do |user| %>
      <tr>
        <td><%= link_to user.name, user %></td>
        <td><%= user.email %></td>
        <td><%= user.created_at.to_date %></td>
        <td><%= (user.confirmation_token.nil? ? (link_to "send invitation",
invite_user_path(user), {:class => 'btn btn-mini'}) : (link_to "resend",
invite_user_path(user), {:class => 'btn btn-mini'})) unless user.confirmed_at %></td>
        <td><%= user.confirmed_at.to_date if user.confirmed_at %></td>
        <td><%= user.sign_in_count if user.sign_in_count %></td>
        <td><%= user.last_sign_in_at.to_date if user.last_sign_in_at %></td>
        <td><%= user.roles.first.name.titleize unless user.roles.first.nil? %></td>
        <td>
          <a data-toggle="modal" href="#role-options-<%= user.id %>" class="btn
btn-mini" type="button">Change role</a>
          <%= render user %>
        </td>
        <td><%= link_to("Delete user", user_path(user), :data => { :confirm => "Are you
sure?" }, :method => :delete, :class => 'btn btn-mini') unless user == current_user
%></td>
      </tr>
    <% end %>
    </tbody>
  </table>
</div>
```

Now, as the site administrator, you can invite individual users to complete the account confirmation process and obtain access to the site.

The logic is complex in the display of the “send invitation” link. First, with `unless user.confirmed_at`, we check to see if the user has already confirmed the account. Next,

we check the user's `confirmation_token` attribute to see if we've already sent an invitation (in which case, a confirmation token was set by Devise). If no confirmation token was set, we display the link "send invitation"; otherwise, we display "resend". Note we also apply the "btn btn-mini" class to make the link a miniature button.

Customizing the Devise Confirmation Email

Here's the mailer view that Devise uses for the confirmation email:

```
<p>Welcome <%= @resource.email %>!</p>

<p>You can confirm your account email through the link below:</p>

<p><%= link_to 'Confirm my account', confirmation_url(@resource, :confirmation_token =>
@resource.confirmation_token) %></p>
```

You may want to customize the confirmation email to be more attractive or informative.

Devise hides all its logic and views (including mailer templates) inside the Devise gem package. Devise is like any other Rails engine; we can override its views in our application.

First, create a folder **app/views/devise/mailer**.

Create a file **app/views/devise/mailer/confirmation_instructions.html.erb**:

```
<p>Welcome <%= @resource.email %>!</p>

<p>We're pleased to invite you to try <%= link_to ENV["DOMAIN"], root_url %>.</p>

<p>Please click the link below to confirm your email address and set your password:</p>

<p><%= link_to 'Confirm my account', confirmation_url(@resource, :confirmation_token =>
@resource.confirmation_token) %></p>
```

Here's a tip. If you're using MailChimp or another mailing list service, you can use the templates offered by the service as a basis for your confirmation email message. For example, start with one of the templates suggested for a welcome message. Copy the HTML version of the message and use it for your confirmation email message.

Customizing Other Devise Emails

When you are ready to launch your website, you may want to customize other Devise email messages. Here are the other messages Devise provides.

"Reset Password" instructions can be overridden by creating a file **app/views/devise/mailer/reset_password_instructions.html.erb**:

```
<p>Hello <%= @resource.email %>!</p>

<p>Someone has requested a link to change your password, and you can do this through
the link below.</p>

<p><%= link_to 'Change my password', edit_password_url(@resource, :reset_password_token
=> @resource.reset_password_token) %></p>

<p>If you didn't request this, please ignore this email.</p>
<p>Your password won't change until you access the link above and create a new one.</p>
```

"Unlock Account" instructions can be overridden by creating a file **app/views/devise/mailer/unlock_instructions.html.erb**:

```
<p>Hello <%= @resource.email %>!</p>

<p>Your account has been locked due to an excessive amount of unsuccessful sign in
attempts.</p>

