CHAPTER 10

Adding Tags to the Photo Gallery

In this chapter, we will extend the photo gallery to support tagging. Tags are simply keywords that are used to describe a particular object. Tagging has become hugely popular for social web applications and is a very useful way of categorizing items that makes it very easy for users to search and browse objects. Some of the most successful web sites that use tagging are Flickr.com, del.icio.us, and Amazon.com.

Along with being an incredibly useful way to allow searching of objects on the site, they also make for a fun browsing experience. Tag clouds are often shown, displaying the most popular tags as the largest, so you can quickly see the most popular topics. Tag clouds make a great starting point to allow people to discover new photos on your site.

To implement the tagging functionality, we are going to make use of a Rails plug-in. We will also use a Rails feature called Remote JavaScript (RJS) templates to implement Ajax effects, allowing the tag list to be dynamically updated on the photo edit page.

The Gallery Tagging Requirements

We will add tags to the RailsCoders photo galleries. Each photo on the site can be assigned a number of tags by the owner of the photo. Users can then browse the photo galleries using tag clouds.

Users of the site can view a tag cloud for all of the photos on the site or for one specific user. Clicking a tag within the tag cloud will show a paginated series of photos for that particular tag, again either showing all photos on the site with that tag or just those belonging to a specific user.

We will also need to add a way of adding tags to a photo. Since this can be performed only by the owner of a photo, we can simply add this to the photo edit page. We could implement this using just a standard text entry box, but to make it easier for the user to add and delete tags, we will implement it using an Ajax-based interface. This means that we will be able to add and delete tags from a tag list without having to reload the entire page; we can just update the specific part of the current page that has changed.

AJAX

Ajax (Asynchronous JavaScript and XML) is a technique that allows web applications to change only the relevant part of a web page, rather than forcing the browser to reload the entire page. This can greatly improve the usability of a web application if used properly. It also makes the application feel more responsive, since only a small amount of data is exchanged between the browser and web server.

The term Ajax was coined by Jesse James Garrett in 2005, but the technology has been around since 1998 when Microsoft released Internet Explorer 5, which included a technology called XMLHttpRequest. This allowed JavaScript on a web page to talk to the server in the background without having to reload a web page.

You could write your own JavaScript to make this happen, but Rails ships with the JavaScript libraries Prototype and script.aculo.us, which make it incredibly simple to implement Ajax features in your application in just a few lines of code.

Basically, a JavaScript action is tied to an event on your web page, such as a link or a button. When this event is triggered by the user, the JavaScript code is executed, which, in turn, sends a request to your application server. However, it does this asynchronously, allowing the user to continue using the page. The Rails application server sends back a snippet of JavaScript code, which updates a specific part of the current web page. The `id` and `class` tags in your XHTML code allow you to reference specific items or sections of the page.

To allow two ways to view the tag data—one for all users and one for a specific user—we will create two controllers.

To view tags by all users, the controller will be accessed by URLs such as `/tags` and `/tags/puppy`. We will create this to act as a normal REST resource, but we will not create a corresponding Tag model as we have with the other resources we have built; the retrieved tags are from the Photo model.

To view tags that only belong to one user, the controller will be accessed by URLs such as `/users/1/tags` and `/users/1/tags/puppy`. This will also act as a REST resource, but this time, it's nested beneath the users resource. We will create a second controller for this. We could handle both types of request with one controller, but we would require conditional statements to retrieve the correct data and render different views depending on the request. That would just complicate our code and make it harder for us to maintain and extend in the future.

We also need to add a way for a user to add and remove tags from a photo. Since tags are only accessible through the Photo model, this should be performed as an action of the photo controller. Therefore, we will add two methods to the existing `user_photo_controller` file: `add_tag` and `remove_tag`.

Tagging with Rails

Implementing our own tagging system from scratch would require some pretty complex SQL queries and take a significant amount of time to build and test properly. However, the Rails community has built a number of Rails extensions implementing tagging functionality. Which library you decide to use depends on your needs, since each library has different advantages.

The currently available libraries follow:

- acts_as_taggable *gem*: This was the first tagging library for Rails but is starting to show its age, and it is currently only available for Rails 1.1. However, it may still be useful if you are working with older Rails code. You can find out more at http://rubyforge.org/projects/taggable.
- acts_as_taggable *plug-in*: This plug-in was developed by David Heinemeier Hansson, the original creator of Rails. It was developed as a demonstration of the Rails has_many :through feature rather than a library intended for production use. Therefore, it is not fully featured and is not in active development. However, many people have modified it and are using it, even though the plug-in itself has not been updated. You can find out more information at http://wiki.rubyonrails.com/rails/pages/ActsAsTaggablePluginHowto.
- acts_as_taggable_on_steroids *plug-in*: This library is based on the acts_as_taggable plug-in but has been extended by Jonathan Viney to add tests, better tag assignment, and a feature to automatically perform tag cloud calculations. You can find more information about this plug-in at http://www.agilewebdevelopment.com/plugins/acts_as_taggable_on_steroids.
- has_many_polymorphs *ActiveRecord plug-in*: This is not a straightforward tagging plug-in; it basically allows you to define self-referential polymorphic associations in your models. It was developed by Evan Weaver and can be easily adapted to provide tagging facilities. You can find more information on how to use it to develop tagging features at http://blog.evanweaver.com/articles/2006/06/02/has_many_polymorphs.

For the RailsCoders site, we are going to use the acts_as_taggable_on_steroids plug-in. Currently, this is the most fully featured and easiest to use tagging library, providing us with a very simple way to add and edit tags and perform searches based on tags. It can also provide us with tag counts so that we can easily produce tag clouds.

