

SAMPLE CHAPTER

Rails 3

IN ACTION

Ryan Bigg
Yehuda Katz





Rails 3 in Action

Ryan Bigg
Yehuda Katz

Chapter 15

brief contents

- 1 ■ Ruby on Rails, the framework 1
- 2 ■ Testing saves your bacon 23
- 3 ■ Developing a real Rails application 44
- 4 ■ Oh CRUD! 83
- 5 ■ Nested resources 99
- 6 ■ Authentication and basic authorization 117
- 7 ■ Basic access control 136
- 8 ■ More authorization 164
- 9 ■ File uploading 213
- 10 ■ Tracking state 243
- 11 ■ Tagging 286
- 12 ■ Sending email 312
- 13 ■ Designing an API 347
- 14 ■ Deployment 385
- 15 ■ Alternative authentication 412
- 16 ■ Basic performance enhancements 434
- 17 ■ Engines 468
- 18 ■ Rack-based applications 516

15

Alternative authentication

This chapter covers

- Authenticating against external services using OmniAuth
- Authenticating with Twitter using OAuth
- Authenticating with GitHub using OAuth

Now that your application has been deployed to a server somewhere (or at least you've gone through the motions of doing that!), we're going to look at adding additional features to your application. One of these is OAuth authentication from services such as Twitter and GitHub.

When you sign into a website, you can generally use a couple of authentication methods. The first of these would be a username and password, with the username being forced to be unique. This method provides a solid way to identify what user has logged into the website, and from that identification the website can choose to grant or deny access to specific parts of the site. You have done this with your Ticketee application, except in place of a username, you're using an email address. An email address is an already unique value for users of a website that also allows you

to have a way of contacting the user if the need arises. On other websites, though, you may have to choose a username (with Twitter), or you could be able to use both a username and email to sign in, as with GitHub.

Entering your email address and a password¹ into every website that you use can be time consuming. Why should you be throwing your email addresses and passwords into every website?

Then along came OAuth. OAuth allows you to authenticate against an OAuth provider. Rather than giving your username/email and password to yet another site, you authenticate against a central provider, which then provides tokens for the different applications to read and/or write the user's data on the application.

In this chapter you're going to be using the OAuth process to let users sign in to your Ticketee application using Twitter and GitHub. You'll not only see how easy this is, but also how you can test to make sure that everything works correctly.

Rather than implementing this process yourself, you can use the OmniAuth gem in combination with the devise gem that you're already using. Although this combination abstracts a lot of the complexity involved with OAuth, it's still helpful to know how this process works. Let's take a look now.

15.1 How OAuth works

OAuth authentication works in a multi-step process. In order to be able to authenticate against other applications, you must first register your application with them. After this process is complete, you're given a unique key to identify your application and a secret passphrase, which is actually a hash. Neither of these should be shared. When your application makes a request to an OAuth provider, it will send these two parameters along as part of the request so the provider knows which application is connecting. Twitter's API documentation has a pretty good description of the process as an image, which you can see as figure 15.1.

First of all (not shown in the figure), a user initiates a request to your application (the Consumer) to announce their intentions to log in with Twitter (the Service Provider). Your application then sends that unique identifier and that secret key (given to you by Twitter when you register your application), and begins the authentication process by requesting a token (A). This token will be used as an identifier for this particular authentication request cycle.

The provider (Twitter) then grants you this token and sends it back to your application. Your application then redirects the user to the provider (B) in order to gain the user's permission for this application to access its data. When signing in with Twitter, your users would see something like figure 15.2.

The user can then choose to Sign In or Cancel on this screen. If they choose Sign In, the application then has access to their data, which authorizes the request token

¹ Ideally, a unique password per site is best for added security. If one site is breached, you do not want your password to be the same across multiple sites, because the attackers would gain access to everything.

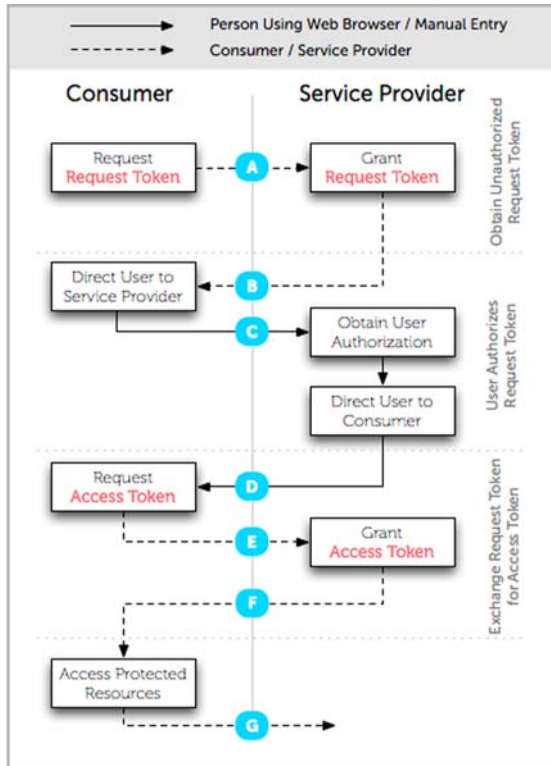


Figure 15.1 Twitter OAuth

You can use your Twitter account to sign in to other sites and services.
 By signing in here, you can use Ticketee without sharing your Twitter password.

Authorize Ticketee to use your account?


This application **will be able to:**

- Read Tweets from your timeline.
- See who you follow.

Sign In **Cancel**

This application **will not be able to:**

- Follow new people.
- Update your profile.
- Post Tweets for you.
- Access your private messages.
- See your Twitter password.