<p>Click the link below to unlock your account:</p>

<p><%= link_to 'Unlock my account', unlock_url(@resource, :unlock_token =>
@resource.unlock_token) %></p>
```

Bulk Invitations

If you are selecting only a few dozen initial users, this process of manual selection will be adequate. If you are ready to launch and want to invite hundreds or thousands of users, you'll need a way to invite multiple users with a single action. We need to implement a "bulk invitations" feature.

You should set up an SMTP relay service such as Mandrill or SendGrid before you attempt to send more than a few dozen email messages. The Configuration chapter of this tutorial showed how to set up Mandrill. See the article Send Email with Rails for details.

We'll add a `bulk_invite` action to the controller 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])
    if @user.update_attributes(params[:user], :as => :admin)
      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

  def invite
    authorize! :invite, @user, :message => 'Not authorized as an administrator.'
    @user = User.find(params[:id])
    @user.send_confirmation_instructions
    redirect_to :back, :only_path => true, :notice => "Sent invitation to
#{@user.email}."
  end

  def bulk_invite
    authorize! :bulk_invite, @user, :message => 'Not authorized as an administrator.'
    users = User.where(:confirmation_token =>
nil).order(:created_at).limit(params[:quantity])
    count = users.count
    users.each do |user|
      user.send_confirmation_instructions
    end
    redirect_to :back, :only_path => true, :notice => "Sent invitation to #{count}
```

```
users."
  end

end
```

The `bulk_invite` action has a few peculiarities. First, we only select users where the `:confirmation_token` attribute is null. Devise sets a confirmation token when we send an invitation; with this constraint, we only invite users who have not been previously invited. Second, we make sure we select the oldest records first with the `order(:created_at)` method. We limit the number of records that are retrieved to the size of the batch we want; then we loop over each to send an invitation. Finally we set a notice and return to the admin page (we use the extra argument `:only_path => true` to avoid a potential security vulnerability).

Replace **config/routes.rb** to add the new action:

```
RailsPrelaunchSignup::Application.routes.draw do
  authenticated :user do
    root :to => 'home#index'
  end
  devise_scope :user do
    root :to => "devise/registrations#new"
    match '/user/confirmation' => 'confirmations#update', :via => :put, :as =>
:update_user_confirmation
  end
  devise_for :users, :controllers => { :registrations => "registrations" }
  match 'users/bulk_invite/:quantity' => 'users#bulk_invite', :via => :get, :as =>
:bulk_invite
  resources :users do
    get 'invite', :on => :member
  end
end
```

If you've named the application "RailsPrelaunchSignup" as we suggested, you can copy the example code above. If not, be careful not to change the application name in your file.

Notice that we add two new routes. The route `match '/user/confirmation'` is there to accommodate the confirmation link in the email message received by invited users. The route `match 'users/bulk_invite/:quantity'` handles the `bulk_invite` action we initiate from the administrative dashboard.

Finally, we will modify the file **app/views/users/index.html.erb** to add links for the `bulk_invite` action:

```
<h3>Users</h3>
<div class="span12">
  <p>
    Send Bulk Invitations:
    <%= link_to "10 &#183;".html_safe, bulk_invite_path(:quantity => '10') %>
    <%= link_to "50 &#183;".html_safe, bulk_invite_path(:quantity => '50') %>
    <%= link_to "100 &#183;".html_safe, bulk_invite_path(:quantity => '100') %>
    <%= link_to "500 &#183;".html_safe, bulk_invite_path(:quantity => '500') %>
    <%= link_to "1000", bulk_invite_path(:quantity => '1000') %>
  </p>
  <table class="table table-condensed">
    <thead>
      <tr>
        <th>Username</th>
        <th>Email</th>
        <th>Requested</th>
        <th>Invitation</th>
        <th>Joined</th>
        <th>Visits</th>
        <th>Most Recent</th>
        <th>Role</th>
        <th></th>
        <th></th>
      </tr>
    </thead>
    <tbody>
    <% @users.each do |user| %>
      <tr>
        <td><%= link_to user.name, user %></td>
        <td><%= user.email %></td>
        <td><%= user.created_at.to_date %></td>
        <td><%= (user.confirmation_token.nil? ? (link_to "send invitation",
invite_user_path(user), {:class => 'btn btn-mini'}) : (link_to "resend",
invite_user_path(user), {:class => 'btn btn-mini'})) unless user.confirmed_at %></td>
        <td><%= user.confirmed_at.to_date if user.confirmed_at %></td>
        <td><%= user.sign_in_count if user.sign_in_count %></td>
        <td><%= user.last_sign_in_at.to_date if user.last_sign_in_at %></td>
        <td><%= user.roles.first.name.titleize unless user.roles.first.nil? %></td>
        <td>
          <a data-toggle="modal" href="#role-options-<%= user.id %>" class="btn
btn-mini" type="button">Change role</a>
          <%= render user %>
        </td>
        <td><%= link_to("Delete user", user_path(user), :data => { :confirm => "Are you
sure?" }, :method => :delete, :class => 'btn btn-mini') unless user == current_user
%></td>
      </tr>
    <% end %>
    </tbody>
```

```
  </table>
</div>
```

We now have the option to send 10, 50, 100, 500, or 1000 invitations at once.

