CHAPTER 11
Searching and browsing

In principle, our alphabetical community index lets any user find any other user, but using it in this way would be terribly cumbersome. In this chapter, we add more convenient and powerful ways to find users. We begin by adding full-text search to RailsSpace by making use of an open-source project called *Ferret*. We then stalker-enable our site with browsing by age, sex, and location.

Adding search and browse capability to RailsSpace will involve the creation of custom pagination and validations, which means that we will start to rely less on the built-in Rails functions. This chapter also contains a surprising amount of geography, some fairly fancy `find`s, and even a little math.

11.1 Searching

Though it was quite a lot of work to get the community index to look and behave just how we wanted, the idea behind it is very simple. In contrast, full-text search—for user information, specs, and FAQs—is a difficult problem, and yet most users probably expect a site such as RailsSpace to provide it. Luckily, the hardest part has already been done for us by the Ferret project,[1] a full-text search engine written in Ruby. Ferret makes adding full-text search to Rails applications a piece of cake through the `acts_as_ferret` plugin.

In this section, we'll make a simple search form (adding it to the main community page in the process) and then construct an action that uses Ferret to search RailsSpace based on a query submitted by the user.

[1] `http://ferret.davebalmain.com/trac/`

11.1.1 Search views

Since there's some fairly hairy code on the back-end, it will be nice to have a working search form that we can use to play with as we build up the search action incrementally. Since we'll want to use the search form in a couple of places, let's make it a partial:

Listing 11.1 app/views/community/_search_form.rthml

```
<% form_tag({ :action => "search" }, :method => "get") do %>
<fieldset>
  <legend>Search</legend>
    <div class="form_row">
      <label for="q">Search for:</label>
      <%= text_field_tag "q", params[:q] %>
      <input type="submit" value="Search" />
    </div>
</fieldset>
<% end %>
```

This is the first time we've constructed a form without using the `form_for` function, which is optimized for interacting with models. For search, we're not constructing a model at any point; we just need a simple form to pass a query string to the search action. Rails makes this easy with the `form_tag` helper, which has the prototype

```
form_tag(url_for_options = {}, options = {})
```

The `form_tag` function takes in a block for the form; when the block ends, it automatically produces the `</form>` tag to end the form. This means that the rhtml

```
<% form_tag({ :action => "search" }, :method => "get") do %>
.
.
.
<% end %>
```

produces the HTML

```
<form action="/community/search" method="get">
.
.
.
</form>
```

Note that in this case we've chosen to have the search form submit using a GET request, which is conventional for search engines (and allows, among other things, direct linking to search results since the search terms appear in URL).

As in the case of the `link_to` in the community index (Section 10.3.3), the curly braces around `{ :action => "search" }` are necessary. If we left them off and wrote instead

```
<% form_tag(:action => "search", :method => "get") %>
.
.
.
<% end %>
```

then Rails would generate

```
<form action="/community/search?method=get" method="post">
.
.
.
</form>
```

instead of

```
<form action="/community/search" method="get">
.
.
.
</form>
```

The other Rails helper we use is `text_field_tag`, which makes a text field filled with the value of `params[:q]`. That is, if `params[:q]` is `"foobar"`, then

```
<%= text_field_tag "q", params[:q] %>
```

produces the HTML

```
<input id="q" name="q" type="text" value="foobar" />
```

We've done a lot of work making useful partials, so the search view itself is beautifully simple:

Listing 11.2 app/views/community/search.rthml

```
<%= render :partial => "search_form" %>
<%= render :partial => "result_summary" %>
<%= render :partial => "user_table" %>
```

We'll also put the search form on the community index page (but only if there is no `@initial` variable, since when the initial exists we want to display only the users whose last names begin with that letter):

Listing 11.3 app/views/community/index.rhtml

```
.
.
.
<% if @initial.nil? %>
  <%= render :partial => "search_form" %>
<% end %>
```

Figure 11.1 The evolving community index page now includes a search form.

You can submit queries to the resulting search page (Figure 11.1) to your heart's content, but of course there's a hitch: It doesn't do anything yet. Let's see if we can ferret out a solution to that problem.

11.1.2 Ferret

As its web page says, "Ferret is a high-performance, full-featured text search engine library written for Ruby." Ferret, in combination with `acts_as_ferret`, builds up an index of the information in any data model or combination of models. In practice, what this means is that we can search through (say) the user specs by associating the special `acts_as_ferret` attribute with the Spec model and then using the method `Spec.find_by_contents`, which is added by the `acts_as_ferret` plugin. (If this all seems overly abstract, don't worry; there will be several concrete examples momentarily.)

Ferret is relatively easy to install, but it's not entirely trouble-free. On OS X it looks something like this:[2]

```
> sudo gem install ferret
Attempting local installation of 'ferret'
```

[2] As with the installation steps in Chapter 2, if you don't have `sudo` enabled for your user, you will have to log in as root to install the ferret gem.

```
Local gem file not found: ferret*.gem
Attempting remote installation of 'ferret'
Updating Gem source index for: http://gems.rubyforge.org
Select which gem to install for your platform (powerpc-darwin7.8.0)
 1. ferret 0.10.11 (ruby)
 2. ferret 0.10.10 (ruby)
 3. ferret 0.10.9 (mswin32)
 .
 .
 .
 39. Cancel installation
> 1
Building native extensions.  This could take a while...
.
.
.
Successfully installed ferret, version 0.10.11
```

The process is virtually identical for Linux; in both Mac and Linux cases, you should choose the most recent version of Ferret labeled "(ruby)", which should be #1. If, on the other hand, you're using Windows, run

```
> gem install ferret
```

and be sure to choose the most recent version of Ferret labeled "mswin32", which probably won't be the first choice.

The second step is to install the Ferret plugin:[3]

```
> ruby script/plugin install svn://projects.jkraemer.net/acts_as_ferret/tags/
stable/acts_as_ferret
A    /rails/rails_space/vendor/plugins/acts_as_ferret
A    /rails/rails_space/vendor/plugins/acts_as_ferret/LICENSE
A    /rails/rails_space/vendor/plugins/acts_as_ferret/rakefile
A    /rails/rails_space/vendor/plugins/acts_as_ferret/init.rb
A    /rails/rails_space/vendor/plugins/acts_as_ferret/lib
A    /rails/rails_space/vendor/plugins/acts_as_ferret/lib/more_like_this.rb
A    /rails/rails_space/vendor/plugins/acts_as_ferret/lib/multi_index.rb
A    /rails/rails_space/vendor/plugins/acts_as_ferret/lib/acts_as_ferret.rb
A    /rails/rails_space/vendor/plugins/acts_as_ferret/lib/instance_methods.rb
A    /rails/rails_space/vendor/plugins/acts_as_ferret/lib/class_methods.rb
A    /rails/rails_space/vendor/plugins/acts_as_ferret/README
```

[3] If you don't have the version control system Subversion installed on your system, you should download and install it at this time (`http://subversion.tigris.org/`). If you have experience compiling programs from source, you should have no trouble, but if you are more comfortable with Windows installations, then you should skip right to `http://subversion.tigris.org/servlets/ProjectDocumentList?folderID=91` and download the `svn-<version>-setup.exe` with the highest version number. Double-clicking on the resulting executable file will then install Subversion.

That may look intimidating, but the good news is that you don't have to touch any of these files. All you have to do is restart the development webserver to activate Ferret and then indicate that the models are searchable using the (admittedly somewhat magical) `acts_as_ferret` function:

Listing 11.4 app/models/spec.rb

```
class Spec < ActiveRecord::Base
  belongs_to :user
  acts_as_ferret
  .
  .
  .
```

Listing 11.5 app/models/faq.rb

```
class Faq < ActiveRecord::Base
  belongs_to :user
  acts_as_ferret
  .
  .
  .
```

Listing 11.6 app/models/user.rb

```
class User < ActiveRecord::Base
  has_one :spec
  has_one :faq
  acts_as_ferret :fields => ['screen_name', 'email'] # but NOT password
  .
  .
  .
```

Notice in the case of the User model that we used the `:fields` options to indicate which fields to make searchable. In particular, we made sure not to include the password field!