Ticketee
 By Fyhdua
 manning.com/katz
 The Rails 3 in Action book applicator
[← Cancel, and return to app](#)

Figure 15.2 Twitter authorization

you were given at the beginning. If they click Cancel, it redirects the user back to the application without giving it access to the data.

In this case, you'll assume the user has clicked Sign In. The user is then redirected back to your application from the provider, with two parameters: an `oauth_token` and a `oauth_verifier`. The `oauth_token` is the request token you were granted at the beginning, and the `oauth_verifier` is a verifier of that token. OmniAuth then uses these two pieces of information to gain an *access token*, which will allow your application to access this user's data. There's also additional data, such as the user's attributes, that gets sent back here. The provider determines the extent of this additional data.

This is just a basic overview of how the process works. All of this is covered in more extensive detail in Section 6 of the OAuth 1.0 spec, which can be found at <http://oauth.net/core/1.0/>.

In the case of your application, you're going to be letting users go through this process with the intention of using their authorization with Twitter to sign them in whenever they wish. After this process has been completed the first time, a user will not be re-prompted to authorize your application (unless they have removed it from their authorized applications list), meaning the authorization process will be seamless for the user.

Let's see how you can use the OmniAuth gem to set up authentication with Twitter in your application.

15.2 Twitter authentication

You're going to be using OmniAuth to let people sign in using Twitter and GitHub as OAuth providers. We'll begin with Twitter authentication and then move on to GitHub.

15.2.1 Setting up OmniAuth

OmniAuth not only supports OAuth providers, but also supports OpenID, CAS, and LDAP. You're only going to be using the OAuth part, which you can install in your application by putting this line in your Gemfile:

```
gem "oa-oauth", :require => "omniauth/oauth"
```

The different parts of OmniAuth are separated out into different gems by an `oa-` prefix so that you can use some parts without including all the code for the other parts. In your Gemfile you're loading the `oa-oauth` gem, which will provide the OAuth functionality you need. The file to load this gem does not have the same name as the gem, so you need to use the `:require` option here and tell the correct file, `omniauth/oauth`, to load.

Next, you need to tell Devise that your `User` model is going to be using OmniAuth. You can do this by putting the `:omniauthable` symbol at the end of the `devise` list in your `app/models/user.rb` so that it now becomes this:

```
devise :database_authenticatable, :registerable, :confirmable,  
      :recoverable, :rememberable, :trackable, :validatable,  
      :token_authenticatable, :omniauthable
```

With OmniAuth set up, you can now configure your application to provide a way for your users to sign in using Twitter. Twitter first requires you to register your application on its site.

15.2.2 *Registering an application with Twitter*

You need to register your application with Twitter before your users can use it to log in to your application. The registration process gives you a unique identifier and secret code for your application (called a consumer key and consumer secret, respectively), which is how Twitter will know what application is requesting a user's permission.

The process works by a user clicking a small Twitter icon on your application, which will then redirect them to Twitter. If they aren't signed in on Twitter, they will first need to do so. Once they are signed in, they will then be presented with the authorization confirmation screen that you saw earlier, shown again in figure 15.3.

On this screen you can see that Twitter knows what application is requesting permission for this user, and that the user can either choose to Allow or Deny. By clicking Allow, the user will be redirected back to your application and then signed in using code that you'll write after you've registered your application.

To register your application with Twitter, you need to go to <http://dev.twitter.com> and click the Create an App link.

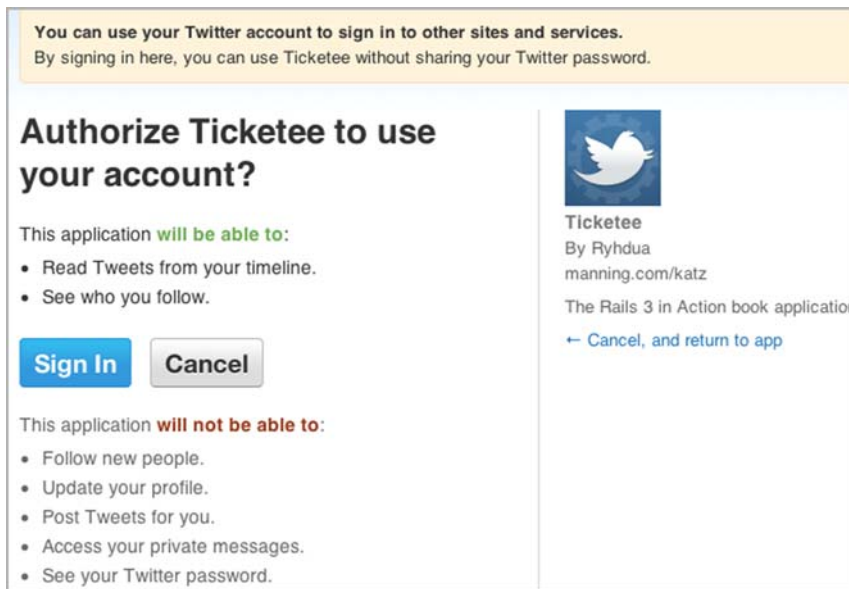


Figure 15.3 Twitter authorization request

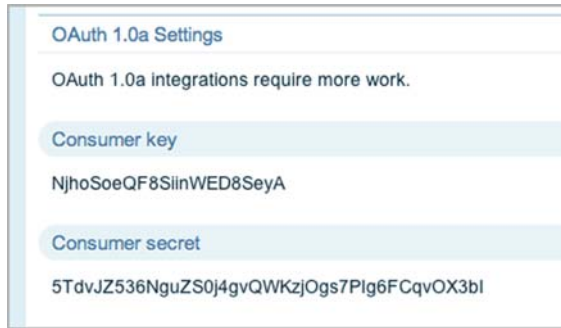


Figure 15.4 A brand-new application!

