CHAPTER 6

Developing a REST-Based Application

For this project, we're going to build a rails application that will provide a way for us to track our fitness goals and results. When we're finished, the final project should

- Allow us to maintain a list of our goals (such as weight loss or bicep size) and maintain records of our progress towards those goals over time.
- Allow us to maintain a record of when we worked out and what we did.
- Allow us to capture and track the progress of our workouts. We need to be able to capture data like how long it took us to jog three miles or how much we were able to lift using the bench press.
- Include an API for our application to open the door for us to make this data available elsewhere (such as on a personal blog).
- Provide a way to graph our progress over time. This will allow us to visually identify any problems in our workouts, such as a long period with no increase in our bench press weight. Seeing steady progress toward our goals is a great motivational asset as well.

REST-Based Development

As a means to accomplish these application goals, we're going to explore the new REST-based support that was added in Rails 1.2. It would be an understatement to say that RESTful development has been adopted by the Rails community big time, as evidenced by the overwhelming amount of discussions and posts that have been seen online since RESTful support was first announced during David Heinemeier Hansson's (DHH's) keynote address at the 2006 RailsConf entitled "A World of Resources" (available for download at `http://media.rubyonrails.org/presentations/worldofresources.pdf`). During that keynote, DHH introduced features of Rails designed to support the development of RESTful applications and challenged developers to embrace the constraints of RESTful development by viewing their controllers with a mindset of supporting CRUD operations (you should recall that we touched on these in Chapter 3).

So What Is REST?

From an academic standpoint, Representational State Transfer (REST) is a software architecture for distributed hypermedia systems. It refers to a collection of principles for how data is defined and accessed. It provides a simple way for external systems to access applications' data over HTTP without having to add an additional layer, such as SOAP. In web applications, REST is often used to describe a development style that provides a clean and unified interface that allows the same interface to serve multiple representations of the same data to various clients. In a REST-based system, we're often talking about interacting with resources. Examples of resources include such things as a Ruby object, a database result, and an image—essentially, a resource is anything that we might want to expose to our users to interact with.

Hopefully, that last paragraph didn't cause your eyes to glaze over too badly. If it did, don't worry, because from this point forward, we'll avoid the definitions and focus on what REST means for us. From a more practical view, when we're talking about building a RESTful application, we're talking about building a simple interface to our application where we're using the four core HTTP methods (`GET`, `POST`, `PUT`, and `DELETE`) to define the actions to perform on a resource rather than by placing method names within the URL as you can see in Table 6-1.

Table 6-1. *Comparison REST and Traditional URLs*

REST URL	Traditional URL
/exercises	/exercises/index
/exercises/1	/exercises/show/1
/exercises/new	/exercises/new
/exercises/1;edit	/exercises/edit/1
/exercises	/exercises/create
/exercises/1	/exercises/update/1
/exercises/1	/exercises/destroy/1

The goal of a REST design is to break things down into nouns and verbs. The verbs that we use in a RESTful application are the core HTTP methods `GET`, `POST`, `PUT`, and `DELETE`, while the nouns are the resources that we've made available. Those core HTTP methods have an obvious similarity to the CRUD database operations that you can see in Table 6-2.

Table 6-2. *Correlation of HTTP Methods to Database Operations*

Operation	Database	HTTP
Find	`SELECT`	`GET`
Create	`INSERT`	`POST`
Update	`UPDATE`	`PUT`
Destroy	`DELETE`	`DELETE`

The goal of REST is to get away from creating an endless series of method names that we might tack onto a URL to interact with an object such as `http://localhost/resource/add_resource` or `http://localhost/resource/list`. Instead, we want to move to using our verbs to interact with our nouns so that those same requests become a POST or GET request to `http://localhost/resources`.

The Value of REST

RESTful design provides a number of advantages; in his keynote, DHH brought up three main advantages for Rails developers:

Consistency: Designing a RESTful application provides us with a greater level of consistency—not just in our URLs but also in our controllers. Controllers built to the RESTful ideals will always have the same seven methods (`index`, `new`, `create`, `show`, `edit`, `update`, and `destroy`).

Simplicity: RESTful design takes away a lot of the questions about where things go; it lets every object focus primarily on its CRUD operations by implementing the seven methods discussed in the previous point. Simplifying these decisions effectively simplifies our design. There are already success stories out there, such as that of Scott Raymond who refactored IconBuffet.com to RESTful principles and discovered that he had reduced the number of actions in his application by 25 percent.

Discoverability: This goes hand in hand with consistency—as more and more developers embrace a common set of constraints, it eases integration between applications.

Our First Resource

Enough theory—let's kick off our project and see how Rails makes building a RESTful application a snap as we explore building our first resource. Open your development directory, and create a new project named exercisr using the instructions from Chapter 2. With our project created, let's take a brief look at some of the tools within Rails that we'll use to support building a RESTful application.