The acts_as_taggable_on_steroids Plug-in

The acts_as_taggable_on_steroids plug-in, though based on the original acts_as_taggable plug-in, has been extended with improved tag assignment methods and tag cloud calculations. This makes it ideal for using on RailsCoders.

The plug-in also comes with a number of tests, meaning that you can easily test the plug-in's functionality to ensure it is working as you expect.

The library uses a Rails feature that allows you to relate two models through a third model. You do this by using the has_many statement and specifying a join model. For instance, with our tagging system, we will join the Photo model and the Tag model together using a join model called Tagging. We would specify that the photo has many tags, but to access these, the request must go through the Tagging model. The statement has_many :tags, :through => :taggings would tell ActiveRecord to do this. In turn, each tag has many taggings, meaning that you can find which objects have been tagged with a specific tag.

It also uses another interesting feature of ActiveRecord—polymorphic associations. This means that the join association is not limited to one particular model, it can be associated with any model, since the model name is stored in the join model itself.

To do this, we need to add a database column specifying the model type. The Tagging join model and the Tag model are defined within the `acts_as_taggable_on_steroids` plug-in, but we still have to create the database tables for the plug-in to be able to work. The plug-in requires two tables: one to store the tag names and one to store the relationships of the tags with other models.

The database table for the Tag model is shown in Table 10-1.

Table 10-1. *The Tags Table*

Field Name	Field Type	Description
`id`	`integer`	The primary key
`name`	`string`	The tag name

The database table for the Tagging model is shown in Table 10-2. We will never access this table directly; it is only used to associate one model with another. However, since it is a real model and not just a database table, we can add extra attributes or callbacks.

Because we are using polymorphic associations, this table includes the column `taggable_type` to store the model name as well as `taggable_id` to store the associated model ID.

Table 10-2. *The Tagging Database Table*

Field Name	Field Type	Description
`id`	`integer`	The primary key
`tag_id`	`integer`	The `id` of the tag
`taggable_id`	`integer`	The `id` of the taggable object
`taggable_type`	`string`	The model name of the taggable object
`created_at`	`datetime`	The time and date that this tagging was created

To use the tagging library with a particular model, you simply add the statement `acts_as_taggable` to a model class definition. Doing this adds a number of instance and class methods to the model, which are used to add tags to an object and to find objects that are tagged with particular tags.

To add tags to an object or list the tags that have been assigned to an object, you use the methods `tag_list` and `tag_list=`. These allow you to access a comma-separated list of the object's tags.

For example, if you have a `photo` object called `@photo`, you would assign tags with the following command:

```
@photo.tag_list = "puppy,dog,cute"
@photo.save
```

After you have saved the tagged object, you can access the tags for that object either with the `tag_list` instance method or by accessing the `tag` objects that belong to the object:

```
@photo.tags_list
```

```
=> "puppy, dog, cute"
```

```
@photo.tags
```

```
=> [#<Tag:0x3136034 @attributes={"name"=>"puppy", "id"=>"4"}>,
#<Tag:0x313600c @attributes={"name"=>"dog", "id"=>"5"}>,
#<Tag:0x3135fe4 @attributes={"name"=>"cute", "id"=>"6"}>]
```

In order to find objects that are tagged with a particular tag, we use the class method `find_tagged_with`, for instance:

```
@photos = Photo.find_tagged_with('puppy')
```

If you want to find objects with any one of multiple tags, you can just specify them as a list separated by commas:

```
@photos = Photo.find_tagged_with('puppy, dog')
```

To find objects that have all of the listed tags, use the `:match_all` parameter:

```
@photos = Photo.find_tagged_with('puppy, dog', :match_all => true)
```

You can also use regular find options such as `:order`, `:limit`, and `:offset` as part of the `find_tagged_with` method.

The plug-in also has a useful method that allows us to easily create tag clouds. A tag cloud is simply a list of tags, but the size of each tag is proportional to its popularity. In order to produce a tag cloud, we need to know the frequency of each tag's use. The plug-in provides an instance method `tag_counts` for the tagged model. It will return an array of hashes containing the tag name, ID, and the number of times that this tag has been used, for instance:

```
Photos.tag_counts
```

```
=> [#<Tag:0x30a2014 @attributes={"name"=>"puppy", "id"=>"4", "count"=>"1"}>,
#<Tag:0x30a1fec @attributes={"name"=>"dog", "id"=>"5", "count"=>"2"}>,
#<Tag:0x30a1fc4 @attributes={"name"=>"cute", "id"=>"6", "count"=>"4"}>]
```

If you wish to use this to find the frequency of use for tags belonging to a specific user, you must extend the `has_many` association of the user to the photos like this:

```
class User < ActiveRecord::Base
  has_many :photos, :extend => TagCountsExtension
end
```

You can then use the tag_count method for a specific user, for example:

```
User.find(1).photos.tag_counts
```

```
=> [#<Tag:0x30671bc @attributes={"name"=>"puppy", "id"=>"4", "count"=>"1"}>,
#<Tag:0x3067194 @attributes={"name"=>"dog", "id"=>"5", "count"=>"1"}>,
#<Tag:0x306716c @attributes={"name"=>"cute", "id"=>"6", "count"=>"2"}>]
```

TagCountsExtension should only be used on associations where you have declared the model to use acts_as_taggable.

Building the Photo Tagging Feature

We will need to update the Photo model to declare that it will use acts_as_taggable. We also need to extend the User model to use the TagCountsExtension.

We need to create two controllers. One will be accessed from the root path; we will call this simply tags_controller. The other will be nested beneath the user resource; we will call this user_tags_controller. We then need to create the relevant mappings for these controllers in the routes file.