On this new page you need to fill in the name, description, and URL fields. The name should be [Your name]’s Ticketee because it needs to be unique; the description can be anything, and the URL can be `http://manning.com/katz`. When you click Create on this application, you’ll see the consumer key and secret that you’ll be using shortly, as shown in figure 15.4.

Although this screen isn’t exactly the prettiest thing around, it does offer you the two key pieces of information that you need: the consumer key and the consumer secret. The other values on this page aren’t important for you to know, as OmniAuth will take care of them for you.

You now need to set up your application to use this consumer key and consumer secret when authenticating with Twitter. You can do this in Devise’s configuration file in your application, which is located at `config/initializers/devise.rb`. In this file, you’ll see the following commented-out OmniAuth configuration:

```
# ==> OmniAuth
# Add a new OmniAuth provider. Check the wiki for more information on setting
# up on your models and hooks.
# config.omniauth :github, 'APP_ID', 'APP_SECRET', :scope =>
#   'user,public_repo'
```

This shows you how to add a new OmniAuth provider, using GitHub as an example. In this example, the `APP_ID` and `APP_SECRET` values would be the consumer key and consumer secret given to you by the provider. Set up a new provider for Twitter by putting these lines underneath the commented-out section:

```
config.omniauth :twitter,
  '[consumer key]',
  '[consumer secret]'
```

This will configure Devise to provide OmniAuth-based authentication for Twitter, but you’re not done yet. You need some way for a user to be able to initiate the sign-in process with Twitter.

15.2.3 Setting up an OmniAuth testing environment

To provide a user with a way to sign in with Twitter, you’ll add a small addition to your menu bar that lets people sign up and sign in using Twitter, as shown in figure 15.5.

When a user clicks this button, your application will begin the OAuth process by requesting a request token from Twitter, and then using that token to redirect to Twitter. From here, the user will authorize your application to have access to their data on Twitter, and then they'll be redirected back to your application. It's the user being redirected back to your application that is the most important part. Twitter will send back the `oauth_token` and `oauth_verifier`, and then your application makes the request for the access token to Twitter. Twitter will then send back this access token and any additional parameters it sees fit, and you'll be able to access this information in a Hash format. For example, Twitter sends back the user's information in the response like this:

```
{
  ...
  "extra" => {
    ...
    "user_hash" => {
      "id" => "14506011"
      "screen_name" => "ryanbigg"
      "name" => "Ryan Bigg",
      ...
    }
  }
}
```

This is quite a stripped-down version of the response you'll be getting back from Twitter, but it contains three very important values. The first is the unique Twitter-provided `id` of the user, the second is their Twitter username, and the third is their display name. Currently in Ticketee, you've been using the user's email to display who you're logged in as. Because Twitter doesn't send back an email address, you'll have to change where you'd usually display an email address to instead display the user's display name or screen name if they've chosen to sign in with Twitter.

First things first though: you need to have a link that a user can click to begin this process, and to make sure that the link is working you're going to need to write a feature. With this feature, you shouldn't always rely on being able to connect to your OAuth providers like Twitter. Instead, you should create fake responses (referred to as mocks) for the requests you'd normally do. By doing this you can substantially speed up the rate at which your tests run, as well as not depend on something like connectivity, which is out of your control.

OmniAuth provides a configuration option for setting whether or not you're in a test mode, which will mock a response rather than making a call to an external service. This option is conveniently called `test_mode`. You can set this option at the bottom of your `config/environments/test.rb` like this:

```
OmniAuth.config.test_mode = true
```

With your test environment now set up correctly, you can write a feature to make sure that users can sign in with Twitter.



Figure 15.5
Sign in with Twitter

15.2.4 Testing Twitter sign-in

Next, you can begin to write your feature to test Twitter authentication in a new file at `features/twitter_auth.feature` as shown in the following listing.

Listing 15.1 `features/twitter_auth.feature`

```
Feature: Twitter auth
  In order to sign in using Twitter
  As a Twitter user
  I want to click an icon and be signed in

Background:
  Given we are mocking a successful Twitter response

Scenario: Signing in with Twitter
  Given I am on the homepage
  When I follow "sign_in_with_twitter"
  Then I should see "Signed in with Twitter successfully."
  And I should see "Signed in as A Twit (@twit)"
```

This is a simple little feature with a short, three-line scenario. The step in your Background will mock out a successful response from Twitter, which will be used by OmniAuth because you declared that you're in test mode in `config/environments/test.rb`. Let's run this feature now with `bin/cucumber features/twitter_auth.feature` so that you can get the step definition for this new step:

```
Given /^I have mocked a successful Twitter response$/ do
  pending # express the regexp above with the code you wish you had
end
```

Put this step definition in a new file at `features/step_definitions/oauth_steps.rb` and define it as shown in the following listing.

Listing 15.2 `features/step_definitions/oauth_steps.rb`

```
Given /^we are mocking a successful Twitter response$/ do
  OmniAuth.config.mock_auth[:twitter] = {
    "extra" => {
      "user_hash" => {
        "id" => '12345',
        "screen_name" => 'twit',
        "display_name" => "A Twit"
      }
    }
  }
end
```

To generate a fake response for OmniAuth, you need to use `set up a key` `OmniAuth.config.mock_auth` hash that has the same name as the authentication provider, which in this case is `:twitter`. This mock response needs to contain the same kind of layout as the normal response would get back, including having each of the keys of the hashes be strings, because this is how your response will be accessed. Twitter's response hash, as stated earlier, contains an `extra` key that contains information about a user, which is

what you'll use to track who has signed into your system using Twitter. You'll store these three attributes in new fields in your database when a user signs up using Twitter.