RESTful Tools

DHH, the creator of Ruby on Rails, has talked a number of times about how the core team tries to add syntactic sugar around certain programming styles or approaches to encourage their use, and that's highly apparent in the Rails support for building REST applications. Let's take a quick look at the three core features of Rails that support RESTful development.

map.resources

The first tool that we want to take a look at is a method that has been added for our routes. Traditionally we've had two methods that we could use within our routes. The oldest of these was `map.connect`, which is used to build our general `/:controller/:action/:id` style of routes in a manner like this:

```
map.connect '', :controller => 'home', :action => 'welcome'
map.connect '/post/:id', :controller => 'post', :action => 'show'
```

```
map.connect '/weather/:year/:month/:date, :controller => 'weather',
                                                    :action => 'archive'
```

While the `map.connect` method was fine for awhile, it did cause Rails developers to be a bit too verbose a bit too often, when they wanted to refer to a specific route. If we wanted to create a link to the preceding routes, the correct way would have been

```
link_to 'Home', :controller => 'home', :action => 'welcome'
link_to 'Show Post', :controller => 'post', :action => 'show', :id => @post
link_to 'Weather Last Christmas', :controller => 'weather', :action => 'archive',
                                    :year => '2006' :month => '12', :date => '25'
```

You can see how quickly it would become bothersome to constantly generate links by manually typing the controller, actions, and ID parameters each time, and you can imagine how quickly this was increasing the noise ratio within our view files. Named routes were added to our Rails arsenal to ease this pain for Rails developers. Building a named route is almost identical to building a regular route—except that we replace the connect method with a custom name of our own choosing for the route. So to convert the regular routes you saw previously, you might write them like this:

```
map.home '', :controller => 'home', :action => 'welcome'
map.post '/post/:id', :controller => 'post', :action => 'show'
map.weather_archive '/weather/:year/:month/:date, :controller => 'weather',
                                                          :action => 'archive'
```

While this may seem like just a minor change, it makes a big difference for us. When we create a named route, Rails provides us with two new URL methods that make our lives much easier: `{named_route}_path` and `{named_route}.url`. So our previous `map.home` route would generate these URL methods as `home_path` and `home_url`, and they would generate the URLs for this route. The only difference between these two methods is that `home_url` will generate a fully qualified link including host and post (i.e., `http://locahost:3000/`) while `home_path` will only generate the relative path (i.e., `/`). So armed with our named routes, we could build links to those routes like this:

```
link_to 'Home', home_path
link_to 'Show Post', post_path(@post)
link_to 'Weather Last Christmas', weather_archive_path('2006', '12', '25')
```

While named routes are an incredibly powerful tool, they were still a bit underpowered for the needs of building RESTful routes, as we would be forced to build numerous named routes to support the seven CRUD style methods for each resource that we wanted to support. Fortunately, a new routing method by the name of `map.resources` was added to Rails; it's like scaffolding for RESTful routes. Let's take a look at a `map.resources` example by creating a resource named exercises. So imagine that we added a route such as this:

```
map.resources :exercises
```

Running `map.resources :exercises` generated a whole mess of dynamically generated routes that map to our RESTful actions, as well as a full plate of useful URL methods (the same as a `named_route` would), which you can see in Table 6-3.

Table 6-3. *map.resources Generated Routes*

Action	HTTP Method	URL	Generated URL Method
Index	GET	/exercises	exercises_path
Show	GET	/exercises/1	exercise_path(:id)
New	GET	/exercises/new	new_exercise_path
Edit	GET	/exercises/1;edit	edit_exercise_path(:id)
Create	POST	/exercises	exercises_path
Update	PUT	/exercises/1	exercise_path(:id)
Destroy	DELETE	/exercises/1	exercise_path(:id)

With one `map.resources` call, we've generated a full suite of RESTful routing actions for our application and saved ourselves a tremendous amount of typing. In fact, to do it by hand would have required us to create a routes configuration like this:

```
ActionController::Routing::Routes.draw do |map|
  # map.resources :exercises
  # gives us all this
  map.exercises 'exercises', :action => 'index', :conditions => {:method => :get}
  map.connect 'exercises', :action => 'create', :conditions => {:method => :post}
  map.formatted_exercise 'exercises.:format', :action => 'index',
                                                :conditions => {:method => :get}
  map.exercise 'exercises/:id', :action => 'edit', :conditions => {:method => :get}
  map.connect 'exercises/:id', :action => 'update', :conditions => {:method => :put}
  map.connect 'exercises/:id', :action => 'destroy',
                                       :conditions => {:method => :delete}
  map.formatted_exercise 'exercises/:id.:format', :action => 'edit',
                                                :conditions => {:method => :get}
  map.connect 'exercises/:id.:format', :action => 'update',
                                               :conditions => {:method => :put}
  map.connect 'exercises/:id.:format', :action => 'destroy',
                                              :conditions => {:method => :delete}
  map.new_exercise 'exercises/new', :action => 'new',
                                                  :conditions => {:method => :get}
  map.formatted_new_exercise 'exercises/new.:format', :action => 'new',
                                                :conditions => {:method => :get}
  map.edit_exercise 'exercise/:id;edit', :action => 'edit',
                                                :conditions => {:method => :get}
  map.edit_exercise 'exercise/:id.:format;edit', :action => 'edit',
                                                :conditions => {:method => :get}
end
```

respond_to

The next tool within Rails that we'll use to support RESTful applications is the `respond_to` method within our controllers. Within a normal non-RESTful method, we might have an action that looks like this:

```
def index
   @exercises = Exercise.find(:all)
end
```

By default, this method will automatically render the first template file named `index` that it finds in its corresponding folder (in this example, `index.rhtml`). But one of our goals with REST is to be able to provide different variations of the same data from a single resource. For that, we'll modify the method to use the `respond_to` method:

```
def index
   @exercises = Exercise.find(:all)

   respond_to do |format|
      format.html
      format.xml { render :xml => @exercises.to_xml }
   end
end
```