Git Workflow

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

```
$ git add -A
$ git commit -m "enable admin to send invitations"
```

```
$ git checkout master
$ git merge --squash invitations
$ git commit -m "enable admin to send invitations"
```

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

```
$ git branch -D invitations
```

Chapter 20

Setting the User's Password

Once the user has been invited and has confirmed the account, we need to provide a way for the user to set a password.

The Devise wiki explains How to Override Confirmations So Users Can Pick Their Own Passwords As Part of Confirmation Activation. So far, we've made small modifications to customize the behavior of Devise; now we'll make extensive modifications, overriding the Confirmations controller and view as well as adding methods to the User model.

Git Workflow

Create a new git branch for this improvement:

```
$ git checkout -b fix-password
```

Customize the Confirmations View

When an invited user clicks a link in the invitation email to confirm an account, we want to ask them to choose a password for their account. The Devise gem supplies a Confirmations view; we need to replace it with a page that includes fields for password and password confirmation.

Create a new folder **app/views/devise/confirmations/**.

Create a file **app/views/devise/confirmations/show.html.erb**:

```
<h2>Account Activation</h2>
<%= simple_form_for resource, :as => resource_name, :url =>
update_user_confirmation_path, :html => {:class => 'form-horizontal', :method => 'put'},
:id => 'activation-form' do |f| %>
  <%= devise_error_messages! %>
  <fieldset>
    <legend>
      Account Activation
      <% if resource.email %>
        for <%= resource.email %>
      <% end %>
    </legend>
    <% if @requires_password %>
      <p>
        <%= f.label :password,'Choose a Password:' %>
        <%= f.password_field :password %>
      </p>
      <p>
        <%= f.label :password_confirmation, 'Password Confirmation:' %>
        <%= f.password_field :password_confirmation %>
      </p>
    <% end %>
    <%= hidden_field_tag :confirmation_token, @confirmation_token %>
    <p><%= f.submit "Activate" %></p>
  </fieldset>
<% end %>
```

Customize the Confirmations Controller

The Devise gem supplies a Confirmations controller; we need to replace it with one that will accommodate our customized view that asks the user to set a password. We'll override the Devise Confirmations controller by creating our own Confirmations controller. Often, in overriding a controller, we inherit from the original controller. In this case, we'll inherit from the Devise Passwords controller that provides methods to set the user's password.

Create a file **app/controllers/confirmations_controller.rb**:

```
class ConfirmationsController < Devise::PasswordsController
  skip_before_filter :require_no_authentication
  skip_before_filter :authenticate_user!

  # POST /resource/confirmation
  def create
    self.resource = resource_class.send_confirmation_instructions(resource_params)
    if successfully_sent?(resource)
      respond_with({}, :location =>
after_resending_confirmation_instructions_path_for(resource_name))
    else
      respond_with(resource)
    end
  end

  # PUT /resource/confirmation
  def update
    with_unconfirmed_confirmable do
      if @confirmable.has_no_password?
        @confirmable.attempt_set_password(params[:user])
        if @confirmable.valid?
          do_confirm
        else
          do_show
          @confirmable.errors.clear #so that we won't render :new
        end
      else
        self.class.add_error_on(self, :email, :password_allready_set)
      end
    end

    if !@confirmable.errors.empty?
      render 'devise/confirmations/new'
    end
  end

  # GET /resource/confirmation?confirmation_token=abcdef
  def show
    with_unconfirmed_confirmable do
      if @confirmable.has_no_password?
        do_show
      else
        do_confirm
      end
    end
    if !@confirmable.errors.empty?
      render 'devise/confirmations/new'
    end
  end
```

```
  protected

  def with_unconfirmed_confirmable
    @confirmable = User.find_or_initialize_with_error_by(:confirmation_token,
params[:confirmation_token])
    self.resource = @confirmable
    if !@confirmable.new_record?
      @confirmable.only_if_unconfirmed {yield}
    end
  end

  def do_show
    @confirmation_token = params[:confirmation_token]
    @requires_password = true
    render 'devise/confirmations/show'
  end

  def do_confirm
    @confirmable.confirm!
    set_flash_message :notice, :confirmed
    sign_in_and_redirect(resource_name, @confirmable)
  end

  # The path used after resending confirmation instructions.
  def after_resending_confirmation_instructions_path_for(resource_name)
    new_session_path(resource_name)
  end

  # The path used after confirmation.
  def after_confirmation_path_for(resource_name, resource)
    after_sign_in_path_for(resource)
  end

end
```