However, first we need to install the acts_as_taggable_on_steroids plug-in.

Installing the acts_as_taggable_on_steroids Plug-in

The acts_as_taggable_on_steroids plug-in is distributed simply as a Rails plug-in. To install the plug-in, use the normal Rails plugin script. Enter the following command:

```
$ ruby script/plugin install ➥
  http://svn.viney.net.nz/things/rails/plugins/acts_as_taggable_on_steroids
```

```
+ ./acts_as_taggable_on_steroids/CHANGELOG
+ ./acts_as_taggable_on_steroids/MIT-LICENSE
+ ./acts_as_taggable_on_steroids/README
+ ./acts_as_taggable_on_steroids/Rakefile
+ ./acts_as_taggable_on_steroids/init.rb
+ ./acts_as_taggable_on_steroids/lib/acts_as_taggable.rb
+ ./acts_as_taggable_on_steroids/lib/tag.rb
+ ./acts_as_taggable_on_steroids/lib/tag_counts_extension.rb
+ ./acts_as_taggable_on_steroids/lib/tagging.rb
+ ./acts_as_taggable_on_steroids/test/abstract_unit.rb
+ ./acts_as_taggable_on_steroids/test/acts_as_taggable_test.rb
+ ./acts_as_taggable_on_steroids/test/database.yml
+ ./acts_as_taggable_on_steroids/test/fixtures/photo.rb
+ ./acts_as_taggable_on_steroids/test/fixtures/photos.yml
+ ./acts_as_taggable_on_steroids/test/fixtures/post.rb
+ ./acts_as_taggable_on_steroids/test/fixtures/posts.yml
+ ./acts_as_taggable_on_steroids/test/fixtures/taggings.yml
+ ./acts_as_taggable_on_steroids/test/fixtures/tags.yml
+ ./acts_as_taggable_on_steroids/test/fixtures/user.rb
+ ./acts_as_taggable_on_steroids/test/fixtures/users.yml
+ ./acts_as_taggable_on_steroids/test/schema.rb
+ ./acts_as_taggable_on_steroids/test/tag_test.rb
+ ./acts_as_taggable_on_steroids/test/tagging_test.rb
```

Creating the Database Tables

To create the database tables needed by the tagging plug-in, we will create a migration and add the changes to the database to that.

Create the migration file using the Rails generator script:

```
$ ruby script/generate migration AddTaggingSupport
```

```
      exists  db/migrate
      create  db/migrate/021_add_tagging_support.rb
```

Now, open the migration file `db/migrate/021_add_tagging_support.rb`, and add the migration code shown in Listing 10-1.

Listing 10-1. *The Migration to Add Tagging Support*

```
class AddTaggingSupport < ActiveRecord::Migration
  def self.up
    create_table :tags, :force => true do |t|
      t.column :name, :string
    end

    create_table :taggings, :force => true do |t|
      t.column :tag_id, :integer
      t.column :taggable_id, :integer
      t.column :taggable_type, :string
      t.column :created_at, :datetime
    end

    add_index :tags, :name
    add_index :taggings, [:tag_id, :taggable_id, :taggable_type]
  end

  def self.down
    drop_table :tags
    drop_table :taggings
  end
end
```

This will create the necessary tables, along with instructing the database to create indexes based on the `tags.name` field and the `taggings.tag_id`, `taggable_id`, and `taggable_type` fields. Since all database queries will be based on these fields rather than the primary key, it makes sense to add these indexes now.

Next, run the `migrate` command to perform these changes to the database:

```
$ rake db:migrate
```

```
== AddTaggingSupport: migrating ============================================
-- create_table(:tags, {:force=>true})
   -> 0.1125s
-- create_table(:taggings, {:force=>true})
   -> 0.0070s
-- add_index(:tags, :name)
   -> 0.0157s
-- add_index(:taggings, [:tag_id, :taggable_id, :taggable_type])
   -> 0.0079s
== AddTaggingSupport: migrated (0.1240s) ===================================
```

Updating the Models

As we have discussed, we need to update both the Photo and User models. Open the Photo model file, `app/models/photo.rb`, and add the statement `acts_as_taggable` as shown in Listing 10-2.

Listing 10-2. *The Modification to the Photo Model*

```
class Photo < ActiveRecord::Base
  acts_as_taggable
  belongs_to :user
  ...
```

We now need to update the user file's relationship with the Photo model, adding the `TagCountsExtension`. Open the User model file, `app/models/user.rb`. Now modify the photo relationship as shown in Listing 10-3.

Listing 10-3. *The Modification to the User Model*

```
require 'digest/sha2'
class User < ActiveRecord::Base
  attr_protected :hashed_password, :enabled
  attr_accessor :password
  ...
  has_many :usertemplates
  has_many :comments
  has_many :photos, :extend => TagCountsExtension
  ...
```

Creating the Controllers

As we discussed in the requirements, we will create a controller for each different way of accessing the tags:

- To view all tags, via URLs such as `/tags` and `/tags/tree`, we will create and use a controller called `tags_controller.rb`.
- To view tags belonging to a specific user, via URLs such as `/user/1/tags` and `/user/1/tags/tree`, we will use a controller called `user_tags_controller.rb`.