Run `bin/cucumber features/twitter.feature` again. This time you'll see that you're missing your link:

```
Scenario: Signing in with Twitter
  Given I am on the homepage
  And I follow "sign_in_with_twitter"
    no link with title, id or text 'sign_in_with_twitter' found ...
```

Rather than have a link that reads `sign_in_with_twitter`, you'll actually be giving the link an `id` attribute of `sign_in_with_twitter` and Capybara will still be able to find this link. The link itself is going to be a small button that you can get from <https://github.com/intridea/authbuttons>. You should download these images (just the 32 x 32px versions) and put them in the `app/assets/images/icons` directory of your application. Leave them named as they are.

To create this new link, open `app/views/layouts/application.html.erb`. This file contains the layout for your application and is responsible for displaying the Sign Up and Sign In links for your application if the user isn't signed in already. It's underneath these links that you want to display your little twitter icon, which you can do by making this small change to this file:

```
<%= link_to "Sign up", new_user_registration_path %>
<%= link_to "Sign in", new_user_session_path %>
<br>
Or use <%= link_to image_tag("icons/twitter_32.png"),
                    user_omniauth_authorize_path(:twitter),
                    :id => "sign_in_with_twitter" %>
```

With this link you use the downloaded icon as the first argument of `link_to` by using `image_tag`. The second argument to `link_to` is the routing helper method `user_omniauth_authorize_path` with the `:twitter` argument. This method is provided by Devise because you've told it your `User` model is `omniauthable`. This routing helper will go to a controller that is internal to Devise, because it will deal with the hand-off to Twitter.

When you run this feature again, the second step of your scenario will still fail, but this time with a different error:

```
And I follow "sign_in_with_twitter"
  The action 'twitter' could not be found
  for Devise::OmniauthCallbacksController
```

By default, Devise handles the callbacks from external services using the `Devise::OmniAuthCallbacksController`. Because different people will want this controller to perform differently, Devise provides a set of common functionality in this controller and expects you to subclass it to define the actions (like your `twitter` action) yourself. To do this, create a new controller for these callbacks by running this command:

```
rails g controller users/omniauth_callbacks
```

This command will generate a new controller at `app/controllers/users/omniauth_callbacks_controller.rb`, but it's not quite what you want. You want this controller to inherit from `Devise::OmniauthCallbacksController`, and you also want it to have a `twitter` action. Before you do that, though, tell Devise to use this new controller for its callbacks. You can do this by changing these lines in your `config/routes.rb` file

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

into this:

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

This will tell Devise to use your newly generated `users/omniauth_callbacks_controller` rather than its own `Devise::OmniauthCallbacksController`, which you'll use as the superclass of your new controller. This `Devise::OmniauthCallbacksController` contains some code that will be used in case something goes wrong with the authentication process.

Now you need to define the `twitter` action in this new controller. This action is going to be called when Twitter sends a user back from having authorized your application to have access. Define this controller using the code from the following listing.

Listing 15.3 `app/controllers/users/omniauth_callbacks_controller.rb`

```
class Users::OmniauthCallbacksController <
  Devise::OmniauthCallbacksController
  def twitter
    @user = User.find_or_create_for_twitter(env["omniauth.auth"])
    flash[:notice] = "Signed in with Twitter successfully."
    sign_in_and_redirect @user, :event => :authentication
  end
end
```

When a request is made to this action, the details for the user are accessible in the `env["omniauth.auth"]` key, with `env` being the Rack environment of this request, which contains other helpful things such as the path of the request.²

You then pass these details to a currently undefined method called `find_or_create_for_twitter`, which will deal with finding a `User` record for this information from Twitter, or creating one if it doesn't already exist. You then set a `flash[:notice]` telling the user they've signed in and use the Devise-provided `sign_in_and_redirect` method to redirect your user to the `root_path` of your application, which will show the `ProjectsController`'s `index` action.

² Covered in much more detail in chapter 17.

To make this action work, you're going to need to define `find_or_create_for_twitter` in your `User` model, which you can do using the code from the following listing.

Listing 15.4 `app/models/user.rb`

```
def self.find_or_create_for_twitter(response)
  data = response['extra']['user_hash']
  if user = User.find_by_twitter_id(data["id"])
    user
  else # Create a user with a stub password.
    user = User.new(:email => "twitter+#{data["id"]}@example.com",
                  :password => Devise.friendly_token(0,20))
    user.twitter_id = data["id"]
    user.twitter_screen_name = data["screen_name"]
    user.twitter_display_name = data["display_name"]
    user.confirm!
    user
  end
end
```

You've defined this class method to take one argument, which is the response you get back from Twitter. In this response, there's going to be the access token that you get back from Twitter that you don't care so much about, and also the `extra` key and its value that you do really care about. It's with these that the application then attempts to find a user based on the `id` key **1** within the `response["extra"]["user_hash"]` (here as `data` to make it easier to type). If it can find this user, it'll return that object.

If it can't find a user with that `twitter_id` attribute, then you need to create one! Because Twitter doesn't pass back an email, you make one up **2**, as well as a password, **3** using Devise's very helpful `friendly_token` method, which generates a secure phrase like `QfVRz8RxHx4Xkqe6uIqL`. The user won't be using these to sign in; Devise needs them so it can validate the user record successfully.

You have to do this the long way, because the `twitter_`-prefixed parameters aren't mass-assignable due to your `attr_accessible` call earlier on in this model, so you must assign them manually one at a time. Store the `id` of the user so you can find it again if you need to re-authenticate this user, the `twitter_screen_name`, and the `twitter_display_name`. Then you need to confirm and save the object, which you can do with the `confirm!` method, and finally you need to return the object as the final line in this `else` block.