Add a Route

We'll need to modify the route:

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

to look like this:

```
devise_for :users, :controllers => { :registrations => "registrations", :confirmations
=> "confirmations" }
```

The **config/routes.rb** should look like this:

```
RailsPrelaunchSignup::Application.routes.draw do
  authenticated :user do
    root :to => 'home#index'
  end
  devise_scope :user do
    root :to => "devise/registrations#new"
    match '/user/confirmation' => 'confirmations#update', :via => :put, :as =>
:update_user_confirmation
  end
  devise_for :users, :controllers => { :registrations => "registrations", :confirmations
=> "confirmations" }
  match 'users/bulk_invite/:quantity' => 'users#bulk_invite', :via => :get, :as =>
:bulk_invite
  resources :users do
    get 'invite', :on => :member
  end
end
```

If you've named the application "RailsPrelaunchSignup" as we suggested, you can copy the example code above. If not, be careful not to change the application name in your file.

Modify the User Model

We need to modify the User model to allow the new user to set a password when they confirm their account.

Add the following methods to the file **app/models/user.rb** above the `private` keyword:

```
# new function to set the password
def attempt_set_password(params)
  p = {}
  p[:password] = params[:password]
  p[:password_confirmation] = params[:password_confirmation]
  update_attributes(p)
end

# new function to determine whether a password has been set
def has_no_password?
  self.encrypted_password.blank?
end

# new function to provide access to protected method pending_any_confirmation
def only_if_unconfirmed
  pending_any_confirmation {yield}
end
```

Git Workflow

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

```
$ git add -A
$ git commit -m "enable a user to set a password"
```

Merge the new code into the master branch and commit it:

```
$ git checkout master
$ git merge --squash fix-password
$ git commit -m "enable a user to set a password"
```

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

```
$ git branch -D fix-password
```

Chapter 21

Collect Email Addresses

You might like to send announcements or newsletters to all your visitors before you launch the site.

You could implement some code to iterate through each user record and send each an email message directly from the application. That is a bad idea because your application would have to open a connection to your SMTP server for every message (which are all identical). Some SMTP relay services such as SendGrid provide an API call that makes it easy to send an array of email addresses in a single request for processing by the relay service. That's an option. But the most practical approach is to use a dedicated service such as MailChimp for managing and sending broadcast email. You won't tie up your app sending bulk email and you'll be able to use services such as delivery tracking and well-designed templates.

When each visitor requests an invitation, you'll need to subscribe them to a MailChimp mailing list.

Git Workflow

Create a new git branch for this feature:

```
$ git checkout -b mailchimp
```

Gibbon Gem

The Gibbon gem is a convenient wrapper for the MailChimp API.

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

```
gem "gibbon"
```

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

Modify the User Model

We already have a `send_welcome_email` method in the User model. That was useful for checking that we can send email from our application. Now we'll remove it. We could continue to use it but MailChimp offers an option to send a welcome message when we subscribe a new user to the mailing list. MailChimp offers nice templates and tracks deliverability so we'll use the MailChimp feature. You might have already created the welcome message for the mailing list; see Accounts You May Need for details if you haven't done so already.