In essence what this does is evaluate the desired response format that was sent with the request in the HTTP accept header and return the corresponding template. Therefore, a normal web request will be served back HTML, while an XML request would be served a list of exercises in XML format.

scaffold_resource

The last tool that we'll explore here is the new scaffolding generator for resource based routing named `scaffold_resource`. Over the last year or so, using the Rails scaffolding had fallen out of favor within the Rails community; and it was considered good for screencasts but not a tool that a professional Rails developer would use. Well, that changed with the release of the `scaffold_resource` generator, as `scaffold_resource` provides a wealth of functionality that truly speeds up your development in a clean and elegant manner. To get an understanding of how the new `scaffold_resource` works, you can run the command with no parameters to view some helpful information about how to use it:

```
ruby script/generate scaffold_resource
```

```
Usage: script/generate scaffold_resource ModelName [field:type, field:type]
. . .
Description:
    The scaffold resource generator creates a model, a controller, and a set of
templates that's ready to use as the starting point for your REST-like,
resource-oriented application. This basically means that it follows a set of
conventions to exploit the full set of HTTP verbs (GET/POST/PUT/DELETE) and is
prepared for multi-client access (like one view for HTML, one for an XML API, one
for ATOM, etc). Everything comes with sample unit and functional tests as well.

    The generator takes the name of the model as its first argument. This model
name is then pluralized to get the controller name. So "scaffold_resource post" will

generate a Post model and a PostsController and will be intended for URLs like
/posts and /posts/45.

    As additional parameters, the generator will take attribute pairs described
by name and type. These attributes will be used to prepopulate the migration to
create the table for the model and to give you a set of templates for the view. For
example, "scaffold_resource post title:string created_on:date body:text
published:boolean" will give you a model with those four attributes, forms to create
 and edit those models from, and an index that'll list them all.

    You don't have to think up all attributes up front, but it's a good idea of
adding just the baseline of what's needed to start really working with the resource.

    Once the generator has run, you'll need to add a declaration to your config/
routes.rb file to hook up the rules that'll point URLs to this new resource. If you
 create a resource like "scaffold_resource post", you'll need to add
"map.resources :posts" (notice the plural form) in the routes file. Then your new
resource is accessible from /posts.

Examples:
   ./script/generate scaffold_resource post # no attributes, view will be anemic
   ./script/generate scaffold_resource post title:string created_on:date ➥
body:text published:boolean
   ./script/generate scaffold_resource purchase order_id:integer ➥
created_at:datetime amount:decimal
```

Reading through that output should get you pretty excited about just how powerful the `scaffold_resource` generator is. From a single command, we can build out a full controller that's prebuilt to not only support the full line of CRUD operations but also HTTP verbs that we need for a REST application. The scaffolding will build our model and a complete set of generic view templates—yet it goes beyond the normal scaffolding for those items by also allowing us to pass in the database elements to be used in our migration to create the tables in the database. If it feels like we've just turbocharged our speed of development, it's because we have. Let's see it in action as we use `scaffold_resource` to build the first resource for our application.

Building the Exercise Resource

In our exercise project, whenever we record a workout, we'll be recording a series of exercises that we did. So, as we think about the attributes of an exercise, we recognize that what we really need here is just an object to identify each exercise that we could perform. Examples of exercises include things such as bench press, leg press, and biceps curl. We'll want to expose this list of possible exercises to an end user, so we'll build it as a resource using the new `scaffold_resource` generator.

At its core, an exercise resource should be a fairly simple thing—it will have to have a name, and a user that it's associated with. Knowing that, we can build our exercise resource by running this command:

```
ruby script/generate scaffold_resource Exercise name:string user_id:integer
```

```
  exists  app/models/
  exists  app/controllers/
  exists  app/helpers/
  create  app/views/exercises
  exists  test/functional/
  exists  test/unit/
  create  app/views/exercises/index.rhtml
  create  app/views/exercises/show.rhtml
  create  app/views/exercises/new.rhtml
  create  app/views/exercises/edit.rhtml
  create  app/views/layouts/exercises.rhtml
  create  public/stylesheets/scaffold.css
  create  app/models/exercise.rb
  create  app/controllers/exercises_controller.rb
  create  test/functional/exercises_controller_test.rb
  create  app/helpers/exercises_helper.rb
  create  test/unit/exercise_test.rb
  create  test/fixtures/exercises.yml
  exists  db/migrate
  create  db/migrate/001_create_exercises.rb
  route  map.resources :exercises
```

Let's take a moment to examine a few important things that the `scaffold_resource` generator has added to our project.

First off, the new `scaffold_resource` generator added a `map.resources` line to our routes configuration (`/config/routes.rb`), which generated all the URL methods that we discussed previously in Table 6-3:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :exercises
end
```

Second, the generator took the extra parameters that we passed it to prepopulate our database migration (`/db/migrate/001_create_exercises.rb`):

```
class CreateExercises < ActiveRecord::Migration
  def self.up
    create_table :exercises do |t|
      t.column :name, :string
      t.column :user_id, :integer
    end
  end

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

When we created our exercise resource via the `scaffold_resource` generator, the generator also built a standard REST-based controller for us (we'll need to make some modifications to the generated controller code before it will work in our application). The generated controllers all have the same seven methods (`index`, `show`, `new`, `edit`, `create`, `update`, and `destroy`). Let's take a quick look at the index method in `exercises_controller` (`/app/controllers/exercies_controller.rb`):

```
  # GET /exercises
  # GET /exercises.xml
  def index
    @exercises = Exercise.find(:all)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @exercises.to_xml }
    end
  end
```

At the top of each of these generated methods, Rails added a comment to provide you with examples of what URLs could be used to access this method. After the comments, you'll see a standard `find` on the exercise model populating an instance variable named `@exercises`.