We should create these controllers and add the relevant mappings to the routes file now. Create `tags_controller.rb` with the Rails `generate` command:

```
$ ruby script/generate controller Tags
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/tags
      exists  test/functional/
      create  app/controllers/tags_controller.rb
      create  test/functional/tags_controller_test.rb
      create  app/helpers/tags_helper.rb
```

Next, create `user_tags_controller.rb`:

```
$ ruby script/generate controller UserTags
```

```
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/user_tags
      exists  test/functional/
      create  app/controllers/user_tags_controller.rb
      create  test/functional/user_tags_controller_test.rb
      create  app/helpers/user_tags_helper.rb
```

Adding the Resource Mappings

We now need to map the URLs to the specific controllers in the routes file. Open the file `config/routes.rb`. Add the tags mapping, and modify the existing users mapping to add the nested tags. We can also add the new methods, `add_tag` and `remove_tag`, for the existing `user_photos` resource mapping. The `add_tag` method uses HTTP `PUT`, while the `remove_tag` method uses HTTP `DELETE`.

Edit the routes file as shown in Listing 10-4, adding the bold lines.

Listing 10-4. *Updates to the Route Mappings File*

```
map.resources :photos
map.resources :tags

map.resources :users, :member => { :enable => :put } do |users|
  users.resources :permissions
  users.resources :entries do |entries|
    entries.resources :comments
  end
  users.resources :tags, :name_prefix => 'user_',
                         :controller => 'user_tags'
  users.resources :photos, :name_prefix => 'user_',
                           :controller => 'user_photos',
                           :member => { :add_tag => :put,
                                        :remove_tag => :delete }
end
```

Writing the Controllers and Views

We can now write the code to actually perform the actions set up in the mappings.

The Tags Controller

The `tag_controller.rb` `index` method displays all of the tags that have been added to photos on the site, regardless of user. Since we want to display this as a tag cloud, we should retrieve the tags using the `tag_counts` method.

The `show` method will show all of the photos that match a particular tag. This is done simply with the method `find_tagged_with`.

Since it is not possible for a user to create, update, or delete tags through this resource, only as an update to a photo, we only have to create the `index` and `show` methods.

Open the `app/controllers/tag_controller.rb` file, and edit it as shown in Listing 10-5.

Listing 10-5. *The Tag Controller File*

```
class TagsController < ApplicationController

  def index
    @tags = Photo.tag_counts(:order => 'name')
  end

  def show
    @photos = Photo.find_tagged_with(params[:id])
  end

end
```

For the tag `index` action, we have requested that the tags be ordered alphabetically using the tag name. We now need to create the views for the `index` and `show` actions.

The Tag Index View

To create the tag cloud for the tag index view, we will create a helper method that takes the array of tags with usage counts and return a series of CSS class names that we can add to the displayed tags. The class names are assigned based on how common a tag is in relation to the other tags. This code is based on a Rails helper developed by Tom Fakes. You can find the original code at http://blog.craz8.com/articles/2005/10/28/acts_as_taggable-is-a-cool-piece-of-code.

First of all, we should add the tag_cloud helper to a helper file. Since we will use this helper from both the tags_controller and the user_tags_controller files, we should add the tag_cloud helper to the applicationwide helper file.

Open app/helpers/application_helper.rb, and add the new tag_cloud helper to the ApplicationHelper module as shown in Listing 10-6.

Listing 10-6. *The Tag Cloud Helper*

```
# Methods added to this helper will be available to all templates in
the application.
module ApplicationHelper
  def yes_no(bool)
    ...
  end

  def tag_cloud(tags, classes)
    max, min = 0, 0
    tags.each do |tag|
      max = tag.count if tag.count > max
      min = tag.count if tag.count < min
    end

    divisor = ((max - min) / classes.size) + 1

    tags.each do |tag|
      yield tag.name, classes[(tag.count - min) / divisor]
    end
  end
end
```

We can now create the index view file, app/views/tags/index.rhtml. Create this file, open it, and enter the code in Listing 10-7.

Listing 10-7. *The Tags Index View*

```
<h2>Most Popular Tags</h2>

<% tag_cloud @tags, %w(tag1 tag2 tag3 tag4 tag5) do |name, css_class| %>
  <%= link_to name, tag_path(name), :class => css_class %>
<% end %>
```

This passes the @tags array (which includes the count attribute) and an array of CSS class names to the tag_cloud helper. %w() is simply a quick way to create an array from a list of words in Ruby.

We use the returned data from the helper in a Ruby block, taking the name and calculated CSS class name and using them to generate a link. This links to the show action of the tag resource.

We now need to create the definitions of the CSS classes that the tag_cloud helper uses. Since we want the size of the text to be proportional to the tag's popularity, we will just set the font-size attribute for each of the classes.

Open the style sheet for the application, public/stylesheets/main.css, and add the CSS code shown in Listing 10-8 to the end of the file.

Listing 10-8. *The CSS Style Sheet for the Tag Cloud*

```
/* Tag cloud styling */
.tag1 { font-size: 100%; }
.tag2 { font-size: 120%; }
.tag3 { font-size: 140%; }
.tag4 { font-size: 160%; }
.tag5 { font-size: 170%; }
.tag6 { font-size: 180%; }
```

The Tag Show View

To create the view for the show action, we simply have to render the partial view that has already been created for the regular photo gallery. Create the show view file, app/views/tags/show.rhtml, and add the code in Listing 10-9.

Listing 10-9. *The Tags Show View File*

```
<h2>Photos Tagged: <%=h params[:id] %></h2>

<ul id="photos">
  <%= render :partial => 'photos/photo', :collection => @photos %>
</ul>
```

The User Tags Controller

The user_tags_controller is used in a similar way to tags_controller, except that it only shows tags and photos for a specific user. Like tags_controller, this controller also needs only the index and show actions, since tags are never edited through this controller.

Open the generated controller, app/controller/user_tags_controller.rb, and edit it as shown in Listing 10-10.

Listing 10-10. *The User Tags Controller File*

```
class UserTagsController < ApplicationController

  def index
    @user = User.find(params[:user_id])
    @tags = @user.photos.tag_counts(:order => 'name')
  end

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

end
```

You will notice that this is almost the same as the tags_controller, except that we first retrieve the user specified in the URL and then search for tags or photos with a specific tag within the scope of that user. This also means that the view files will also be very similar.