11.1.3 Searching with `find_by_contents`

Apart from implying that he occasionally chases prairie dogs from their burrows, what does it mean when we say that a user `acts_as_ferret`? For the purposes of RailsSpace search, the answer is that `acts_as_ferret` adds a function called `find_by_contents`

that uses Ferret to search through the model, returning results corresponding to a given query string (which, in our case, comes from the user-submitted search form). The structure of our search action builds on `find_by_contents` to create a list of matches for the query string:

Listing 11.7 app/controllers/community_controller.rb

```
def search
  @title = "Search RailsSpace"
  if params[:q]
    query = params[:q]
    # First find the user hits...
    @users = User.find_by_contents(query, :limit => :all)
    # ...then the subhits.
    specs = Spec.find_by_contents(query, :limit => :all)
    faqs  =  Faq.find_by_contents(query, :limit => :all)
    .
    .
    .
```

Here we've told Ferret to find all the search hits in each of the User, Spec, and FAQ models.

Amazingly, that's all there is to it, as far as search goes: Just those three lines are sufficient to accomplish the desired search. In fact, if you submit a query string from the search form at this point, the results should be successfully returned—though you will probably find that your system takes a moment to respond, since the first time Ferret searches the models it takes a bit of time while it builds an index of search results. This index, which Ferret stores in a directory called `index` in the Rails root directory, is what makes the magic happen—but it is also the source of some problems (see the sidebar "A dead Ferret").

A dead Ferret[4]

Occasionally, when developing with Ferret, the search results will mysteriously disappear. This is usually associated with changes in the database schema (from a migration, for example). When Ferret randomly croaks in this manner, the solution is simple:

[4] "He's not dead—he's resting!"

1. Shut down the webserver.
2. Delete Ferret's `index` directory.
3. Restart the webserver.

At this point, Ferret will rebuild the index the next time you try a search, and everything should work fine.

Now that we've got the search results from Ferret, we have to collect the users for display; this requires a little Ruby array manipulation trickery:

Listing 11.8 app/controllers/community_controller.rb

```
def search
  if params[:q]
    query = params[:q]
    # First find the user hits...
    @users = User.find_by_contents(query, :limit => :all)
    # ...then the subhits.
    specs = Spec.find_by_contents(query, :limit => :all)
    faqs  =  Faq.find_by_contents(query, :limit => :all)

    # Now combine into one list of distinct users sorted by last name.
    hits = specs + faqs
    @users.concat(hits.collect { |hit| hit.user }).uniq!
    # Sort by last name (requires a spec for each user).
    @users.each { |user| user.spec ||= Spec.new }
    @users = @users.sort_by { |user| user.spec.last_name }
end
```

This introduces the `concat` and `uniq!` functions, which work like this:

```
> irb
irb(main):001:0> a = [1, 2, 2, 3]
=> [1, 2, 2, 3]
irb(main):002:0> b = [4, 5, 5, 5, 6]
=> [4, 5, 5, 5, 6]
irb(main):003:0> a.concat(b)
=> [1, 2, 2, 3, 4, 5, 5, 5, 6]
irb(main):004:0> a
=> [1, 2, 2, 3, 4, 5, 5, 5, 6]
irb(main):005:0> a.uniq!
=> [1, 2, 3, 4, 5, 6]
irb(main):006:0> a
=> [1, 2, 3, 4, 5, 6]
```

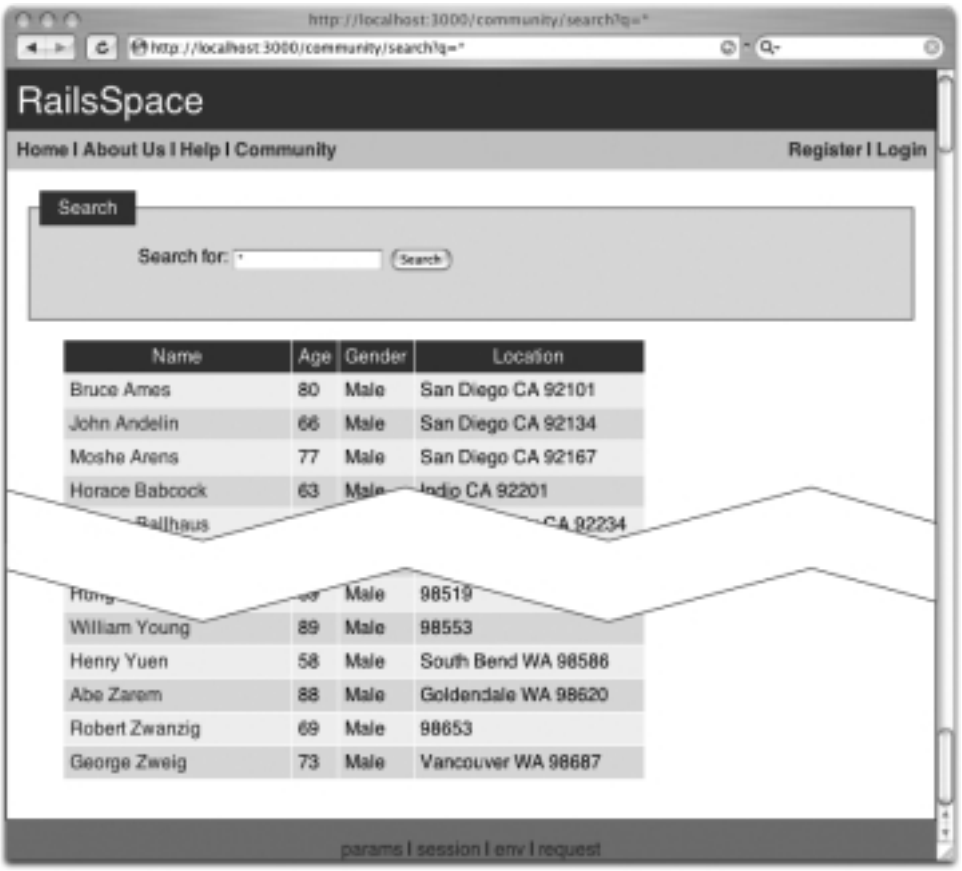

Figure 11.2 Search results for `q=*`, returning unpaginated results for all users.

You can see that `concat` concatenates two arrays—`a` and `b`—by appending `b` to `a`, while `a.uniq!` modifies `a`[5] by removing duplicate values (thereby ensuring that each element is unique).

We should note that the line

```
@users = @users.sort_by { |user| user.spec.last_name }
```

also introduces a new Ruby function, used here to sort the users by last name; it's so beautifully clear that we'll let it pass without further comment.

At this stage, the search page actually works, as you can see from Figure 11.2. But, like the first cut of the RailsSpace community index, it lacks a result summary and

[5] Recall from Section 6.6.2 that the exclamation point is a hint that an operation mutates the object in question.

pagination. Let's make use of all the work we did in Section 10.4 and add those features to the search results.

11.1.4 Adding pagination to search

Now that we've collected the users for all of the search hits, we're tantalizingly close to being done with search. All we have to do is paginate the results and add the result summary. In analogy with the pagination from Section 10.4.1, what we'd really like to do is this:

Listing 11.9 app/controllers/community_controller.rb

```
def search
  if params[:q]
    .
    .
    .
    @pages, @users = paginate(@users)
  end
end
```

Unfortunately, the built-in `paginate` function only works when the results come from a single model. It's not too hard, though, to extend `paginate` to handle the more general case of paginating an arbitrary list—we'll just use the `Paginator` class (on which `paginate` relies) directly. Since we'd like the option to paginate results in multiple controllers, we'll put the `paginate` function in the Application controller:

Listing 11.10 app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  .
  .
  .
  # Paginate item list if present, else call default paginate method.
  def paginate(arg, options = {})
    if arg.instance_of?(Symbol) or arg.instance_of?(String)
      # Use default paginate function.
      collection_id = arg  # arg is, e.g., :specs or "specs"
      super(collection_id, options)
    else
      # Paginate by hand.
      items = arg  # arg is a list of items, e.g., users
      items_per_page = options[:per_page] || 10
      page = (params[:page] || 1).to_i
```