After we remove the `send_welcome_email` method, we'll add the following two methods:

- `add_user_to_mailchimp`
- `remove_user_from_mailchimp`

We'll remove the filter:

- `after_create :send_welcome_email`

and add two filters:

- `after_create :add_user_to_mailchimp`
- `before_destroy :remove_user_from_mailchimp`

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

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

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

  after_create :add_user_to_mailchimp
  before_destroy :remove_user_from_mailchimp

  # override Devise method
  # no password is required when the account is created; validate password when the
user sets one
  validates_confirmation_of :password
  def password_required?
    if !persisted?
      !(password != "")
    else
      !password.nil? || !password_confirmation.nil?
    end
  end

  # override Devise method
  def confirmation_required?
    false
  end

  # override Devise method
  def active_for_authentication?
    confirmed? || confirmation_period_valid?
  end

  def send_reset_password_instructions
    if self.confirmed?
      super
    else
      errors.add :base, "You must receive an invitation before you set your password."
    end
  end

  # new function to set the password
  def attempt_set_password(params)
    p = {}
    p[:password] = params[:password]
    p[:password_confirmation] = params[:password_confirmation]
    update_attributes(p)
  end
```

```
  # new function to determine whether a password has been set
  def has_no_password?
    self.encrypted_password.blank?
  end

  # new function to provide access to protected method pending_any_confirmation
  def only_if_unconfirmed
    pending_any_confirmation {yield}
  end

  private

  def add_user_to_mailchimp
    return if email.include?(ENV['ADMIN_EMAIL'])
    mailchimp = Gibbon::API.new
    result = mailchimp.lists.subscribe({
      :id => ENV['MAILCHIMP_LIST_ID'],
      :email => {:email => self.email},
      :double_optin => false,
      :update_existing => true,
      :send_welcome => true
      })
    Rails.logger.info("Subscribed #{self.email} to MailChimp") if result
  end

  def remove_user_from_mailchimp
    mailchimp = Gibbon::API.new
    result = mailchimp.lists.unsubscribe({
      :id => ENV['MAILCHIMP_LIST_ID'],
      :email => {:email => self.email},
      :delete_member => true,
      :send_goodbye => false,
      :send_notify => true
      })
    Rails.logger.info("Unsubscribed #{self.email} from MailChimp") if result
  end

end
```

We've removed one method, `send_welcome_email`, and added two methods:

- `add_user_to_mailchimp`
- `remove_user_from_mailchimp`

and added the necessary filters.

We remove the `send_welcome_email` method because MailChimp gives us an option to send a welcome email when a visitor subscribes to a mailing list. It's easier to compose an attractive

welcome email using MailChimp's templates than to create one from scratch. And MailChimp tracks whether the message has been delivered and opened. Be sure to log in to MailChimp and create a welcome email message for members of the "visitors" email list before testing the application.

The `add_user_to_mailchimp` method will connect with the MailChimp server and issue an API call to subscribe the user to the mailing list. We exit the method immediately if the email address is the address for the administrator set in the **config/application.yml** file because we don't want to subscribe the administrator when we run `rake db:seed` or `rake db:reset`.

We instantiate the Gibbon object which provides all the connectivity. The Gibbon gem looks in the environment variables for the `MAILCHIMP_API_KEY` value so we don't need to specify it here. We assign the Gibbon object to the `mailchimp` variable (we could name it anything).

Gibbon offers a `lists.subscribe` method which takes five parameters:

- id – environment variable to identify the MailChimp list
- email – address of the visitor (inside a hash)
- double_optin – setting `true` sends a double opt-in confirmation message
- update_existing – updates a subscriber record if it already exists
- send_welcome – sends a "Welcome Email" to the new subscriber

The parameters are described further in the MailChimp API Documentation.

If the application successfully adds the new subscriber, we write a message to the logger.

If we get an error when trying to add the subscriber, Gibbon will raise an exception.

We add the `remove_user_from_mailchimp` method to delete the user as necessary. Refer to the documentation for the MailChimp API Documentation for an explanation of the options. We've elected not to `send_goodbye`; the `send_notify` is set to true in case you've set your MailChimp settings to send notification emails to yourself.

Before you test the feature, be sure you've added the `MAILCHIMP_LIST_ID` to the **config/application.yml** file.

Our "Collect Email Addresses" feature is complete.

Git Workflow

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

```
$ git add -A
$ git commit -m "collect email addresses with MailChimp"
```

Merge the new code into the master branch and commit it:

```
$ git checkout master
$ git merge --squash mailchimp
$ git commit -m "collect email addresses with MailChimp"
```

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

```
$ git branch -D mailchimp
```

Chapter 22

Social Sharing

If your visitors are excited about your offer, they may want to share their discovery with friends. You can encourage visitors to share on Facebook and Twitter by providing "social sharing" buttons with your "thank you" message. We'll also include Google+; if you want to add social sharing buttons for other services you'll find the code is very similar.

User Story

Here's the user stories we'll implement:

```
*Post to Twitter After Sign Up*
As a user
I want an option to post to Twitter after I sign up
so my followers will learn about the site

*Post to Facebook After Sign Up*
As a user
I want an option to post to Facebook after I sign up
so my Facebook friends will learn about the site
```

Git Workflow

Create a new git branch for this feature:

```
$ git checkout -b social-share
```

Implementation of Social Sharing

Social sharing buttons for Twitter, Facebook, and Google+ are implemented in HTML and JavaScript. There are several gems that provide Rails view helpers for "Facebook Like" and "Tweet" buttons. And there are numerous jQuery plugins for the same. But it's easy to add the markup directly.

Twitter, Facebook, and Google+ give you documentation and tools to create social sharing buttons for your website:

- Twitter Tweet Button
- Facebook Like Button
- Google+ +1 Button

We won't use the example code that the social networks provide; instead, we'll set up our page to download JavaScript widgets from each of the social networks. The implementation is similar for each service. A JavaScript widget determines if a link or div is present on the page and applies a set of transformations to create a graphical button.

JavaScript for Social Sharing Buttons

Modify the file **app/assets/javascripts/application.js**:

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require_tree .
$('document').ready(function() {

  // display validation errors for the "request invitation" form
  if ($('.alert-error').length > 0) {
    $("#request-invite").modal('toggle');
  }

  // use Ajax to submit the "request invitation" form
  $('#invitation_button').on('click', function() {
    var email = $('form #user_email').val();
    var dataString = 'user[email]='+ email;
    $.ajax({
      type: "POST",
      url: "/users",
      data: dataString,
      success: function(data) {
        $('#request-invite').html(data);
        loadSocial();
      }
    });
    return false;
  });

})

// load social sharing scripts if the page includes a Twitter "share" button
function loadSocial() {

    //Twitter
    if (typeof (twttr) != 'undefined') {
      twttr.widgets.load();
    } else {
      $.getScript('http://platform.twitter.com/widgets.js');
    }

    //Facebook
    if (typeof (FB) != 'undefined') {
      FB.init({ status: true, cookie: true, xfbml: true });
    } else {
      $.getScript("http://connect.facebook.net/en_US/all.js#xfbml=1", function () {
        FB.init({ status: true, cookie: true, xfbml: true });
      });
    }

    //Google+
    if (typeof (gapi) != 'undefined') {
      $(".g-plusone").each(function () {
```

```
        gapi.plusone.render($(this).get(0));
      });
    } else {
      $.getScript('https://apis.google.com/js/plusone.js');
    }
}
```

Notice we add a call to the `loadSocial()` function within the `success` callback of the `$.ajax` function. We also define the `loadSocial()` function.

We call the function `loadSocial()` after the "request invitation" form is successfully submitted. After the Ajax call returns "success", we render the "#request-invite" div with the **_thankyou.html.erb** partial (defined in the RegistrationsController); immediately following, we call the function `loadSocial()`. The `loadSocial()` function uses the jQuery `.getScript` method to load a JavaScript file from each of the social networks. The script won't attempt to download a JavaScript file if the widget is already present.

Add Social Sharing Buttons to the "Thank You" Message

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

```
<h1>Thank you</h1>
<div id="request-invite" class="modal" style="display: 'block';">
  <div class="modal-header">
    <a class="close" data-dismiss="modal">&#215;</a>
    <h3>Thank you!</h3>
  </div>
  <div class="modal-body" style="margin-bottom: 120px; overflow: visible">
    <p>We have received your request for an invitation to example.com.</p>
    <p>We'll contact you when we launch.</p>
    <p>Share your discovery with your friends!</p>
    <div id="tweet" style="display:inline-block;">
      <a class="twitter-share-button" data-count="horizontal" data-text="A new site I've
discovered" data-url="http://railsapps.github.io/rails-prelaunch-signup/"
data-via="example" data-related="example" href="https://twitter.com/share"></a>
    </div>
    <span style="display:inline-block; vertical-align:text-bottom;">
      <div class="fb-like" data-href="http://railsapps.github.io/
rails-prelaunch-signup/" data-layout="button_count" data-send="false"
data-show-faces="false" data-width="90">
      </div>
    </span>
    <div class="g-plusone" data-annotation="inline"
data-href="http://railsapps.github.io/rails-prelaunch-signup/" data-size="medium"
data-width="120">
    </div>
  </div>
</div>
```