The User Tags Index View

We will use the same tag_cloud helper method as the tags_controller index view. Create the file app/views/user_tags/index.rhtml, and add the view code in Listing 10-11.

Listing 10-11. *The User Tags Index View*

```
<h2><%= @user.username %>'s Most Popular Tags</h2>

<p><%= link_to "Show all user's tags", tags_path %></p>

<% tag_cloud @tags, %w(tag1 tag2 tag3 tag4 tag5) do |name, css_class| %>
  <%= link_to name, tag_path(name), :class => css_class %>
<% end %>
```

Since we have retrieved the specified user's details, we can use that to display the user's name as the title of the page. We have also added a link to go to the root-level tag view, showing all of the tags on the site.

The User Tags Show View

This view, showing the user's photos tagged with a specific word, also makes use of the existing thumbnail partial view that we wrote for the photo gallery. Create the file app/views/user_tags/show.rhtml, and enter the code shown in Listing 10-12.

Listing 10-12. *The User Tags Show View*

```
<h2><%= @user.username %>'s Photos Tagged: <%=h params[:id] %></h2>

<p>
  <%= link_to "Show all photos tagged with #{h(params[:id])}", tag_path(h(params[:id])) %>
</p>

<ul id="photos">
  <%= render :partial => 'photos/photo', :collection => @photos %>
</ul>
```

Along with showing all the photos in the specified user's gallery tagged with the requested word, we have also included a link to show all photos on the site tagged with this word.

Adding Tags to a Photo

We need to develop the controller methods and interface to allow users to add tags to their photos.

In the routes file, we added mappings for two extra methods for the `user_photo` resource. Take another look at the mapping in the `routes.rb` file:

```
users.resources :photos, :name_prefix => 'user_', :controller => 'user_photos',
                         :member => { :add_tag => :put, :remove_tag => :delete }
```

This adds the actions `add_tag` and `remove_tag` to the nested resource, which is accessible through the URLs `/user/1/photos/2;add_tag` and `/user/1/photos/2;remove_tag`. We can use the shortcuts `user_add_tag_photo_path` and `user_remove_tag_photo_path` to access these in views and controllers. We also need to make sure that we specify the correct HTTP method to access these: `PUT` for `add_tag` and `DELETE` for `remove_tag`.

Allowing the User to Add Tags to a Photo

First of all, we will develop the code necessary to add tags to a `photo` object. We will just add this to the existing `user_photos_controller file`. Open the file `app/controllers/user_photos_controller.rb`. Within the `UserPhotosController` class, create the new action method shown in Listing 10-13.

Listing 10-13. *The add_tag Method*

```
def add_tag
  @photo = @logged_in_user.photos.find(params[:id])
  @photo.tag_list += ',' + params[:tag][:name]
  @photo.save
  @new_tag = @photo.reload.tags.last
end
```

The `add_tag` method is very simple. First, it retrieves the photo to which a tag is being added. This is found using the `user_id` of the currently logged-in user and the `id` parameter given in the URL.

As you will recall, we add tags to an object by specifying them as a comma-separated list in a string. Since we don't want to remove the tags that are already given for the tag, we just add the new tag, prefixed by a comma, to the end of the string. We then save the photo object.

Finally, we retrieve this new tag as a Tag model. This allows the view to access this new tag as it would any other tag, rather than having to deal with it as just a string.

Normally, we would then automatically render an HTML view or redirect to a different action. We could simply redirect to the `edit` action, which would reload the entire edit page in the user's browser. However, we are going to use Ajax techniques to update just the existing list of tags on the edit page.

To do this, we need to display the list of tags and add the form to enter a tag onto the photo edit page.

Edit the file `app/views/user_photos/edit.rhtml` as shown in Listing 10-14.

Listing 10-14. *The Updated user_photos Edit File*

```
<h2>Editing photo</h2>

<%= error_messages_for :photo %>

<%= link_to image_tag(@photo.public_filename('thumb')),
            user_photo_path(:user_id => @photo.user, :id => @photo) %>

<h3>Tags</h3>
<ul id="taglist">
  <%= render :partial => 'edit_tag', :collection => @photo.tags %>
</ul>

<% remote_form_for(:tag,
                   :url => user_add_tag_photo_path(:id => @photo),
                   :method => :put,
                   :complete => "Field.clear('tag-name')") do |f| %>
  <%= f.text_field :name, :id => 'tag-name' %>
  <%= submit_tag 'Add Tag' %>
<% end %>

<% form_for(:photo,
            :url => user_photo_path(:user_id => @photo.user, :id => @photo),
            :html => { :method => :put }) do |f| %>
  <p>Title:<br /><%= f.text_field 'title' %></p>
  <p>Description:<br /><%= f.text_area 'body', :rows => 6, :cols => 40 %></p>
  <p><%= submit_tag "Save" %> or <%= link_to 'cancel', user_photos_path %></p>
<% end %>
```

Here, we have added a list of the tags using the `render :partial` command, so we need to write this partial view. Create the file `app/views/user_photos/_edit_tag.rhtml`, and enter the partial view code in Listing 10-15. Since we will not use this partial in the photos controller, we should place it in the `user_photos` view directory.

Listing 10-15. *The edit_tag Partial View*

```
<li id="tag-<%= edit_tag.id %>">
  <span><%= edit_tag.name %></span>
</li>
```

Note that we are adding the `id` of the tag to the `id` attribute of the `<li>` tag. Although the `class` and `id` attributes of HTML objects are often used just for styling using CSS, we can also use them to find a specific part of the document. In this case, we will use this to allow us to delete a tag from the tag list by specifying exactly which page element we wish to remove.