```
      result_pages = Paginator.new(self, items.length, items_per_page, page)
      offset = (page - 1) * items_per_page
      [result_pages, items[offset..(offset + items_per_page - 1)]]
    end
  end
end
```

There is some moderately advanced Ruby here, but we'll go through it step by step. In order to retain compatibility with the original `paginate` function, the first part of our `paginate` checks to see if the given argument is a symbol or string (such as, for example, `:specs` as in Section 10.4.1), in which case it calls the original `paginate` function using `super` (a usage we saw before in Section 9.5).

If the first argument is not a symbol or string, we assume that it's an array of items to be paginated. Using this array, we create the result pages using a `Paginator` object, which is initialized as follows:

```
Paginator.new(controller, item_count, items_per_page, current_page=1)
```

In the context of the Application controller, the first argument to `new` is just `self`, while the item count is just the length of `items` and the items per page is either the value of `options[:per_page]` or 10 (the default). We get the number of the current page by using

```
page = (params[:page] || 1).to_i
```

which uses the `to_i` function to convert the result to an integer, since `params[:page]` will be a string if it's not `nil`.[6]

Once we've created the results pages using the Paginator, we calculate the array indices needed to extract the page from `items`, taking care to avoid off-by-one errors. For example, when selecting the third page (`page = 3`) with the default pagination of 10,

```
offset = (page - 1) * items_per_page
```

yields

```
offset = (3 - 1) * 10 = 20
```

This means that

```
items[offset..(offset + items_per_page - 1)]
```

is equivalent to

```
items[20..39]
```

which is indeed the third page.

[6] Calling `to_i` on `1` does no harm since it's already an integer.

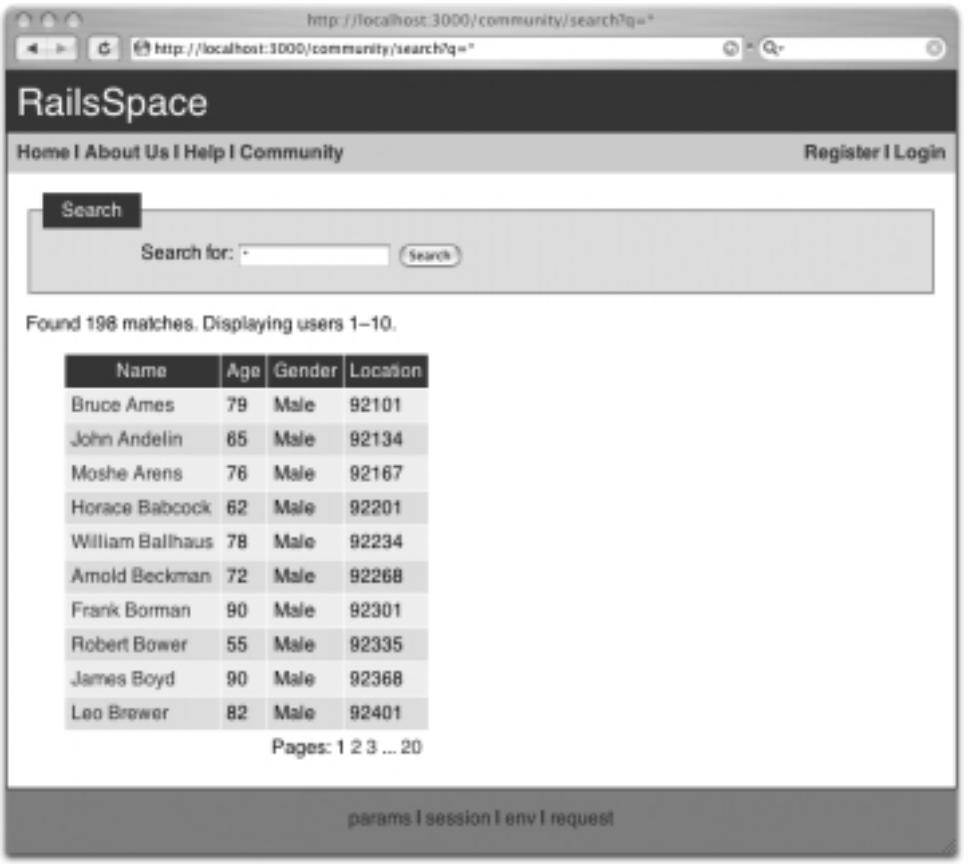

Figure 11.3 Search results for `q=*`, returning paginated results for all users.

Finally, at the end of `paginate`, we return the two-element array

```
[result_pages, items[offset..(offset + items_per_page - 1)]]
```

so that the object returned by our `paginate` function matches the one from the original `paginate`.

That's a lot of work, but it's worth it; the hard-earned results appear in Figure 11.3. Note that if you follow the link for (say) page 2, you get the URL of the form

```
http://localhost:3000/community/search?page=2&q=*
```

which contains the query string as a parameter. This works because back in Section 10.4.1 we told `pagination_links` about the `params` variable:

Listing 11.11 app/views/community/_user_table.rhtml

```
.
.
.
  Pages: <%= pagination_links(@pages, :params => params) %>
.
.
.
```

Rails knows to include the contents of `params` in the URL.

11.1.5 An exception to the rule

We're not quite done with search; there's one more thing that can go wrong. Alas, some search strings cause Ferret to croak. In this case, as seen in Figure 11.4, Ferret raises the exception

```
Ferret::QueryParser::QueryParseException
```

indicating its displeasure with the query string `"--"`.[7]

The way to handle this in Ruby is to wrap the offending code in a `begin...rescue` block to catch and handle the exception:

Listing 11.12 app/controllers/community_controller.rb

```
def search
  if params[:q]
    query = params[:q]
    begin
      # First find the user hits...
      @users = User.find_by_contents(query, :limit => :all)
      # ...then the subhits.
      specs = Spec.find_by_contents(query, :limit => :all)
      faqs  =  Faq.find_by_contents(query, :limit => :all)
      # Now combine into one list of distinct users sorted by last name.
      hits = specs + faqs
      @users.concat(hits.collect { |hit| hit.user }).uniq!
      # Sort by last name (requires a spec for each user).
      @users.each { |user| user.spec ||= Spec.new }
      @users = @users.sort_by { |user| user.spec.last_name }

      @pages, @users = paginate(@users)
    rescue Ferret::QueryParser::QueryParseException
```

Continues

[7] This appears to be fixed as of Ferret 0.11.0.

Figure 11.4 Ferret throws an exception when given an invalid search string.

```
      @invalid = true
    end
  end
end
```

Here we tell `rescue` to catch the specific exception raised by Ferret parsing errors, and then set the `@invalid` instance variable so that we can put an appropriate message in the view (Figure 11.5):

Listing 11.13 app/views/community/search.rhtml

```
<%= render :partial => "search_form" %>
<% if @invalid %>
<p>Invalid character in search.</p>
```

Figure 11.5 The ferret query parse exception caught and handled.

```
<% end %>
<%= render :partial => "result_summary" %>
<%= render :partial => "user_table" %>
```

And with that, we're finally done with search!

11.2 Testing search

Testing the search page is easy in principle: Just hit `/community/search` with an appropriate query string and make sure the results are what we expect. But a key part of testing search should be to test the (currently untested) pagination. Since we're using the default pagination value of 10, that means creating at least eight more users to add to the three currently in our users fixture:[8]

Listing 11.14 test/fixtures/users.yml

```
# Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html
valid_user:
  id: 1
  screen_name: millikan
```

Continues

[8] Even though one of these users is invalid, it still exists in the test database when the Rails test framework loads the fixtures; Ferret doesn't know anything about validations, so it gamely finds all three users.