These fields are not yet fields in your database, so you'll need to add them in. You can do this by creating a new migration using this command:

```
rails g migration add_twitter_fields_to_users
```

In this migration you want to add the fields to your table, which you can do by adding them to your migration, as shown in the following listing.

Listing 15.5 db/migrate/[timestamp]_add_twitter_fields_to_users.rb

```
class AddTwitterFieldsToUsers < ActiveRecord::Migration
  def change
    add_column :users, :twitter_id, :string
    add_column :users, :twitter_screen_name, :string
    add_column :users, :twitter_display_name, :string
  end
end
```

With this migration set up, you can run it on your development and test databases with `rake db:migrate` and `rake db:test:prepare` respectively. Now when you run your feature again with `bin/cucumber features/twitter_auth.feature`, you'll see that your new `User` object is being created and that you can see the "Signed in with Twitter successfully." message:

```
Scenario: Signing in with Twitter
  Given I am on the homepage
  And I follow "sign_in_with_twitter"
  Then I should see "Signed in with Twitter successfully."
  Then I should see "Signed in as A tweet"
    Failed assertion, no message given. (MiniTest::Assertion)
```

The final step of your feature is now failing, but this is a pretty easy one to fix. You need to change where it would normally display a user's email to display something like "A Tweet (@tweet)" if the `twitter_id` attribute is set. To do this, define a new method in your `User` model above the `to_s` method, using the code from the following listing.

Listing 15.6 app/models/user.rb

```
def display_name
  if twitter_id
    "#{twitter_display_name} (@#{twitter_screen_name})"
  else
    email
  end
end
```

If the `twitter_id` attribute is set in this method, then you assume the `twitter_display_name` and `twitter_screen_name` attributes are set also and use those to display the twitter name. If it isn't set, then you'll fall back to using the `email` field. You'll be able to use this method later on to check if the `github_id` field is set and use the values for that instead.³

Now you need to change the occurrences of where `user.email` is referenced to use the `display_name` method. The first occurrence of this is in `app/models/user.rb` in your `to_s` method, which should now become

³ Alternatively, you could add a feature to let the user pick which one they would like to display.

```
def to_s
  "#{display_name} (#{admin? ? "Admin" : "User"})"
end
```

The rest of the occurrences are found in a handful of views throughout your application, and you'll need to fix these up now. The first of these is the first line of `app/views/admin/permissions/index.html.erb`, which should now become this:

```
<h2>Permissions for <%= @user.display_name %></h2>
```

Next, there's one in the application layout at `app/views/layouts/application.html.erb`:

```
Signed in as <%= current_user.email %>
```

This needs to become simply

```
Signed in as <%= current_user %>
```

By placing an object like this in the view, the `to_s` method will be called on it automatically, which is of course the `to_s` method in the `User` model.

Finally, you'll need to update the `app/views/tickets/show.html.erb` page in the same manner, changing this

```
<%= @ticket.user.email %>
```

to this:

```
<%= @ticket.user.display_name %>
```

That's it! That's all the occurrences of calls to the `email` attribute in places where it's shown to users has been changed to `display_name` instead. So does this mean that your feature will now run? Find out with a quick run of `bin/cucumber features/twitter_auth.feature`:

```
1 scenario (1 passed)
5 steps (5 passed)
```

All green, all good. Now users are able to sign up and sign in by clicking the Twitter icon in your application rather than providing you with their email and password. The first time a user clicks this icon, they'll be redirected off to Twitter, which will ask them to authorize your application to access their data. If they choose Allow, they will be redirected back to your application. With the parameters sent back from the final request, you'll attempt to find a `User` record matching their Twitter ID or, if there isn't one, create one instead. Then you'll sign them in.

After that, when the user attempts to sign in using the Twitter icon, they'll still be redirected back to Twitter, but this time Twitter won't ask them for authorization again. Instead, Twitter will instantly redirect them back to your application; the whole process will seem pretty smooth, albeit with the delay that can normally be expected from doing two HTTP requests.

Go ahead, try launching `rails server` now and accessing the application at `http://localhost:3000` by clicking the small Twitter icon on the sign-in page. You'll be

redirected off to Twitter, which deals with the authentication process before sending you back to the application.

Did you break anything? Let's see by running `rake cucumber:ok spec`:

```
63 scenarios (63 passed)
737 steps (737 passed)
# and
72 examples, 0 failures, 19 pending
```

Nope, it seems like everything is functioning correctly. Let's make a commit:

```
git add .
git commit -m "Added OmniAuth-driven support for signing in with Twitter"
```

With the work you've done in this section, users will now be able to easily sign into your application using Twitter. You can see this for yourself by starting a server using `rails s` and clicking the Twitter icon if you've got a Twitter account.

If your users don't have a Twitter account, then their only other choice at the moment is to provide you with their email address and a password, and that's not really useful to anyone who has a GitHub but not a Twitter account. So let's see how you can authenticate people using GitHub's OAuth next, while recycling some of the Twitter-centric code in the process.

15.3 GitHub authentication

We've shown how you can let people authenticate using Twitter's OAuth. GitHub also provides this service, and the OmniAuth gem you're using can be used to connect to that too, in much the same way as you did with Twitter. Rather than re-doing everything that you did in the previous section again and changing occurrences of "twitter" to "github," you'll be seeing how you can make the code that you've written so far support both Twitter and GitHub in a clean fashion. When you're done, you're going to have a little GitHub icon next to your Twitter one so that people can use GitHub, Twitter, or email to sign in, making your sign in/sign up area look like figure 15.6.

As was the case with Twitter, your first step will be registering an application with GitHub.