We are using a Rails helper method `remote_form_for` to create the form where a user enters a new tag. This works in a similar way to the `form_for` helper that we have used in all of our `new` and `edit` views so far, except `remote_form_for` uses XMLHttpRequest to submit the form in the background rather than as a regular HTTP `POST`, which would force a page reload. This is achieved using a JavaScript library, which collects the form elements then submits them to our application. You can then process this in exactly the same way as you would a regular HTTP request. As you can see in the `add_tag` method, we still use `params[:tag][:name]` to access the form parameters.

The `remote_form_for` helper takes the same parameters as the `form_for` helper, so we still specify the destination URL and HTTP method. But we can also use special callbacks to perform JavaScript actions on the page. These callbacks are shown in Table 10-3.

Table 10-3. *The Ajax Callbacks*

Callback	Called When
`:loading`	The remote document is being loaded by the browser.
`:loaded`	The browser has finished loading the remote document.
`:interactive`	The user can interact, even if the document has not finished loading.
`:success`	The remote document has loaded and has a success HTTP Status code.
`:failure`	The remote document has loaded but does not have a success HTTP Status code.
`:complete`	The remote document has been completely loaded.

We are using the `:complete` callback to clear the tag name form field. This allows the user to enter a number of tags quickly, without having to manually clear the form field first.

So now that we have a form that can call the `add_tag` action in the background, we need to define what gets returned to the browser.

We have already seen how Rails can easily respond to different types of requests with the `respond_to` statement. This time, we will respond only to JavaScript requests.

The response that we want to send to the browser is a piece of JavaScript code that will instruct the browser to add the new tag to the end of the existing tag list. If we were sending an HTML page, we would write an `.rhtml` file. However, since we want to send JavaScript, Rails uses a different type of file to allow us to define a JavaScript response. These files are called Remote JavaScript (RJS) files.

RJS files work in a very similar way to an `.rhtml` or `.rxml` view file—you simply create a file with the same name as the action that you are responding to but with the suffix of `.rjs`.

Since our action method is called `add_tag`, create the corresponding RJS file, `app/views/user_photos/add_tag.rjs`. Enter the RJS code shown in Listing 10-16 to this file.

Listing 10-16. *The add_tag RJS File*

```
page.insert_html :bottom, 'taglist', { :partial => 'edit_tag',
                                       :locals => {:edit_tag => @new_tag} }
page.visual_effect :highlight, "tag-#{@new_tag.id}", :duration => 2
```

RJS files allow us to change data that is currently on the page. In this instance, we insert a new instance of the `edit_tag` partial at the bottom of the page element tag list. We need to set the new tag object, `@new_tag`, as a local variable to the partial.

We then call the visual effect method, telling it to highlight the newly created tag for a period of 2 seconds.

If we wanted to support browsers that were not capable of processing JavaScript, we could use the `respond_to` statement. If we wanted to support both JavaScript and HTML responses, we would add the following lines:

```
format.html
format.js
```

This renders the relevant template based on how the request was received.

RJS TEMPLATES

RJS template files are simply snippets of Ruby code that use a DSL to generate JavaScript code that is then sent to the requesting browser.

RJS relies on the Prototype and script.aculo.us JavaScript libraries that are shipped with Rails and should be included as part of the application layout with the tag `<%= javascript_include_tag :defaults %>`.

When you write an RJS file, you have access to an object called `page`, which is simply an instance of the Rails `JavaScriptGenerator` class. All the desired responses are made as method calls to the `page` object.

You can call a large number of available methods that allow you to change, remove, or add content to the page; make sections draggable; produce alert boxes; hide or display page elements; and so on.

To find out more about RJS, there is a very useful list of resources at the Ruby Inside blog `http://www.rubyinside.com/16-rjs-resources-and-tutorials-for-rails-programmers-5.html`.

For a more advanced reference to RJS templates, the e-book *RJS Templates for Rails* by Cody Fauser (O'Reilly, 2006) is a worthwhile purchase.

Before we try this out, we should add the `delete_tag` method and update the partial to allow us to easily delete tags.

Removing a Tag from a Photo Object

To give the users the option of removing a tag from one of their photo objects, we have to write the action method to remove the particular tag, write the response (in this case another RJS file), and add an option to the user interface to allow the users to perform this action easily.

Since we are using a partial to list the tags on the edit photo page, we can simply update this partial to show a delete link next to each tag.

Reopen this partial view file, app/views/user_photo/_edit_tag.rhtml, and edit it as shown in Listing 10-17 to add the delete link.

Listing 10-17. *Updated edit_tag Partial View*

```
<li id="tag-<%= edit_tag.id %>">
  <span><%= edit_tag.name %></span>
  <small>
    [<%= link_to_remote 'delete',
          :url => user_remove_tag_photo_path(:id => @photo.id,
                                             :tag_id => edit_tag.id),
          :method => :delete %>]
  </small>
</li>
```

This uses the Rails helper method link_to_remote. This works in the same way as the remote_form_for helper, making the remote request in the background without making the browser reload the whole page. This time, we have to specify the method to be DELETE, since we have specified in the routes file that the remove_tag action can only be accessed by the DELETE method. If you wished to perform other JavaScript actions before or after the link_to_remote method, you could also add the Ajax callbacks mentioned earlier. However, we do not require any callbacks to be executed.

We now need to write the remove_tag action method itself and the corresponding RJS file. Open the file app/controllers/user_photos_controller.rb, and add the remove_tag method shown in Listing 10-18 after the add_tag method but before the closing end statement.