```
  email: ram@example.com
  password: electron

invalid_user:
  id: 2
  screen_name: aa/noyes
  email: anoyes@example,com
  password: sun

# Create a user with a blank spec.
specless:
  id: 3
  screen_name: linusp
  email: lpauling@example.com
  password: 2nobels
```

Of course, we could hand-code eight more users, but that's a pain in the neck. Fortunately, Rails has anticipated our situation by enabling embedded Ruby in YAML files, which works the same way that it does in views. This means we can generate our extra users automatically by adding a little ERb to `users.yml`:

Listing 11.15 test/fixtures/users.yml

```
.
.
.
# Create 10 users so that searches can invoke pagination.
<% (1..10).each do |i| %>
user_<%= i %>:
  id: <%= i + 3 %>
  screen_name: user_<%= i %>
  email: user_<%= i %>@example.com
  password: foobar
<% end %>
```

Note that our generated users have ids given by `<%= i + 3 %>` rather than `<%= i %>` in order to avoid conflicts with the previous users' ids.

With these extra 10 users, a search for all users using the wildcard query string `"*"` should find a total of 13 matches, while displaying matches 1–10:

Listing 11.16 test/functional/community_controller_test

```
.
.
.
class CommunityControllerTest < Test::Unit::TestCase
```

```
  fixtures :users
  fixtures :specs
  fixtures :faqs

  .
  .
  .

  def test_search_success
    get :search, :q => "*"
    assert_response :success
    assert_tag "p", :content => /Found 13 matches./
    assert_tag "p", :content => /Displaying users 1&ndash;10./
  end
end
```

This gives

```
> ruby test/functional/community_controller_test.rb -n test_search_success
Loaded suite test/functional/community_controller_test
Started
.
Finished in 0.849541 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
```

Despite being short, this test catches several common problems, and proved valuable while developing the search action.

11.3 Beginning browsing

Because Ferret does the heavy search lifting, browsing for users—though less general than search—is actually more difficult. In this section and the next (Section 11.4), we'll set out to create pages that allow each user to find others by specifying age (through a birthdate range), sex, and location (within a particular distance of a specified zip code)—the proverbial "A/S/L" from chat rooms. In the process, we'll create a nontrivial custom form (with validations) and also gain some deeper experience with the Active Record `find` function (including some fairly fancy SQL).

11.3.1 The browse page

Let's start by constructing a browse page, which will be a large custom (that is, non-`form_for`) form. On the back-end, the action is trivial for now:

Listing 11.17 app/views/controllers/community_controller.rb

```
    def browse
      @title = "Browse"
    end
```

The browse view is also trivial, since it just pushes the hard work into a partial:

Listing 11.18 app/views/community/browse.rhtml

```
<%= render :partial => "browse_form" %>
<%= render :partial => "result_summary" %>
<%= render :partial => "user_table" %>
```

This brings us to the browse form itself, which is relatively long but whose structure is simple. Using Rails tag helpers and the `params` variable, we build up a form with fields for each of the A/S/L attributes:

Listing 11.19 app/views/community/_browse_form.rhtml

```
<% form_tag({ :action => "browse" }, :method => "get") do %>
<fieldset>
  <legend>Browse</legend>
  <div class="form_row">
    <label for="age">Age:</label>
    <%= text_field_tag "min_age", params[:min_age], :size => 2 %>
    &ndash;
    <%= text_field_tag "max_age", params[:max_age], :size => 2 %>
  </div>
  <div class="form_row">
    <label for="gender">Gender:</label>
    <%= radio_button_tag :gender, "Male",
                         params[:gender] == 'Male',
                         :id => "Male" %>Male
    <%= radio_button_tag :gender, "Female",
                         params[:gender] == 'Female',
                         :id => "Female" %>Female
  </div>
  <div class="form_row">
    <label for="location">Location:</label>
    Within
    <%= text_field_tag "miles", params[:miles], :size => 4 %>
    miles from zip code:
    <%= text_field_tag "zip_code", params[:zip_code],
                       :size => Spec::ZIP_CODE_LENGTH %>
  </div>
```

```
  <%= submit_tag "Browse", :class => "submit" %>
</fieldset>
<% end %>
```

As in Section 11.1.1, we use `text_field_tag`, which has the function prototype

```
text_field_tag(name, value = nil, options = {})
```

so that if, for example, `params[:min_age]` is 55, the code

```
<%= text_field_tag "min_age", params[:min_age], :size => 2 %>
```

produces the HTML

```
<input id="min_age" name="min_age" size="2" type="text" value="55" />
```

Similarly, we have the radio button helper,

```
radio_button_tag(name, value, checked = false, options = {})
```

Then if `params[:gender]` is `"Female"`, the code

```
<%= radio_button_tag :gender, "Female",
                     params[:gender] == 'Female',
                     :id => "Female" %>Female
```

produces

```
<input checked="checked" id="Female" name="gender" type="radio" value="Female" />
```

with the `Female` box "checked"[9] since `params[:gender] == 'Female'` is true.

With the browse form partial thus defined, the browse view is already in its final form (Figure 11.6).

11.3.2 Find by A/S/L (hold the L)

The browse form already "works" in the sense that it doesn't break if you submit it, and it even remembers the values you entered (Figure 11.7). Apart from that, though, it doesn't actually do anything. Let's take the first step toward changing that:

Listing 11.20 app/views/controllers/community_controller.rb

```
  def browse
    @title = "Browse"
    return if params[:commit].nil?
```

Continues

[9] It's actually filled in rather than checked since it's a radio button and not a checkbox, but we can't help the terminology used by the HTML standard.

Figure 11.6 The final browse form.

```
  specs = Spec.find_by_asl(params)
  @pages, @users = paginate(specs.collect { |spec| spec.user })
end
```

In keeping with our usual practice, we've hidden as many details as possible beneath an abstraction layer, in this case the function `find_by_asl`, which we've chosen to be a class method for the Spec model.

Figure 11.7 The browse form with some values submitted.

We'll implement `find_by_asl` momentarily, but first we need to explain the line

```
return if params[:commit].nil?
```

You may have noticed in Figure 11.7 that the string `browse=commit` appears in the URL;[10] this means that `params[:commit]` tells us if the form has been submitted. As a result,

```
return if params[:commit].nil?
```

returns if the form hasn't been submitted, thereby causing Rails to render the browse form immediately. (In previous chapters, we used the `param_posted?` function defined in Section 6.6.5 to detect form submission via `POST` requests, but, like the search form, the browse form uses a GET request instead.)

Having addressed the case of hitting the browse page directly, it's now time to handle browse form submission by writing `find_by_asl` for browsing by age and gender. (Though fairly tricky, the age and gender searches are much easier than the search by location, so we defer the latter to Section 11.4.2.) Browsing by age and gender involves the trickiest database query so far, so we'll discuss each piece of the puzzle before assembling them into the final `find_by_asl` method. For concreteness, let's consider the case of searching for all female RailsSpace members between the ages of 55 and 65.[11]

First, let's consider the essential form of the query we need to make. In MySQL, the code to select females with ages between 55 and 65 would look something like this:

```
SELECT * FROM specs WHERE
ADDDATE(birthdate, INTERVAL 55 YEAR) < CURDATE() AND
ADDDATE(birthdate, INTERVAL 66 YEAR) > CURDATE() AND
gender = 'Female'
```

This uses `CURDATE()`, which returns the current date, as well as the MySQL `ADDDATE` function, which is convenient for doing date arithmetic. For example, we use the code

```
ADDDATE(birthdate, INTERVAL 66 YEAR) > CURDATE()
```

to select specs with birthdates that give a date *after* the current date when you add 66 years to them—which will be true for anyone age 65 or younger.

Next, we'll introduce a new aspect of the `:conditions` option in `find`. Recall from Section 10.3.1 that we can ensure safe SQL queries by using question marks as

[10] `browse=commit` is inserted automatically by the Rails `submit_tag` helper.

[11] Recall that our sample data is based on Caltech distinguished alumni, with made-up ages starting at 50.

string place-holders; for example, assuming a suitable `params` variable, we could use the following to find all RailsSpace users of a particular gender:

```
Spec.find(:all, :conditions => ["gender = ?", params[:gender]])
```

The new syntax, which we will use in `find_by_asl`, uses a symbol and a full hash instead:

```
Spec.find(:all, :conditions => ["gender = :gender", params])
```

In this case, when building up the SQL query corresponding to this particular `find`, Rails knows to insert an escaped-out version of `params[:gender]` in place of `:gender`.

Finally, the last piece of the puzzle is Ruby's array append syntax `<<`, which we can demonstrate using an `irb` session:

```
> irb
irb(main):001:0> a = []
=> []
irb(main):002:0> a << "foo"
=> ["foo"]
irb(main):003:0> a << "bar" << "baz"
=> ["foo", "bar", "baz"]
irb(main):004:0> a.join(" AND ")
=> "foo AND bar AND baz"
```

Note from line `003` that array appends can be chained together. We've also anticipated a key step in building up the `find` conditions by joining the array elements on `" AND "` in the final line.

Our strategy for `find_by_asl` is to make an array of strings with one element for each potential part of the `WHERE` clause. We'll then join that array with `" AND "` for use in `:conditions`. A call to `find` will then perform the query using the SQL string we've constructed. Putting everything together leads to the following method:

Listing 11.21 app/models/spec.rb

```
# Find by age, sex, location.
def self.find_by_asl(params)
  where = []
  # Set up the age restrictions as birthdate range limits in SQL.
  unless params[:min_age].blank?
    where << "ADDDATE(birthdate, INTERVAL :min_age YEAR) < CURDATE()"
  end
  unless params[:max_age].blank?
    where << "ADDDATE(birthdate, INTERVAL :max_age+1 YEAR) > CURDATE()"
  end
  # Set up the gender restriction in SQL.
  where << "gender = :gender" unless params[:gender].blank?
```

```
    if where.empty?
      []
    else
      find(:all,
           :conditions => [where.join(" AND "), params],
           :order => "last_name, first_name")
    end
  end
```

Note that we've elected to return an empty list if there are no restrictions; another option would be to return *all* users in that case, but we think returning no users makes more sense. We've also added the obligatory ordering by last name, first name in the call to `find`.

By the way, it's worth noting that our method for performing queries in `find_by_sql` violates database independence, which has both advantages and disadvantages (see the sidebar "Getting database religion").

Getting database religion

In building up a `where` string for use in the `:conditions` option, we have used MySQL-specific code such as `ADDDATE`, thereby violating database agnosticism (and thus becoming *database theists*). This is not such a bad choice, when you consider the alternative. To maintain database independence, we would have to select all of the users and then apply the various conditions to the resulting Ruby array. For a sufficiently small user base, this would be no problem, but it scales horribly with the number of users, since it requires loading a significant part of the database into memory for every call to `find_by_sql`. Building up a query string, on the other hand, allows us to perform the query all at once in the database—thereby making use of exactly what databases are good at.

In the present case, our judgment is that the benefit of breaking database-independence outweighs the cost, but we should be mindful that we would have to rewrite `find_by_sql` if we ever switched to a database other than MySQL.

With `find_by_asl` thus defined, the browse form is live, and searches by age and gender work essentially as advertised (Figure 11.8). What remains is to add location search—a decidedly nontrivial task, but one we will nevertheless rise to accomplish.

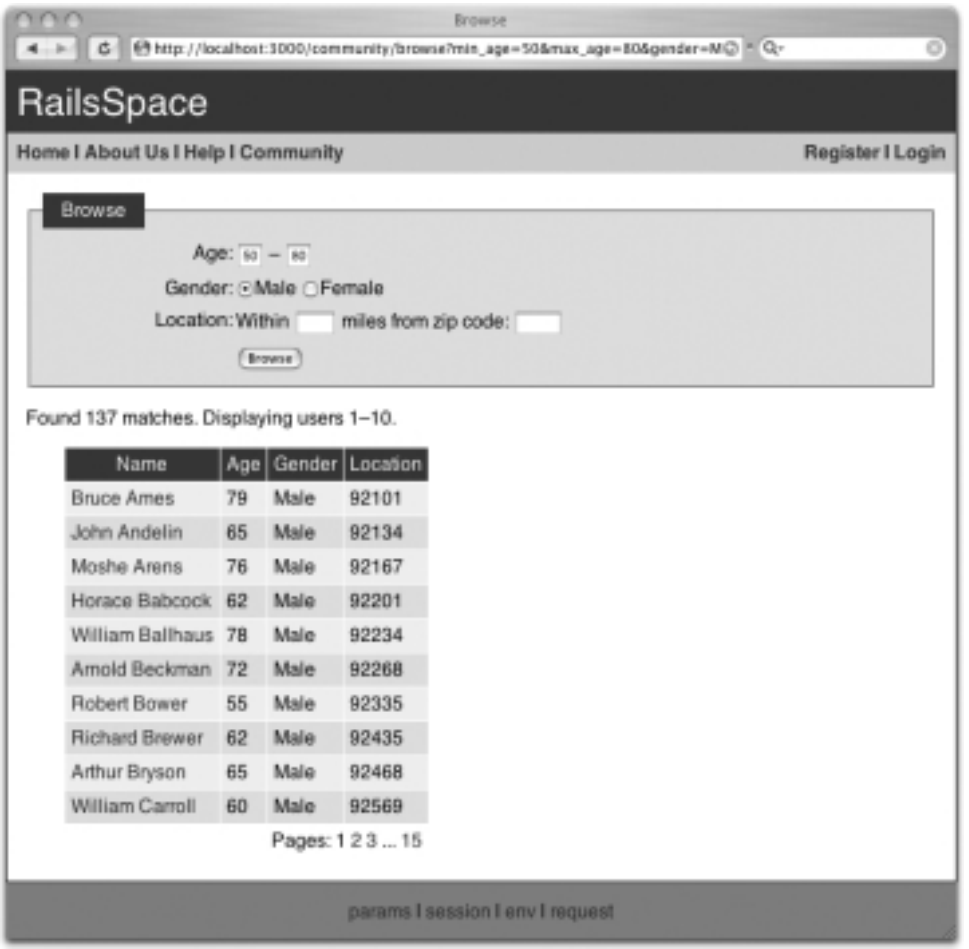

Figure 11.8 Browsing our database by restricting spec parameters.

11.4 Location, location, location

In order to add the "L" in "A/S/L" to our browse feature, we need to include some geographical knowledge in the RailsSpace database. Once we've done that, we'll be in a position to make the distance calculation needed to find all locations within a certain radius of a given zip code. While we're at it, we'll use our newfound geographical prowess to add some polish to the user display tables.

11.4.1 A local database of geographical data

We need to populate our local database with the locations (in latitude and longitude) of various zip codes. We'll use a free zip code database found at

`http://www.populardata.com/`

That data works fine on OS X and Linux, but you need to massage it a little bit to get it to work on Windows; a Windows-friendly version of the data (as well as a copy of the original) can be found at

`http://www.RailsSpace.com/book`

After you download the file (text version), unzip it and rename it to `geo_data.csv`. Since we want all the RailsSpace databases (development, test, and eventually production) to have the geographical information, we'll put the data-loading step in a migration; for convenience, move `geo_data.csv` to the `db/migrate` directory. Then, create the migration, which creates a table called `geo_data` together with the relevant columns:

```
> ruby script/generate migration CreateGeoData
    exists  db/migrate
    create  db/migrate/007_create_geo_data.rb
```

Here is the migration itself:

Listing 11.22 db/migrate/007_create_geo_data.rb

```
class CreateGeoData < ActiveRecord::Migration
  def self.up
    create_table :geo_data do |t|
      t.column :zip_code,  :string
      t.column :latitude,  :float
      t.column :longitude, :float
      t.column :city,      :string
      t.column :state,     :string
      t.column :county,    :string
      t.column :type,      :string
    end
    add_index "geo_data", ["zip_code"], :name => "zip_code_optimization"

    csv_file = "#{RAILS_ROOT}/db/migrate/geo_data.csv"
    fields = '(zip_code, latitude, longitude, city, state, county)'

    execute "LOAD DATA INFILE '#{csv_file}' INTO TABLE geo_data FIELDS " +
            "TERMINATED BY ',' OPTIONALLY ENCLOSED BY \"\"\"\" " +
            "LINES TERMINATED BY '\n' " + fields
  end
```

Continues