The `respond_to` block, though, is a pretty exciting thing, as it allows us to support multiple content requests from a single method. Rails has supported these `respond_to` blocks since Rails 1.1 came out in the spring of 2006, but it's within the context of the new RESTful routing that I think they really come alive.

One of the big problems with previous versions of the `respond_to` blocks was that they responded solely based on what was sent in the request header as the `Accept` content type. So a client that sent `Accept: text/html` was given an HTML template, and a client that sent `Accept: text/xml` was sent an XML template.

The trouble would come in, though, when a client sent an incorrect `Accept` request. RSS is a great example of that, as technically the client wants to pull down a specially formatted XML feed, yet many RSS requests come in as `Accept: text/html`.

Rails 1.2 fixed that by changing the way the routing works to also evaluate the file format requested—it evaluates the file extensions within the URL and gives those a higher priority over what's in the `Accept` header. In this way, a client that makes a `GET` request to `/exercises.xml` will be served the XML template even if the request came with `Accept: text/html` in the header. Nifty!

Let's take a quick glance at the full exercises controller that `scaffold_resource` generated for us:

```
class ExercisesController < ApplicationController
  # GET /exercises
  # GET /exercises.xml
  def index
    @exercises = Exercise.find(:all)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @exercises.to_xml }
    end
  end

  # GET /exercises/1
  # GET /exercises/1.xml
  def show
    @exercise = Exercise.find(params[:id])

    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @exercise.to_xml }
    end
  end

  # GET /exercises/new
  def new
    @exercise = Exercise.new
  end

  # GET /exercises/1;edit
  def edit
    @exercise = Exercise.find(params[:id])
  end
```

```
  # POST /exercises
  # POST /exercises.xml
  def create
    @exercise = Exercise.new(params[:exercise])

    respond_to do |format|
      if @exercise.save
        flash[:notice] = 'Exercise was successfully created.'
        format.html { redirect_to exercise_url(@exercise) }
        format.xml  { head :created, :location => exercise_url(@exercise) }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @exercise.errors.to_xml }
      end
    end
  end

  # PUT /exercises/1
  # PUT /exercises/1.xml
  def update
    @exercise = Exercise.find(params[:id])

    respond_to do |format|
      if @exercise.update_attributes(params[:exercise])
        flash[:notice] = 'Exercise was successfully updated.'
        format.html { redirect_to exercise_url(@exercise) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @exercise.errors.to_xml }
      end
    end
  end

  # DELETE /exercises/1
  # DELETE /exercises/1.xml
  def destroy
    @exercise = Exercise.find(params[:id])
    @exercise.destroy

    respond_to do |format|
      format.html { redirect_to exercises_url }
      format.xml  { head :ok }
    end
  end
end
```

Not only did the scaffolding generate a RESTful controller for us but it also built the full suite of associated templates for this controller. Extra nice is the fact that each of the templates also takes full advantage of the URL helpers, such as this one that was generated for the new method in `/app/views/exercises/new.rhtml`:

```
<h1>New exercise</h1>

<%= error_messages_for :exercise %>

<% form_for(:exercise, :url => exercises_path) do |f| %>
  <p>
    <b>Name</b><br />
    <%= f.text_field :name %>
  </p>

  <p>
    <b>User</b><br />
    <%= f.text_field :user_id %>
  </p>

  <p>
    <%= submit_tag "Create" %>
  </p>
<% end %>

<%= link_to 'Back', exercises_path %>
```

As you can see, the new `scaffold_resource` generator can be a huge timesaver for kick-starting a RESTful application. Now, let's continue our project development by adding in an authentication system.

■ **Note** The only thing that I don't like about the `scaffold_resource` generator is that it also creates a corresponding layout file in `/app/views/layouts`. Since I prefer to create a global layout file named `application.rthml`, this scaffold generated layout causes unnecessary issues. Go ahead and remove the generated `exercises.rhtml` file.

Adding RESTful Authentication

If experience has taught us anything by now, it's that if we build anything remotely useful, someone else is going to want to use it as well. We'll anticipate that our friends are also going to want to use the application by building in multiuser support from the get-go. Fortunately, Rick Olsen has made our job much easier, as he has already adapted his popular `acts_as_authenticated` plug-in to a REST-based implementation named `restful_authentication`. Using this plug-in will

allow us to jump start our application with a full multiuser login and authentication system without sacrificing any of our RESTful ideals.