Figure 15.6 GitHub login

15.3.1 Registering and testing GitHub auth

To register an application with GitHub, you must first be signed in. Then you can visit <https://github.com/account/applications/new> and fill in the form that it provides. After that, you'll need to copy the Client ID and Client Secret values and put them in your `config/initializers/devise.rb` file under your Twitter details, like this:

```
config.omniauth :github, "[Client ID]", "[Client Secret]"
```

With GitHub now set up in your application, you can write the feature to ensure that its authentication is working. To begin testing your application's ability to

authenticate users from GitHub, you're going to write a new feature at `features/github_auth.feature` and fill it with the content from the following listing.

Listing 15.7 `features/github_auth.feature`

```
Feature: GitHub auth
  In order to sign in using GitHub
  As a GitHub user
  I want to click an icon and be signed in

Background:
  Given I have mocked a successful GitHub response

Scenario: Signing in with GitHub
  Given I am on the homepage
  And I follow "sign_in_with_github"
  Then I should see "Signed in with Github successfully."
  Then I should see "Signed in as A GitHubber"
```

Although it may look like all you've done here is replace all the references to Twitter with GitHub... actually, that's precisely what you've done! This is because there should be little difference in how the user interacts with your site to sign in with Twitter or GitHub. The differences should only be behind the scenes, as this is how a user would expect an application to behave.⁴

When you run this new feature with `bin/cucumber features/github_auth.feature`, you'll see that you've got an undefined step:

```
Given /^I have mocked a successful GitHub response$/ do
  pending # express the regexp above with the code you wish you had
end
```

Define this step in `features/step_definitions/oauth_steps.rb` underneath the one for Twitter. It goes like this:

```
Given /^I have mocked a successful GitHub response$/ do
  OmniAuth.config.mock_auth[:github] = {
    "extra" => {
      "user_hash" => {
        "id" => '12345',
        "email" => 'githubber@example.com',
        "login" => "githubber",
        "name" => "A GitHubber"
      }
    }
  }
end
```

GitHub returns a similar hash to that of Twitter, containing an `extra` key with a `user_hash` key nested inside. Within this nested hash you've got the three parameters that you'll be storing on your end: the id, the login, and a name.

⁴ Also known as Principle of least surprise (POLS) or more colloquially, "keep it simple, stupid!" (KISS).

When you run your feature again, you'll be through this undefined step and now up to the next failing step:

```
And I follow "sign_in_with_github"
  no link with title, id or text 'sign_in_with_github' found
```

This means that your `sign_in_with_github` link doesn't exist yet, so you're going to need to create it like you did with your `sign_in_with_twitter` link. You could do this by copying and pasting the Twitter link code underneath itself in `app/views/layouts/application.html.erb`, ending up with something like this:

```
Or use <%= link_to image_tag("icons/twitter_32.png"),
                  user_omniauth_authorize_path(:twitter),
                  :id => "sign_in_with_twitter" %>
<%= link_to image_tag("icons/github_32.png"),
            user_omniauth_authorize_path(:github),
            :id => "sign_in_with_github" %>
```

This code in your application layout is going to get ugly as you add providers, and it's quite a lot of duplication! What would be more sensible is moving this code into a helper method in a new file such as `app/helpers/oauth_helper.rb`, defining it as shown in the following listing.

Listing 15.8 `app/helpers/oauth_helper.rb`

```
module OAuthHelper
  def auth_provider(name)
    link_to image_tag("icons/#{name}_32.png"),
            user_omniauth_authorize_path(name),
            :id => "sign_in_with_#{name}"
  end
end
```

Then in place of the ugly code in your application layout, you'd put this instead:

```
Or use <%= auth_provider(:twitter) %> <%= auth_provider(:github) %>
```

How's that for simplicity? Well, you could make it even cleaner by accepting any number of arguments to your method, by turning it into this:

```
def auth_providers(*names)
  names.each do |name|
    concat(link_to(image_tag("icons/#{name}_32.png"),
                    user_omniauth_authorize_path(name),
                    :id => "sign_in_with_#{name}"))
  end
  nil
end
```

This helper uses the `concat` method to output the links to your view. If you didn't use this, it wouldn't render them at all. You could then write this in your application layout:

```
Or use <%= auth_providers(:twitter, :github) %>
```

Now isn't that way nicer? If at any time you want to add or remove one of the links, you only have to add or remove arguments to this method.

When you run this feature again with `bin/cucumber features/github_auth.feature`, you'll see that you're on to the next error:

```
The action 'github' could not be found for Users::OmniauthCallbacksController
```

As you did with Twitter, you're going to need to define a github action in the `Users::OmniauthCallbacksController`. This action will find or create a user based on the details sent back from GitHub, using a class method you'll define after in your User model. Sound familiar? You can duplicate the twitter action in this controller and create a new github action from it like this:

```
def github
  @user = User.find_or_create_for_github(env["omniauth.auth"])
  flash[:notice] = "Signed in with GitHub successfully."
  sign_in_and_redirect @user, :event => :authentication
end
```

But like the provider links in your application layout, this is not very clean and gets exceptionally more complex the more providers you have. Rather than doing it this way, you'll define a class method for your controller that will dynamically define these methods for you. Define this method in `app/controllers/users/omniauth_callbacks_controller.rb` by using the code from the following listing.