```
    def self.down
      drop_table :geo_data
    end
  end
```

```
> rake db:migrate
  (in /rails/rails_space)
  == CreateGeoData: migrating ================================================
  -- create_table("geo_data")
     -> 0.0883s
  -- execute("LOAD DATA INFILE '/rails/rails_space/config/../db/migrate/geo_data."
csv' INTO TABLE
geo_data FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY \"\"\"\" LINES"
TERMINATED BY '\n'(zip_code, latitude, longitude, city, state, county, type)")
     -> 0.9792s
  == CreateGeoData: migrated (1.0684s) =======================================
```

The migration is a bit advanced, and it would take us too far afield to go into all the details,[12] but once you've run it as above, you should be able to see a promising table called `geo_data` in the database (Figure 11.9). You can see there that the geographical database contains a correspondence between zip codes and latitude/longitude, as well as the city, state, and even county of each location.

Since we will want to manipulate GeoData objects using Active Record—using, in particular, the `find_by_zip_code` method automatically created due to the `zip_code` database column—we need to create a (virtually) blank model just to tell Rails that `GeoDatum`[13] inherits from `ActiveRecord::Base`:

Listing 11.23 app/models/geo_datum.rb

```
class GeoDatum < ActiveRecord::Base
end
```

With that, we're ready to do the actual distance search.

11.4.2 Using GeoData for location search

When last we left the browse action, we had optimistically called our new find function `find_by_asl`.

Now that the RailsSpace database is geographically aware, it's time to add the "L."

[12] It's worth noting, though, that through the `execute` command we can execute arbitrary SQL queries in a migration.

[13] Yes, the Rails inflector knows that the singular of `GeoData` is `GeoDatum`.

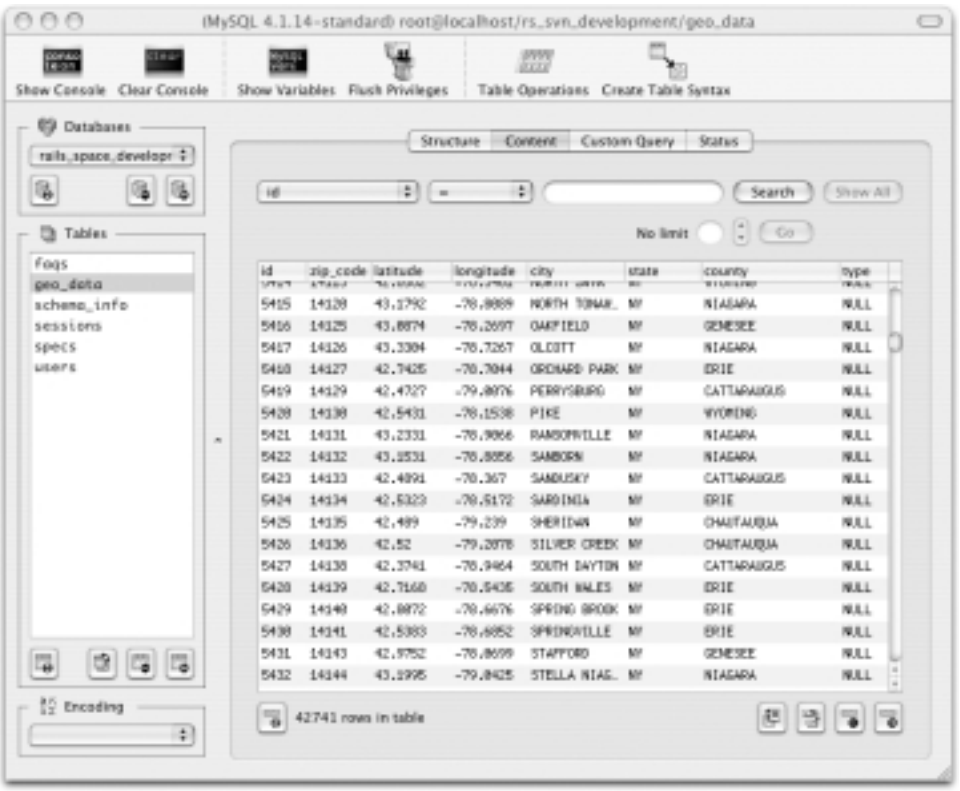

Figure 11.9 The geographical data in the database.

Our strategy will be to take the user-submitted zip code, find the location in the geographical database, and then select every spec whose zip code is within the given number of miles of that location. If this sounds suspiciously like math, you're right: We'll have to use a formula for calculating distances on a sphere as a function of latitude and longitude.[14] We'll start by writing a function in the Spec model that returns a string appropriate for calculating the distance between the given point and an arbitrary location (as identified by longitude and latitude):

[14] Seriously, what were the chances that a couple of Caltech Ph.D.'s could write a whole book without using at least a little math?

Listing 11.24 app/models/spec.rb

```
  .
  .
  .
  private

  # Return SQL for the distance between a spec's location and the given point.
  # See http://en.wikipedia.org/wiki/Haversine_formula for more on the formula.
  def self.sql_distance_away(point)
    h = "POWER(SIN((RADIANS(latitude - #{point.latitude}))/2.0),2) + " +
        "COS(RADIANS(#{point.latitude})) * COS(RADIANS(latitude)) * " +
        "POWER(SIN((RADIANS(longitude - #{point.longitude}))/2.0),2)"
    r = 3956 # Earth's radius in miles
    "2 * #{r} * ASIN(SQRT(#{h}))"
  end
end
```

As noted in the comments, this uses the *haversine formula* for calculating distances on a sphere, which can be found (among other places) at

`http://en.wikipedia.org/wiki/Haversine_formula`

We're now ready to add distance search to `find_by_asl`. We can get the location using `GeoDatum.find_by_zip_code` and then add the requirement that the (SQL) distance away is less than or equal to the miles supplied through the browse form:

Listing 11.25 app/models/spec.rb

```
  # Find by age, sex, location.
  def self.find_by_asl(params)
    where = []
    .
    .
    .
    where << "gender = :gender" unless params[:gender].blank?

    # Set up the distance restriction in SQL.
    zip_code = params[:zip_code]
    unless zip_code.blank? and params[:miles].blank?
      location = GeoDatum.find_by_zip_code(zip_code)
      distance = sql_distance_away(location)
      where << "#{distance} <= :miles"
    end

    if where.empty?
      []
    else
```

```
    find(:all,
         :joins => "LEFT JOIN geo_data ON geo_data.zip_code = specs.zip_code",
         :conditions => [where.join(" AND "), params],
         :order => "last_name, first_name")
  end
end
```

By the way, if you were nervous about the appearance of `latitude` in `sql_distance_away`, that's a good sign—after all, the Spec model has no such column, so there's no way such a query could work. The solution, as you might infer from the `find` above, is to *join* the `geo_data` and `specs` tables, which effectively endows each spec with latitude and longitude attributes. So far in this book, we've used lots of joins, but they've always been implicit, since Active Record handles the details for us; in this case, we need an explicit join in order to do the age, sex, and location select all in one step. Moreover, we want to find users by age and gender even if the zip code is blank, which means we need a specific kind of operation called a *left join*[15]—with an ordinary join, a search for female users, say, would return only those users who specified a zip code.

With the location query string added in `find_by_asl`, we can finally bask in the glory of being able to find all the youngest female RailsSpace users within 250 miles of Caltech (Figure 11.10). Oops—well, that's pretty much what we expected. Fine—how about the old men (Figure 11.11)? Yup, that's the Caltech we know and love!

11.4.3 Location names

You may have noticed that in nonempty search results, the location is identified by zip code alone (Figure 11.11). In real life, we expect that many users would elect to type in their city and state as well, but—now that we have a geographical database—it would be nice to fill in those fields automatically based on zip code. That's the aim of this section.

Our first step is to polish the city name strings, which (as you can see from Figure 11.9) are currently ALL CAPS; we need a way to convert, for example, `LOS ANGELES` to `Los Angeles`. To do this, we'll add a couple of (very closely related) functions to the String class to capitalize each word in a space-separated string:

[15] See, for example, `http://www.w3schools.com/sql/sql_join.asp`, or do a web search for *SQL join* to get more information on the different types of joins.

Figure 11.10 All female RailsSpace users between the ages of 50 and 65 within 250 miles of Caltech.

Listing 11.26 lib/string.rb