To install `restful_authentication`, open a command prompt and run the following command:

```
ruby script/plugin install http://svn.techno-weenie.net/projects/plugins/ ➥
restful_authentication
```

```
./restful_authentication/README
./restful_authentication/Rakefile
./restful_authentication/generators/authenticated/USAGE
./restful_authentication/generators/authenticated/authenticated_generator.rb
./restful_authentication/generators/authenticated/templates/activation.rhtml
./restful_authentication/generators/authenticated/templates/authenticated_system.rb
./restful_authentication/generators/authenticated/templates/ ➥
authenticated_test_helper.rb
./restful_authentication/generators/authenticated/templates/controller.rb
./restful_authentication/generators/authenticated/templates/fixtures.yml
./restful_authentication/generators/authenticated/templates/functional_test.rb
./restful_authentication/generators/authenticated/templates/helper.rb
./restful_authentication/generators/authenticated/templates/login.rhtml
./restful_authentication/generators/authenticated/templates/migration.rb
./restful_authentication/generators/authenticated/templates/model.rb
./restful_authentication/generators/authenticated/templates/model_controller.rb
./restful_authentication/generators/authenticated/templates/model_functional_test.rb
./restful_authentication/generators/authenticated/templates/model_helper.rb
./restful_authentication/generators/authenticated/templates/notifier.rb
./restful_authentication/generators/authenticated/templates/notifier_test.rb
./restful_authentication/generators/authenticated/templates/observer.rb
./restful_authentication/generators/authenticated/templates/signup.rhtml
./restful_authentication/generators/authenticated/templates/ ➥
signup_notification.rhtml
./restful_authentication/generators/authenticated/templates/unit_test.rb
./restful_authentication/install.rb
Restful Authentication Generator
====

This is a basic restful authentication generator for rails, taken from acts as
authenticated.  Currently it requires Rails 1.2 (or edge).

To use:

  ./script/generate authenticated user sessions --include-activation

The first parameter specifies the model that gets created in signup (typically a
user or account model).  A model with migration is created, as well as a basic
controller with the create method.
```

```
The second parameter specifies the sessions controller name.  This is the controller
 that handles the actual login/logout function on the site.

The third parameter (--include-activation) generates the code for a ActionMailer and

its respective Activation Code through email.

You can pass --skip-migration to skip the user migration.

From here, you will need to add the resource routes in config/routes.rb.

  map.resources :users, :sessions

Also, add an observer to config/environment.rb if you chose the --include-activation
 option
  config.active_record.observers = :user_observer # or whatever you named your model
```

In the same manner that `acts_as_authenticated` did in our MonkeyTasks projects, the Restful Authentication plug-in merely installs a generator that we use to create our authentication system. Documentation for this generator should have been output during the plug-in installation, but you can also find it in the plug-in's `readme` file (`/vendor/plugins/restful_authentication/README`). We're not going to bother with adding the mailer and activation code functionality, since we already built that functionality in the MonkeyTasks project; plus, we're designing this application for a smaller user base (i.e., just our friends and family). With that said, let's use the generator to build our authentication system! Back at the command prompt in the root our application, type the following command:

```
ruby script/generate authenticated user sessions
```

```
Don't forget to:

  - add restful routes in config/routes.rb
    map.resources :users, :sessions
    map.activate '/activate/:activation_code', :controller => 'users',
                                               :action => 'activate'

Try these for some familiar login URLs if you like:

  map.signup '/signup', :controller => 'users', :action => 'new'
  map.login  '/login', :controller => 'sessions', :action => 'new'
  map.logout '/logout', :controller => 'sessions', :action => 'destroy'

-------------------------------------------------------------------------
```

```
      exists  app/models/
      exists  app/controllers/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/sessions
      create  app/views/user_notifier
      exists  test/functional/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/users
      exists  test/functional/
      exists  test/unit/
      create  app/models/user.rb
      create  app/controllers/sessions_controller.rb
      create  app/controllers/users_controller.rb
      create  lib/authenticated_system.rb
      create  lib/authenticated_test_helper.rb
      create  test/functional/sessions_controller_test.rb
      create  test/functional/users_controller_test.rb
      create  app/helpers/sessions_helper.rb
      create  app/helpers/users_helper.rb
      create  test/unit/user_test.rb
      create  test/fixtures/users.yml
      create  app/views/sessions/new.rhtml
      create  app/views/users/new.rhtml
      create  db/migrate
      create  db/migrate/002_create_users.rb
```

The generator has added two new controllers to our application: a sessions controller and a users controller. It's also added a user model; associated views, helpers, and tests; and the authentication library to our project. The generator was even nice enough to remind us to add the necessary resource routes to our routes configuration. We'll take advantage of that reminder and go ahead and add those routes now. Edit your `routes.rb` file to look like this:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :exercises
  map.home '', :controller => 'sessions', :action => 'new'
  map.resources :users, :sessions
  map.signup '/signup', :controller => 'users', :action => 'new'
  map.login  '/login', :controller => 'sessions', :action => 'new'
  map.logout '/logout', :controller => 'sessions', :action => 'destroy'
end
```

We now need to set up our application to use the authentication system by default. When the generator ran, it added the necessary commands into the two controllers that it generated (sessions and users). But since we want all of the controllers in our application to use the authentication system, we need to move those commands out of those controllers and into

the application controller (so that all controllers will inherit it). Open our three controllers (`users_controller.rb`, `sessions_controller.rb`, and `application_controller.rb`) in `/app/controllers`; remove the following lines out of the sessions and users controllers and add them into the application controller:

```
  # Be sure to include AuthenticationSystem in Application Controller instead
  include AuthenticatedSystem
  # If you want "remember me" functionality, add this before_filter to ➥
     Application Controller
  before_filter :login_from_cookie
```

Afterward, our application controller (`/app/controllers/application.rb`) should look like this:

```
class ApplicationController < ActionController::Base
  session :session_key => '_exercisr_session_id'
  include AuthenticatedSystem
end
```

Our application controller currently has two lines. The first line is automatically generated and is simply used to create the session keys that our application will use for any sessions it creates. The second line is the one that we just added, and it includes the authenticated system library from our `/lib` folder.

If we had wanted to enable a "remember me" level of functionality (i.e., setting a cookie in the user's browser that would enable them to be logged into the application without having to reenter their username and password each time), we could have also placed the `before_filter :login_from_cookie` line into this controller like we did for MonkeyTasks.

Meanwhile, our users controller (`/app/controllers/users_controller.rb`) should look like this:

```
class UsersController < ApplicationController
  # render new.rhtml
  def new
  end

  def create
    @user = User.new(params[:user])
    @user.save!
    self.current_user = @user
    redirect_back_or_default('/')
    flash[:notice] = "Thanks for signing up!"
  rescue ActiveRecord::RecordInvalid
    render :action => 'new'
  end
end
```

The users controller allows users to be added to our application and contains two methods. The `new` method is where our `/signup` route directs to and is used to display the form to create a new user account, which you can see in Figure 6-1.

Figure 6-1. *The default signup form from Restful Authentication*

This signup form will POST to the `create` method in our users controller, which will create a new user from the form submission.

Finally, we have our sessions controller (`/app/controllers/sessions_controller.rb`), which handles the logging in and logging out functionality of our site:

```
class SessionsController < ApplicationController
  # render new.rhtml
  def new
  end

  def create
    self.current_user = User.authenticate(params[:login], params[:password])
    if logged_in?
      if params[:remember_me] == "1"
        self.current_user.remember_me
        cookies[:auth_token] = { :value => self.current_user.remember_token ,
                            :expires => self.current_user.remember_token_expires_at }
      end
      redirect_back_or_default('/')
      flash[:notice] = "Logged in successfully"
    else
      render :action => 'new'
    end
  end

  def destroy
    self.current_user.forget_me if logged_in?
    cookies.delete :auth_token
    reset_session
    flash[:notice] = "You have been logged out."
    redirect_back_or_default('/')
  end
end
```

The sessions controller contains three methods.

- `new`: This method is where our `/login` route is pointed. It displays the basic login form that a user would submit to log in to our application. That login form submits to the `create` method in this controller.
- `create`: This method is used to actually log in a user by creating a new session once the user has been authenticated.
- `destroy`: Finally, we have the `destroy` method, which removes our authenticated session, effectively logging out the user. This is where the `/logout` route is pointed.

Migrations

Finally, let's take a quick glance at the migration file that the generator created. Open `002_create_users.rb` in `/db/migrate`:

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table "users", :force => true do |t|
      t.column :login,                     :string
      t.column :email,                     :string
      t.column :crypted_password,          :string, :limit => 40
      t.column :salt,                      :string, :limit => 40
      t.column :created_at,                :datetime
      t.column :updated_at,                :datetime
      t.column :remember_token,            :string
      t.column :remember_token_expires_at, :datetime
    end
  end

  def self.down
    drop_table "users"
  end
end
```

If we wanted to add any custom fields to our users model, such as capturing the first name, last name, or address, we could add them in here. However, for our application, we'll be just fine with the defaults. Go ahead and close this file, and let's run this migration to add the exercises and users tables to our database:

```
rake db:migrate
```

```
== CreateExercises: migrating ================================================
-- create_table(:exercises)
   -> 0.0780s
== CreateExercises: migrated (0.0780s) =======================================

== CreateUsers: migrating ====================================================
-- create_table("users", {:force=>true})
   -> 0.0620s
== CreateUsers: migrated (0.0620s) ===========================================
```

Fire up your web server (probably with the `mongrel rails_start` command) and load our application (it should be available at `http://localhost:3000`). You should be greeted by the login form shown in Figure 6-2.

Figure 6-2. *The default login form created by Restful Authentication*

Refining the Look

Let's go ahead and create a layout template to get the visuals of our application more in line with the original goal as we build out our functionality. You can download the style sheets and images from the code archive. We need to create a new layout file named `application.rhtml` in `/app/views/layouts` and place the following content in it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <meta http-equiv="Content-type" content="text/html; charset=utf-8">
      <title><%= @title || "Exercisr" %></title>
      <link rel="stylesheet" type="text/css" href="http://yui.yahooapis.com ➥
/2.2.2/build/reset-fonts-grids/reset-fonts-grids.css">
      <%= stylesheet_link_tag 'styles' %>
      <%= javascript_include_tag :defaults %>
   </head>
```

```
    <body>
      <div id="doc2" class="yui-t2">
        <div id="hd" class="box grad blue">
          <%= image_tag 'grad_black.png' %>
          <h1 id="masthead"><%= link_to "Exercisr", home_path %></h1>
        </div>

        <div id="bd">
          <div id="yui-main">
            <div class="yui-b">
              <%= yield %>
            </div>
          </div>
          <% if logged_in? %>
          <div class="yui-b sidebar">
            <ul>
              <li><%= link_to 'Exercises', exercises_path %></li>
              <li><%# link_to 'Workouts', workouts_path %></li>
              <li><%# link_to 'Goals', goals_path %></li>
              <li><%= link_to 'Logout', logout_path %></li>
            </ul>
          </div>
          <% end %>
        </div>