Listing 10-18. *The remove_tag Action Method*

```
def remove_tag
  @photo = @logged_in_user.photos.find(params[:id])
  @tag_to_delete = @photo.tags.find(params[:tag_id])

  if @tag_to_delete
    @photo.tags.delete(@tag_to_delete)
  else
    render :nothing => true
  end
end
```

The method first retrieves the photo being modified. Since it only searches the photos within the scope of the @logged_in_user, it is impossible for someone who is not logged in as the photo owner to modify the photo's tags.

We then search the tags set for this photo for the tag specified in the request parameters. Only if this tag exists for this photo do we attempt to delete the tag and respond with the RJS file.

To delete the tag, we simply need to remove it from the tags associated with this photo. Since we already have the @tag_to_delete object, we simply call the delete method on @photo.tags to remove the specified tag object.

We now need to write the RJS template to define the response to this request. Create the file app/views/user_photos/remove_tag.rjs, and add the RJS code in Listing 10-19.

Listing 10-19. *The remove_tag RJS Template*

```
page.remove "tag-#{@tag_to_delete.id}"
page.visual_effect :highlight, 'taglist', :duration => 2
```

This RJS file simply removes the page element with the id of the tag that we are deleting. Since our page renders each tag with the tag object id prefixed with tag-, we can specify exactly which item in the list we wish to remove.

We then highlight the entire tag list for 2 seconds to show the user that the list has changed.

Linking to the Tag Browser

Finally, we should add some links to make it quick and easy for a visitor to browse the site using tags.

We should add a link to the menu sidebar, linking to the root-level tag index view, and we should also show all of the tags for a particular photo on the photo show page. We can also add a link to show users' tags from their profile pages.

Adding Tags to the Sidebar Menu

Open the sidebar menu partial file, app/views/layouts/_menu.rhtml, and add a link to the tags_controller index view as shown in Listing 10-20.

Listing 10-20. *Adding the Tag Index Link to the Sidebar Menu*

```
...
<li><%= link_to 'Blogs', all_blogs_path %></li>
<li><%= link_to 'Photos', photos_path %></li>
<li><%= link_to 'Photo Tags', tags_path %></li>

<li><hr size="1" width="90%" align="left"/></li>
...
```

Adding Tag Links to the Photo Show View

When a user views a particular photo, this is shown by the view app/views/user_photos/show.rhtml. Open this file now. Beneath the photo title and description, we will show the list of tags for this photo.

We could simply use the tag_list method, which would render a string listing all of the tags for this photo separated by commas. While this is fine, it would be much more useful to render each tag as a link, linking to the user_tags controller's show action. We will do this by cycling through the tags and creating a link for each tag.

Modify the file by adding the code as shown in Listing 10-21.

Listing 10-21. *The Updated user_photos Show View*

```
.. .
<p><%=h @photo.body %></p>

<p>Tags:
  <% @photo.tags.each do |tag| %>
    <%= link_to tag.name, user_tag_path(@photo.user, tag.name)%>
  <% end %>
</p>

<% if is_logged_in? && @photo.user_id == logged_in_user.id %>
...
```

Adding a Link on the Users Profile Page

Finally, we should add a link to the user's tag index page on their individual profile. Open the user profile's `show` view, `app/views/users/show.rhtml`, and add a link to the bottom of the page as shown in Listing 10-22.

Listing 10-22. *The Updated User Show View*

```
...
<p>
  <%= link_to "See all of #{@user.username}'s photos",
        user_photos_path(:user_id => @user) %>
</p>
<p>
  <%= link_to "#{@user.username}'s Tags",
        user_tags_path(:user_id => @user) %>
</p>
```

Now that all of the pieces are in place, we can run through the feature and manually test the functions.

Manually Testing

Fire up your browser, and go to the application home page, `http://localhost:3000/`. Log in to the site as one of the users you have created. Now, go to your photos page. If you have not uploaded any photos as this user, you should upload a few now.

View one of these photos by clicking on the photo itself. Since this photo belongs to you, the "edit" and "delete" links will be shown on this page. Click the "edit" link. This will show your photo along with edit boxes for the title and description, and a new text box allowing you to enter a tag.

Enter a tag that describes the picture, and press Return/Enter or click the Add Tag button. This new tag will appear in the tag list above the tag text entry box, as shown in Figure 10-1, and be highlighted briefly.