```
class String
  .
  .
  .
  # Capitalize each word (space separated).
  def capitalize_each
    space = " "
    split(space).each{ |word| word.capitalize! }.join(space)
  end

  # Capitalize each word in place.
  def capitalize_each!
    replace capitalize_each
  end
end
```

Note in the second function `capitalize_each!` that we use the `replace` function, which is a special Ruby function to replace `self` with another object (thereby mutating it); in this case, the given string (`self`) gets replaced by the result of `capitalize_each`.

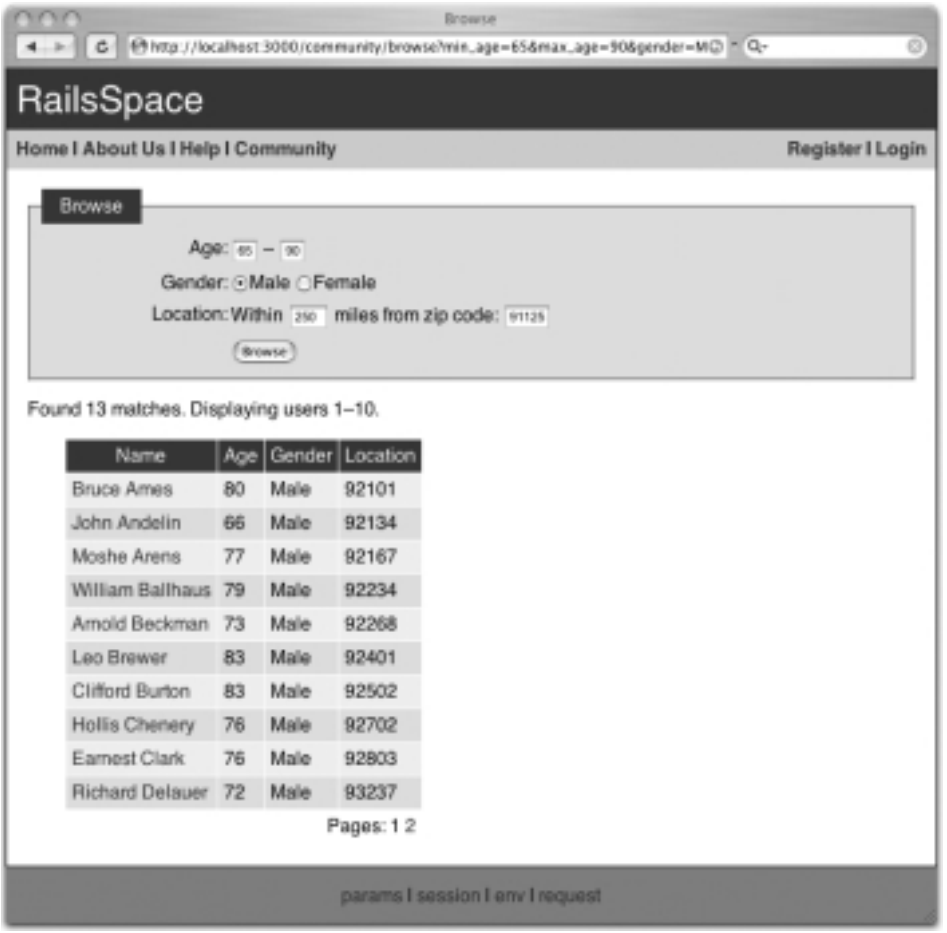

Figure 11.11 All male RailsSpace users between the ages of 65 and 90 within 250 miles of Caltech.

It's important to emphasize that the Ruby `capitalize!` method (on which `capitalize_each` relies) converts both `los` and `LOS` to `Los`, so that the all-caps city names will be properly converted:

```
> ruby script/console
Loading development environment.
>> "LOS ANGELES".capitalize_each
=> "Los Angeles"
```

After restarting the development webserver to load the changes to `string.rb`, we'll be ready to look up the city and state based on zip code and then format the city name appropriately:

Listing 11.27 app/models/spec.rb

```
# Return a sensibly formatted location string.
def location
  if not zip_code.blank? and (city.blank? or state.blank?)
    lookup = GeoDatum.find_by_zip_code(zip_code)
    if lookup
      self.city  = lookup.city.capitalize_each if city.blank?
      self.state = lookup.state if state.blank?
    end
  end
  [city, state, zip_code].join(" ")
end
```

We use the test `if lookup` since our database doesn't have city/state values for all zip codes. Note that we don't override the user-defined city and state if they're already present; we'll trust our users enough to give them the power to trump the data in our geographical database.[16]

Having city and state in addition to zip code really fleshes out the search results, as seen in Figure 11.12.

11.4.4 Adding browse validation

There's only one problem left with the RailsSpace browse page: Putting in invalid data breaks the form rather badly (Figure 11.13). This is the first time we've had to validate a form that wasn't simply the display for a model, so instead of using built-in model validations we have to do things (mostly) by hand.

The validations themselves are fairly simple. We want to verify that the maximum and minimum ages are valid integers, that the number of miles is a valid floating point number, and that the zip code is correctly formatted and exists in our database. We'll create a Spec model object, to which we will attach the errors, so that we can display them using `error_messages_for('spec')`.

It will be helpful when validating the input to have methods to detect invalid integers and floats. We'll add relevant methods to the Object class, taking advantage of Ruby's policy of raising an ArgumentError exception for a failed numerical conversion:

```
> irb
irb(main):001:0> Integer("foo")
ArgumentError: invalid value for Integer: "foo"
        from (irb):1:in 'Integer'
        from (irb):1
```

[16] One of the authors once lived in 90048, which shows up as West Hollywood in many databases but is actually located in the city of Los Angeles.

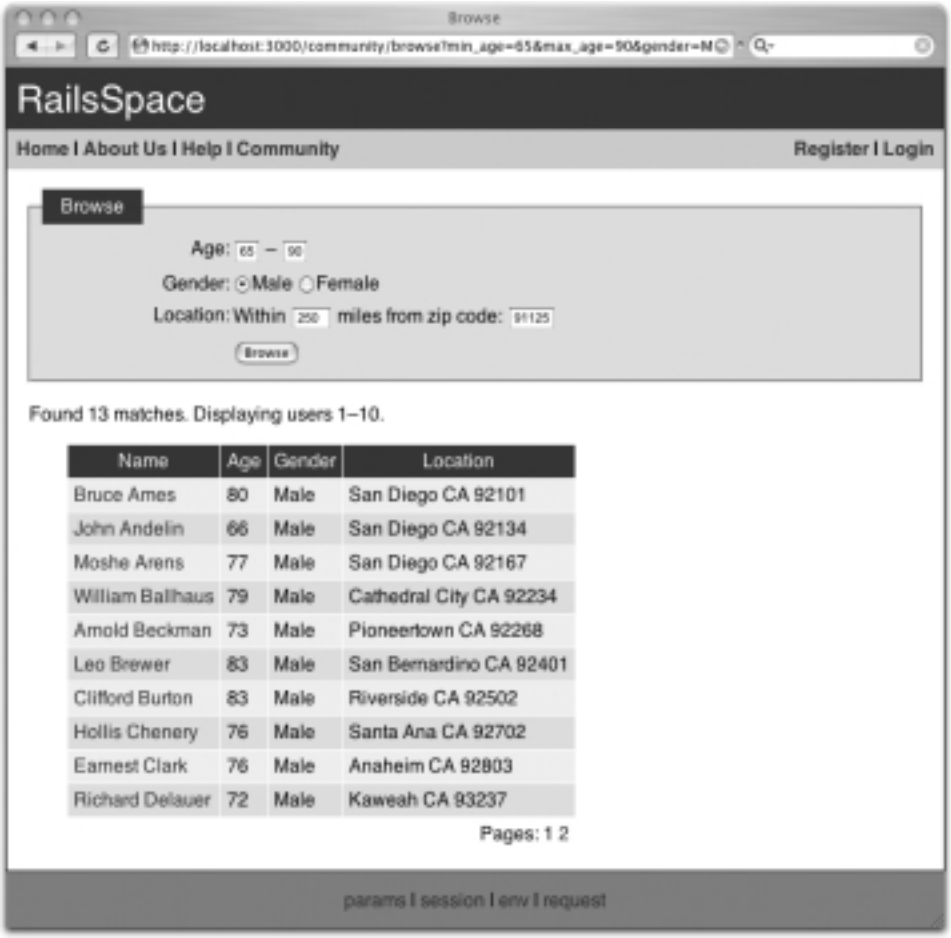

Figure 11.12 Browse results with city and state lookup.