Listing 15.9 `app/controllers/users/omniauth_callbacks_controller.rb`

```
def self.provides_callback_for(*providers)
  providers.each do |provider|
    class_eval %Q{
      def #{provider}
        @user = User.find_or_create_for_#{provider}(env["omniauth.auth"])
        flash[:notice] = "Signed in with #{provider.to_s.titleize}
        ↪successfully."
        sign_in_and_redirect @user, :event => :authentication
      end
    }
  end
end
```

← 1 Evaluate code

As with your `auth_providers` method in `OauthHelper`, you can call this method in your controller (after removing the twitter and github methods already in it):

```
provides_callback_for :twitter, :github
```

The `provides_callback_for` method will iterate through each of the arguments passed in, defining a new method dynamically using `class_eval` 1, which will evaluate the code you pass in within the context of the current class. The `%Q{}` encapsulation will provide a `String` object that you can put double quotes and single quotes in without having to escape them.

You then need to replace any occurrences that you previously had of either “twitter” or “github” with the provider variable from the current iteration, using interpolation to put it into the quoted string. The `provides_callback_for` method will then define a new action in your controller for the specified providers. This has greatly decreased the repetition in your controller’s code, at the expense of a small easy-to-understand bit of `class_eval` “magic.”

When you run your feature again with `bin/cucumber features/github.feature`, you’ll see that it’s now hitting your new `github` action, because it can’t find a method that you use in it:

```
undefined method `find_or_create_for_github' for ...
(eval):3:in `github'
```

In this error output you’re seeing that Rails is unable to find a `find_or_create_for_github` method on a class, which is the `User` class. You created one of these for Twitter, and unlike the provider links and the callback actions, you’re not able to easily create a bit of smart code for your model. But you can separate out the concerns of the model into separate files, which would make it easier to manage. Rather than filling your `User` model with methods for each of your providers, you’ll separate this code out into another module and then extend your class with it.

You can do this by creating a new directory at `app/models/user` and placing a file called `app/models/user/omniauth_callbacks.rb` inside it. You should put the content from the following listing inside this file.

Listing 15.10 `app/models/user/omniauth_callbacks.rb`

```
class User < ActiveRecord::Base
  module OmniauthCallbacks
    def find_or_create_for_twitter(response)
      data = response['extra']['user_hash']
      if user = User.find_by_twitter_id(data["id"])
        user
      else # Create a user with a stub password.
        user = User.new(:email => "twitter+#{data["id"]}@example.com",
                       :password => Devise.friendly_token[0,20])
        user.twitter_id = data["id"]
        user.twitter_screen_name = data["screen_name"]
        user.twitter_display_name = data["display_name"]
        user.confirm!
        user
      end
    end
  end
end
```

In this file you define an `OmniauthCallbacks` module inside your `User` class. Inside this module, you’ve put the `find_or_create_for_twitter` method straight from your `User` model, except you’ve removed the `self` prefix to the method name. You can

now go ahead and remove this method from the `User` model, making it temporarily unavailable.

By separating out the concerns of your model into separate modules, you can decrease the size of the individual model file and compartmentalize the different concerns of a model when it becomes complicated, like your `User` model has.

To make this method once again available, you need to extend your model with this module. You can do this by making the first two lines of your model into

```
class User < ActiveRecord::Base
  extend OmniauthCallbacks
```

The `extend` method here will make the methods available for the module on the class itself as class methods.

TIP It's generally a good idea to put any `extend` or `include` calls at the beginning of a class definition so that anybody else reading it will know if the class has been modified in any way. If an `extend` is buried deep within a model, then it can be difficult to track down where its methods are coming from.

By adopting a convention of putting things that can potentially seriously modify your class at the top of the class definition, you're giving a clear signal to anyone (including your future self who may have forgotten this code upon revisiting) that there's more code for this model in other places.

You can now define your `find_or_create_by_github` method in the `User::OmniauthCallbacks` module by using the code from the following listing.

Listing 15.11 `app/models/user/omniauth_callbacks.rb`

```
def find_or_create_for_github(response)
  data = response['extra']['user_hash']
  if user = User.find_by_github_id(data["id"])
    user
  else # Create a user with a stub password.
    user = User.new(:email => data["email"],
                   :password => Devise.friendly_token[0,20])
    user.github_id = data["id"]
    user.github_user_name = data["login"]
    user.github_display_name = data["name"]
    user.confirm!
    user
  end
end
```

← 1 Create user

You're lucky this time around, as the form of the data you get back from GitHub isn't too different to Twitter, coming back in the `response['extra']['user_hash']` key. In the case of other providers, you may not be so lucky. The form of the data sent back is not standardized, and so providers will choose however they like to send back the data.

Included in the data you get back from GitHub is the user's email address, which you can use ❶ to create the new user, unlike with the `find_or_create_for_twitter` method where you had to generate a fake email. The added bonus of this is that if a user wishes to sign in using either GitHub or their email, they would be able to do so after resetting their password.

The final lines of this method should be familiar; you're setting the `github_id`, `github_user_name` and `github_display_name` fields to store some of the important data sent back from GitHub. You're able to re-sign-in people who visit a second time from GitHub based on the `github_id` field you save. Finally, you confirm the user so that you're able to sign in as them.

With the `find_or_create_for_github` method defined, has your feature progressed? Find out with a run of `bin/cucumber features/github_auth.feature`:

```
And I follow "sign_in_with_github"
  undefined method `find_by_github_id' for ...
```

Ah, it would appear that you're not quite done! You need to define the `github` fields in your `users` table so that your newly added method can reference them. Go ahead and create a migration to do this now by running this command:

```
rails g migration add_github_fields_to_users
```

You can then alter this migration to add the fields you need by using the code from the following listing.