The results are pretty: graphical buttons for Twitter, Facebook, and Google+ sharing aligned nicely in a row. To get that result, we need a mess of elements with tweaked style rules. Twitter expects a link styled with the class `twitter-share-button`; the Twitter JavaScript widget then transforms it into a dynamic button that displays a count of past tweets. We have to apply a `display:inline-block` style rule to force it to appear in a row with the other buttons. The Facebook JavaScript widget expects a div with the class `fb-like`; we wrap it in a span element and apply some style rules to force it to align nicely. The Google+ JavaScript widget expects a div with the class `g-plusone`; this one plays nicely with others and doesn't need any tweaking. One last tweak to the class `modal-body` applies a style rule that increases the height of the modal window so there's room for the dialog box that appears when the visitor clicks the Facebook "Like" button.

For demonstration purposes we've set up the social sharing buttons to point to the project page for the Rails Prelaunch Signup app. You'll want to replace the placeholders with pointers to your own website:

Item	Placeholder
Twitter data-via	example

Twitter data-related	example
Twitter data-text	A new site I've discovered (default text for the tweet)
Twitter data-url	http://railsapps.github.io/rails-prelaunch-signup/
Facebook data-href	http://railsapps.github.io/rails-prelaunch-signup/
Google+ data-href	http://railsapps.github.io/rails-prelaunch-signup/

Git Workflow

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

```
$ git add -A
$ git commit -m "add social sharing buttons"
```

Merge the new code into the master branch and commit it:

```
$ git checkout master
$ git merge --squash social-share
$ git commit -m "add social sharing buttons"
```

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

```
$ git branch -D social-share
```

Chapter 23
Testing

Be sure to set all required settings in the **config/application.yml** file.

Initialize the database by entering:

```
$ rake db:reset
```

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 "Request invite" button on the home page. When you click on the button, you should see the modal window for "Request Invitation." Enter an email address and submit the form.

You should see a "thank you" confirmation message in the modal window.

Click the "Tweet" button to see a pop-up window that gives you an option to post a tweet (you can cancel if you don't want to tweet). Clicking the Facebook and Google+ buttons result in immediate updates to your Facebook and Google+ pages.

Check for email at the address you entered. You should find the welcome message for the MailChimp list.

Close the modal window and log in with the administrator's account. To sign in as the administrator (unless you've changed it), use:

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

Click "Admin" in the navbar for the administrative dashboard. You'll see a list of users.

Click the "send invitation" button and check the server log file displayed in the console window. You should see the text from an invitation email message in the console window.

If you've set the **config/environments/development.rb** file to include

```
config.action_mailer.perform_deliveries = true
```

the application will send an invitation to the user's email address.

Click the "Confirm my account" link in the invitation message (or copy it from the console window). You should see the "Account Activation" page. Create a new password and click "Activate." You should see the application home page with the message, "Your account was successfully confirmed. You are now signed in."

If you log in as the administrator, you'll see the date the user joined and most recently logged in.

Stop the server with Control-C.

Chapter 24

Deploy

Heroku provides low cost, easily configured Rails application hosting.

For your convenience, here is a Tutorial for Rails on Heroku. See the article for details about preparing your application to deploy to Heroku.

Be sure to set up SSL before you make your application available in production. See the Heroku documentation on SSL or use CloudFlare as described in the Accounts You May Need section.

After you've prepared your application as described in the Tutorial for Rails on Heroku article, 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
```

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 using the `heroku config:add` command:

```
$ heroku config:add MANDRILL_USERNAME='Your_Username'
MANDRILL_API_KEY='Your_Mandrill_API_Key'
$ heroku config:add MAILCHIMP_LIST_ID='Your_List_ID'
$ heroku config:add ADMIN_NAME='First User' ADMIN_EMAIL='user@example.com'
ADMIN_PASSWORD='changeme'
$ heroku config:add 'ROLES=[admin, user]'
heroku config:add EMAIL_ADDRESS='me@example.com' DOMAIN='example.com'
```

Complete Heroku deployment with:

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

You've deployed the application.

Chapter 25

Comments

Credits

Daniel Kehoe implemented the application and wrote the tutorial.

Thank you to Kathy Onu for tutorial testing and proofreading.

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.

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