```
irb(main):002:0> Float("bar")
ArgumentError: invalid value for Float(): "bar"
        from (irb):2:in 'Float'
        from (irb):2
```

This behavior suggests the following tests for valid ints and floats, using the same `begin...rescue` syntax we used in Section 11.1.5 to catch the Ferret exception:

Listing 11.28 lib/object.rb

```
class Object

  # Return true if the object can be converted to a valid integer.
```

Continues

Figure 11.13 Browsing for "foo" instead of integers.

```
  def valid_int?
    begin
      Integer(self)
      true
    rescue ArgumentError
      false
    end
  end

  # Return true if the object can be converted to a valid float.
  def valid_float?
    begin
      Float(self)
      true
    rescue ArgumentError
      false
    end
  end
end
```

By the way, we put these methods in the Object class (which is the base class for all Ruby objects) rather than in String because we want to be able to test `nil`; as it turns

out, `nil.valid_int?` is true (`Integer(nil) == 0`) while `nil.valid_float?` is false (`Float(nil)` raises an ArgumentError exception).

In order for these new functions to be loaded, we need to add a line to the Application helper:

Listing 11.29 app/helpers/application_helper.rb

```
module ApplicationHelper
  require 'string'
  require 'object'
  .
  .
  .
```

Then, restart the webserver so that the new Object functions will be included.

In order to catch the form entry errors, we'll define a function called `valid_input?` and then wrap `find_by_asl` and pagination inside `if valid_input?`:

Listing 11.30 app/views/controllers/community_controller.rb

```
def browse
  @title = "Browse"
  return if params[:commit].nil?
  if valid_input?
    specs = Spec.find_by_asl(params)
    @pages, @users = paginate(specs.collect { |spec| spec.user })
  end
end
```

Of course, we have to write `valid_input?`, which is reasonably long but is straightforward. As mentioned above, we'll first create a new spec, on which we will accumulate errors for display in the view. We then march through the different requirements for valid input, adding an error for each one that fails. The only mildly tricky part is the use of `@spec.valid?` to verify the zip code format; this just piggybacks on the zip code validation we already built. The full function appears as follows:

Listing 11.31 app/views/controllers/community_controller.rb

```
.
.
.
private

# Return true if the browse form input is valid, false otherwise.
```

Continues

```
def valid_input?
  @spec = Spec.new
  # Spec validation (with @spec.valid? below) will catch invalid zip codes.
  zip_code = params[:zip_code]
  @spec.zip_code = zip_code
  # There are a good number of zip codes for which we have no information.
  location = GeoDatum.find_by_zip_code(zip_code)
  if @spec.valid? and not zip_code.blank? and location.nil?
    @spec.errors.add(:zip_code, "does not exist in our database")
  end
  # The age strings should convert to valid integers.
  unless params[:min_age].valid_int? and params[:max_age].valid_int?
    @spec.errors.add("Age range")
  end
  # The zip code is necessary if miles are provided.
  miles = params[:miles]
  if miles and not zip_code
    @spec.errors.add(:zip_code, "can't be blank")
  end
  # The number of miles should convert to a valid float.
  unless miles.nil? or miles.valid_float?
    @spec.errors.add("Location radius")
  end
  # The input is valid iff the errors object is empty.
  @spec.errors.empty?
end
```

Note that a line such as `@spec.errors.add("Location radius")` simply leads to an error string of the form `"Location radius is invalid"`, while `@spec.errors.add(:zip_code, "can't be blank")` gives the error string `"Zip code can't be blank"`.

The code as it stands is already sufficient to protect our form from invalid input, but it would be inconsiderate not to tell our users what the problems are. Unfortunately, if we simply use the code

```
<%= error_messages_for('spec') %>
```

as we have in previous chapters, we'll get error messages like "2 errors prohibited this spec from being saved," which would be confusing in the context of our browse form—we're browsing for users, not trying to save a spec. The solution involves using the `sub` string method, which simply substitutes one string for another:[17]

```
> irb
irb(main):001:0> s = "foo bar baz"
```

[17] The closely related global substitution method `gsub` is also useful; where `sub` replaces only the first occurrence of a particular string, `gsub` replaces all of them.

```
=> "foo bar baz"
irb(main):002:0> s.sub("baz", "quux")
=> "foo bar quux"
```

Applying this idea to the browse view yields the following:

Listing 11.32 app/views/community/browse.rhtml

```
<%= error_messages_for('spec').sub('prohibited this spec from being saved',
                                   'occurred') %>
<%= render :partial => "browse_form" %>
<%= render :partial => "result_summary" %>
<%= render :partial => "user_table" %>
```

This way, we get errors of the form "3 errors occurred" (Figure 11.14), which makes a lot more sense.

11.4.5 The final community home page

Having built the index, search, and browse pages, we'll end by adding the browse partial to make the RailsSpace community page a one-stop shop for finding RailsSpace users (Figure 11.15):

Figure 11.14 Browse form with a nice description of the errors.

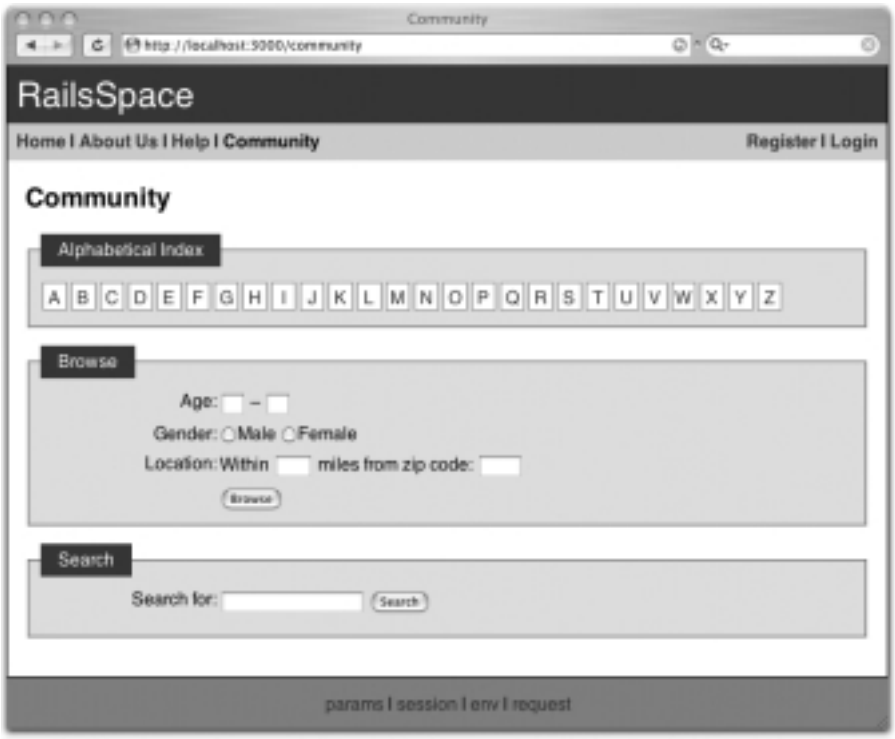

Figure 11.15 The final community page with index, browse, and search.

Listing 11.33 app/views/community/index.rhtml

```
<h2><%= @title %></h2>
<fieldset>
  <legend>Alphabetical Index</legend>
  <% @letters.each do |letter| %>
    <% letter_class = (letter == @initial) ? "letter_current" : "letter" %>
    <%= link_to(letter, { :action => "index", :id => letter },
                        :class => letter_class) %>
  <% end %>
  <br clear="all" />
</fieldset>

 <%= render :partial => "result_summary" %>
 <%= render :partial => "user_table" %>

<% if @initial.nil? %>
  <%= render :partial => "browse_form" %>
  <%= render :partial => "search_form" %>
<% end %>
```