Listing 15.12 db/migrate/[timestamp]_add_github_fields_to_users.rb

```
class AddGithubFieldsToUsers < ActiveRecord::Migration
  def change
    add_column :users, :github_id, :integer
    add_column :users, :github_user_name, :string
    add_column :users, :github_display_name, :string
  end
end
```

Alright, you can now run this migration using `rake db:migrate` and `rake db:test:prepare` to add these fields to your `users` table. Now you can run your feature again with `bin/cucumber features/github_auth.feature` to see this output:

```
Scenario: Signing in with GitHub
  Given I am on the homepage
  And I follow "sign_in_with_github"
  Then I should see "Signed in with Github successfully."
  Then I should see "Signed in as A GitHubber (githubber)"
    expected there to be content "Signed in as A Githubber"
```

The third step of your scenario is now passing, but the fourth is failing because you're not displaying the GitHub-provided name as the "Sign in as ..." line in your application. You can easily rectify this by changing the `display_name` method in `app/`

models/user.rb to detect if the `github_id` field is set like it does already with the `twitter_id` field.

Underneath the display name output for the `if twitter_id` case in `app/models/user.rb`, add these two lines:

```
elsif github_id
  "#{github_display_name} (#{github_user_name}) "
```

The entire method is transformed into this:

```
def display_name
  if twitter_id
    "#{twitter_display_name} (@#{twitter_screen_name}) "
  elsif github_id
    "#{github_display_name} (#{github_user_name}) "
  else
    email
  end
end
```

When you run `bin/cucumber features/github_auth.feature` again, you should see that it's all passing:

```
1 scenario (1 passed)
5 steps (5 passed)
```

Now users are able to use GitHub to sign in to your site, as well as Twitter or their email address if they please. Make a commit for the changes that you've done, but first make sure everything's running with a quick run of `rake cucumber:ok spec`:

```
64 scenarios (64 passed)
746 steps (746 passed)
# and
56 examples, 0 failures
```

All systems green! Time to commit:

```
git add .
git commit -m "Add GitHub authentication support"
git push
```

You've seen how you can support another authentication provider, GitHub, along with supporting Twitter and email-based authentication too. To add another provider you'd only need to follow these six easy steps:

- 1 Create a new client on the provider's website, which differs from provider to provider.
- 2 Add the new client's information to `config/initializers/devise.rb` as a new provider.
- 3 Write a test for your new provider to make sure that people can always use it to sign in.

- 4 Add the provider icon to your listed providers in `app/views/layouts/application.html.erb` by passing another argument to the `auth_providers` helper method that you defined in `OauthHelper`.
- 5 Add a callback to the `Users::OmniauthCallbacksController` by using the `provides` method. Again, passing another argument to this method is all you need.
- 6 Define the `find_or_create_for_[provider]` method in the `User::OmniauthCallbacks` module.

Due to the flexibility offered by Devise and OmniAuth, there's no provider-specific configuration you need to do: it all works beautifully. For a full list of providers, check out the `omniauth` project on GitHub: <https://github.com/intridea/omniauth>.

See for yourself if GitHub's authentication is working by launching `rails server` again and going to `http://localhost:3000` and clicking the GitHub icon.

15.4 Summary

In this chapter you've seen how easy it is to implement authentication using two OAuth providers: Twitter and GitHub. You did this using the OmniAuth integration, which is available in Devise versions after 1.2.

For the Twitter section, you implemented the complete flow in a very simple manner using the features given to you by Devise, such as the routing helper, which initially sends a request off to the provider. Before OmniAuth came along, this process was incredibly tedious. It's truly amazing what OmniAuth offers you in terms of integrating with these providers.

When you got to the GitHub section, rather than copying and pasting the code you created for Twitter, you saw how you could reduce repetition in your code by using methods that iterate through a list of providers to display the icons or to provide callbacks.

Now that you've got multiple ways to allow people to sign in to your application, the barrier of entry is lowered because people can choose to sign in with a single click (after they've authorized the application on the relevant provider), rather than filling in the sign-in form each time. You've also got a great framework in place if you want to add any more providers.

Your application is at a pretty good state now, but you've not yet made sure that it can perform as efficiently as possible. If thousands of users flock to your application, how can you code it in such a way as to reduce the impact on your servers? In the next chapter, we look at how you can implement some basic performance enhancements to make your application serve requests faster, or even create a way by which a request skips the application altogether.

Rails 3 IN ACTION

Ryan Bigg • Yehuda Katz



Rails 3 is a full stack, open source web framework powered by Ruby and this book is an introduction to it. Whether you're just starting or you have a few cycles under your belt, you'll appreciate the book's guru's-eye-view of idiomatic Rails programming.

You'll master Rails 3.1 by developing a ticket tracking application that includes RESTful routing, authentication and authorization, state maintenance, file uploads, email, and more. You'll also explore powerful features like designing your own APIs and building a Rails engine. You will see Test Driven Development and Behavior Driven Development in action throughout the book, just like you would in a top Rails shop.

What's Inside

- Covers Rails 3.1 from the ground up
- Testing and BDD using RSpec and Cucumber
- Working with Rack

It is helpful for readers to have a background in Ruby, but no prior Rails experience is needed.

Ryan Bigg is a Rails developer in Sydney, recognized for his prolific and accurate answers on IRC and StackOverflow.

Yehuda Katz is a lead developer on SproutCore, known for his contributions to Rails 3, jQuery, Bundler, and Merb.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/Rails3inAction

“Takes you on an excellent Rails 3 adventure!”

—Anthony J. Topper
Penn State Harrisburg

“Conversational and current. A wellspring of information.”

—Jason Rogers, Dell Inc.

“An essential roadmap for the newest features in Rails 3.”

—Greg Vaughn
Improving Enterprises

“Essential, effective Rails techniques and habits for the modern Rubyist.”

—Thomas Athanas
Athanas Empire, Inc.

“A holistic book for a holistic framework.”

—Josh Cronmeyer
ThoughtWorks Studios

ISBN 13: 978-1-935182-27-6
ISBN 10: 1-935182-27-7



9 781935 182276