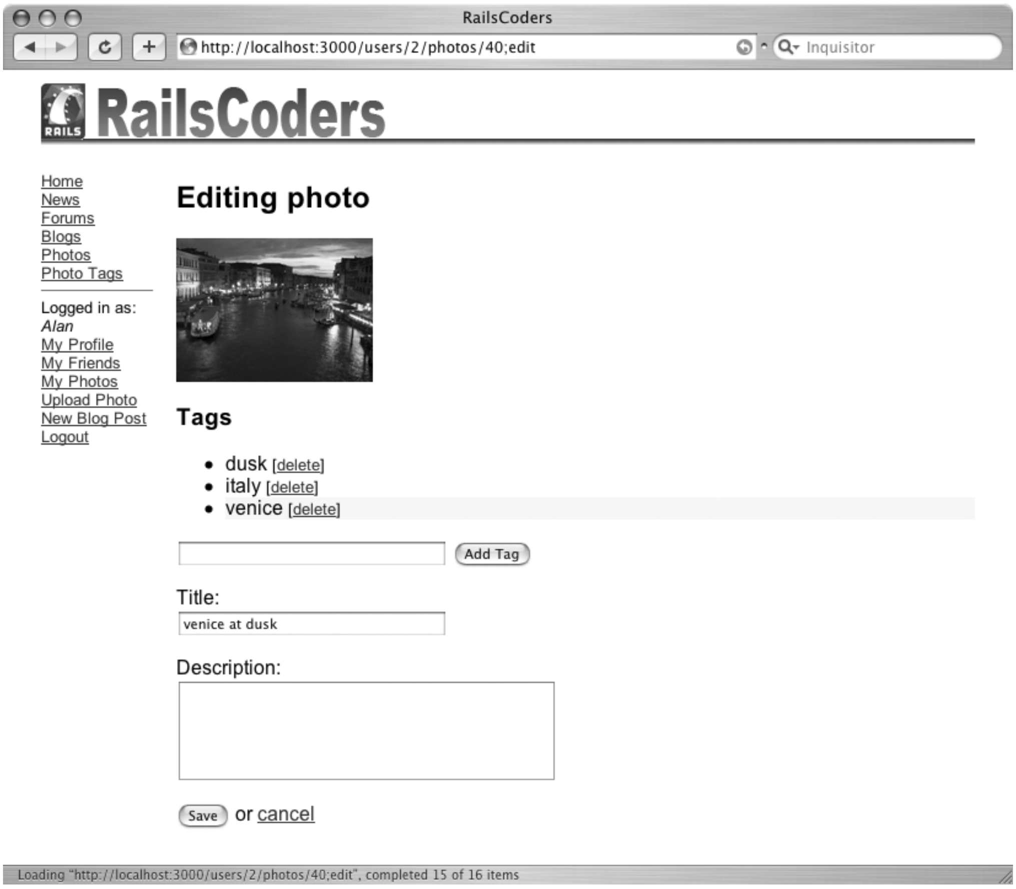


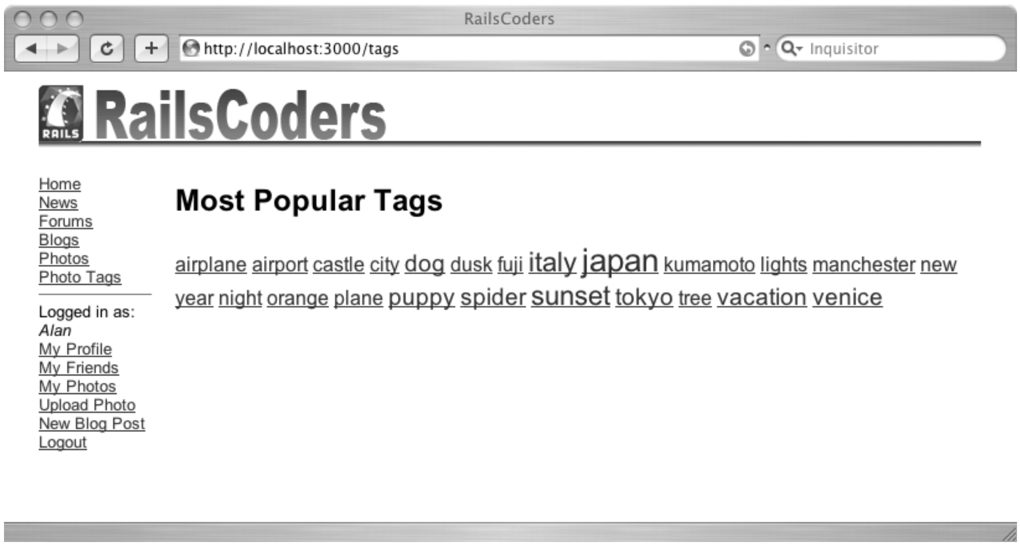
Figure 10-1. *The "Editing photo" screen with the tag entry box*

Do this with a number of photos, using the same tag and new tags. This will give us an interesting tag cloud view.

Next, click the Photo Tags link in the sidebar menu to display all of the tags added on the site with their popularity shown by the size of the font, as shown in Figure 10-2.

Now try clicking one of the tags to show all of the photos for that particular tag. You will notice that the tag being viewed is simply given as part of the URL.

Log out of the site and log in as a different user. Try adding some tags to photos owned by this other user, and take a look at the tag views for both the previous user and this new user—you will see that the tag clouds are unique for each user, as expected.



Most Popular Tags

airplane airport castle city dog dusk fuji italy japan kumamoto lights manchester new year night orange plane puppy spider sunset tokyo tree vacation venice

Figure 10-2. *The photos tag cloud*

Further Development of the Tagging System

This tagging feature can be extended further in a number of ways:

- Right now, the tag `index` action shows all tags. This is fine for a small site with a limited number of tags, but it will soon become excessive on a large-scale site. You could limit the number of tags shown by the `index` action to a hundred or so of the most popular tags, which should be about a page full of tags.
- Also, the pages showing all photos with a specific tag are not paginated. You should consider paginating them if you anticipate a large number of photos.
- When showing photos that are tagged with a keyword, you could retrieve other tags used to tag the same photos. This would produce a related tags list, allowing the users to browse the photo galleries easily.
- You could also show a list of users who have most frequently used a particular tag.
- You may also wish to add tags to other objects on the site, such as blog entries.

Summary

In this chapter, we have added a complete tagging system to the photo gallery. We used the `acts_as_taggable_on_steroids` plug-in to add tagging features to the Photo model and developed an interface for the user.

This involved developing an Ajax-based system to allow the tags to be added and deleted dynamically from the photo edit page. This used the `remote_form_for` and `remote_link_to` helpers and RJS templates to send the data and dynamically construct JavaScript from the Ruby RJS file.

We also created a tag cloud of all the tags on the site, allowing users to quickly see which tags are the most popular.

In the next chapter, we will look at how we can integrate with other web applications, in particular Google Maps and Flickr, by using their public APIs.