        <div id="ft" class="box grad blue"><%= image_tag 'grad_white.png' %></div>
      </div>
    </body>
</html>
```

Assuming that you still have a Mongrel instance running our application, reload the application in a web browser, and you should be treated with something like the login form shown in Figure 6-3, which is a little easier on the eyes (or at least good enough until we can afford to get a decent graphic artist to design something nicer).

Figure 6-3. *The login form with our layout and style sheet applied*

■ **Note** You may have noticed that some of the links in our sidebar are placed within <%# %> tags instead of the normal <%= %> tags. These tags essentially comment out the code contained within them so that it isn't executed. I did this because, otherwise, the application would break if we tried to execute those lines, as we cheated a little by adding links to resources that we haven't built yet. But don't worry; we'll be building those resources soon.

Creating a New User

We're at a good point to go ahead and create our first user account within the system. Before we do that though, we'll need to make one minor change to the way that the Restful Authentication plug-in works. If you open the sessions controller (`/app/controllers/session_controller.rb`), you'll see that there are three methods in this controller (though only two with code). Currently, both the `create` and `destroy` methods utilize a method from the Authenticated System library (`/lib/authenticated_system.rb`) named `redirect_back_or_default`. This method is a great tool for user friendliness, as it will return users to the original page that they requested once they have logged in. So if a user bookmarked a page in our application and tried to access it while not logged in, that user would first be directed to the login page; however, after logging in, this method would send the user back to the originally requested page. That's a wonderful user experience, but to make it work, we need to change the parameter that's being passed in this method. Currently, the method is sending users back to the login form by default (even after they're logged in), so let's change that to a basic welcome page. To do that, we'll first need to build a good starting page.

Creating a Home Page

If this was an application that we wanted to push out professionally, we'd want to build a nice interactive welcome page that provided users with sample data, tutorials, and so on. However, since our application is really just for our own use and maybe a few friends, we can make do with a static page that gives the user a generic welcome to the site at login. To accomplish this, we'll simply create an additional method and template within our sessions template. This is a little bit of a kludge, as it's not really the session controller's responsibility to present a welcome template to the user, but considering the simple needs of our application, it's one that we can live with. If we were going to have more than a single informational page or wanted to add more functionality to these pages, we would want to create a new controller to mange them.

Let's add a new `welcome` method within our sessions controller (`/app/controllers/session_controller.rb`):

```
def welcome
end
```

And we'll also create the associated template (`/app/views/sessions/welcome.rhtml`) for this method to display the following welcome text:

```
<h1>Welcome to Exercisr</h1>
<h3>A RESTful place to keep track of your workouts</h3>
```

■ **Note** Technically, we didn't have to create the `welcome` method within this controller, since there was no code that needed to be executed within that method. However, I feel that it's a good practice to always include a method for any template that you create in an application to avoid confusion when looking at the code at some point in the future.

With our welcome template defined now, let's add a named route to our routes configuration (`/config/routes.rb`) to access it (don't forget that the order of routes in this file is important):

```
ActionController::Routing::Routes.draw do |map|
  map.resources :exercises

  map.home '', :controller => 'sessions', :action => 'new'
  map.resources :users, :sessions
  map.welcome '/welcome', :controller => 'sessions', :action => 'welcome'
  map.signup '/signup', :controller => 'users', :action => 'new'
  map.login '/login', :controller => 'sessions', :action => 'new'
  map.logout '/logout', :controller => 'sessions', :action => 'destroy'
end
```

Finally, with our new named route to our welcome page, we can modify the default destinations of our methods in the session controller (`/app/controllers/session_controller.rb`) by changing the `create` method to go to our new welcome view and the `destroy` method to redirect back to the login page:

```
class SessionsController < ApplicationController

  def welcome
  end

  def new
  end

  def create
    self.current_user = User.authenticate(params[:login], params[:password])
    if logged_in?
      if params[:remember_me] == "1"
        self.current_user.remember_me
        cookies[:auth_token] = { :value => self.current_user.remember_token ,
                      :expires => self.current_user.remember_token_expires_at }
      end
      redirect_back_or_default(welcome_path)
      flash[:notice] = "Logged in successfully"
```

```
    else
      render :action => 'new'
    end
  end

  def destroy
    self.current_user.forget_me if logged_in?
    cookies.delete :auth_token
    reset_session
    flash[:notice] = "You have been logged out."
    redirect_back_or_default(login_path)
  end
end
```

The users controller (`/app/controllers/user_controller.rb`) also utilizes `redirect_back_or_default` in the `create` method when a user first signs up, so we'll need to also modify its default page:

```
def create
    @user = User.new(params[:user])
    @user.save!
    self.current_user = @user
    redirect_back_or_default(welcome_path)
    flash[:notice] = "Thanks for signing up!"
  rescue ActiveRecord::RecordInvalid
    render :action => 'new'
  end
```

With our code changes in place, you can go to `localhost:3000/signup` and create a new user account; afterward, you should be directed to our new welcome template, which you can see in Figure 6-4.

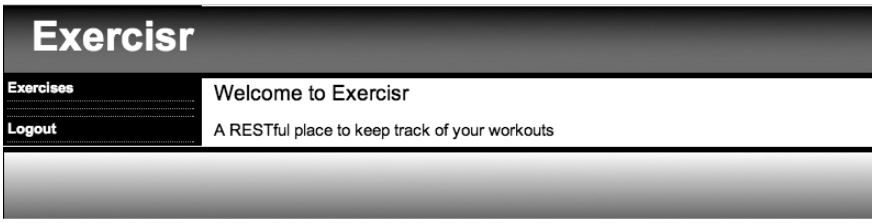

Figure 6-4. *Our welcome template*

Completing the Exercise Resource

If we're going to allow our friends and family to also have access to our Exercisr application, we'll need to provide some means to keep users' data separated. For our exercises resource, we only want each user to be able to view and edit his own list of exercises.

Building the Model Associations

The first step in maintaining separation among each users list of exercises is to build the associations between a user and an exercise. So within the user model (`/app/models/user.rb`) we'll add the following association:

```
has_many :exercises, :dependent => :destroy, :order => 'name asc'
```

Next, within our exercises model (`/app/models/exercise.rb`), we'll add the reciprocal association back to our user model:

```
belongs_to :user
```

With those associations added, we can now make a call to `current_user.exercises` within our controllers to pull back a list of the currently logged in user's exercises. However, before we start modifying our controllers to utilize that functionality—as long as we're already in our exercise model—let's go ahead and add some important validations to our model. Afterward, our exercise model will look like this:

```
class Exercise < ActiveRecord::Base
  belongs_to :user
  validates_presence_of :name
  validates_uniqueness_of :name, :scope => :user_id
end
```

For an exercise to be valid, we're going to require that it has a name. Secondly, we also want to ensure that a user doesn't submit the same exercise twice (for example, it would be confusing to allow a user to have bench press in her list twice). That uniqueness, though, has to be scoped to only a specific user, as we don't want to block two different users from adding the same exercise name.

Rescoping the Exercise Controller

While the generated code provided us with all the basic CRUD operations that we need to interact with each of our resources, it's doing it on a global level. What we need to do is reduce the scope, so that each of these actions is only capable of interacting with data that's associated to the current user. That way, little sister Susie can't accidentally (or maliciously) go in and delete all of our workout results for the last few months.

You should recall that when we created our scaffolded exercises controller (`/app/controllers/exercise_controller.rb`), it created a standard set of RESTful methods for us. If you look at them in your code editor, you'll notice that each of those methods starts by setting an `@exercise` instance variable (or `@exercises` in the case of the `index` method). The problem that we have is that the scaffolding had no idea that we wanted to limit the scope of our finds to only the exercises for a specific user, so it's currently set to retrieve any or all exercises regardless of the user, using code like `@exercises = Exercise.find(:all)` or `@exercise = Exercise.new`.

We'll modify this to always use the currently logged in user as the scope for these requests with the new associations that we built. To do that, we'll first need to ensure that no one could access this controller without first being logged in, so we'll add a `before_filter :login_required` call to the top of the controller.

Once we've added that filter, our next step will be to scope all of those finder methods based on the current user. We'll do that by changing our lookups to use our new association to find the exercises—so `@exercises = Exercise.find(:all)` will become `@exercises = current_user.exercises.find(:all)` and `@exercise = Exercise.new` will become `@exercise = current_user.exercises.build`. With those changes in place, our exercises controller should look like this:

```
class ExercisesController < ApplicationController
  before_filter :login_required

  # GET /exercises
  # GET /exercises.xml
  def index
    @exercises = current_user.exercises.find(:all)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @exercises.to_xml }
    end
  end

  # GET /exercises/1
  # GET /exercises/1.xml
  def show
    @exercise = current_user.exercises.find(params[:id])

    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @exercise.to_xml }
    end
  end

  # GET /exercises/new
  def new
    @exercise = current_user.exercises.build
  end

  # GET /exercises/1;edit
  def edit
    @exercise = current_user.exercises.find(params[:id])
  end
```

```
  # POST /exercises
  # POST /exercises.xml
  def create
    @exercise = current_user.exercises.build(params[:exercise])

    respond_to do |format|
      if @exercise.save
        flash[:notice] = 'Exercise was successfully created.'
        format.html { redirect_to exercises_url}
        format.xml  { head :created, :location => exercise_url(@exercise) }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @exercise.errors.to_xml }
      end
    end
  end

  # PUT /exercises/1
  # PUT /exercises/1.xml
  def update
    @exercise = current_user.exercises.find(params[:id])

    respond_to do |format|
      if @exercise.update_attributes(params[:exercise])
        flash[:notice] = 'Exercise was successfully updated.'
        format.html { redirect_to exercises_url }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @exercise.errors.to_xml }
      end
    end
  end

  # DELETE /exercises/1
  # DELETE /exercises/1.xml
  def destroy
    @exercise = current_user.exercises.find(params[:id])
    @exercise.destroy

    respond_to do |format|
      format.html { redirect_to exercises_url }
      format.xml  { head :ok }
    end
  end

end
```

The Exercise Views

With our exercises controller finished, it's merely a matter of making a few modifications to the templates that were generated by the resource scaffolding to finish out our exercise resource.

One of the things that I like to do to keep my templates clean and DRY is to move the forms that are used to create and edit a resource into a single, separate partial, so let's create a new file in `/app/views/exercises` named `_form.rhtml` and place the following code into it:

```
<p>
    <label for="exercise_name">Name</label> <br />
    <%= f.text_field :name %>
</p>
<p>
    <%= submit_tag "Save" %>
</p>
```

With our form partial built, we can now rewrite several of our templates to utilize it; for example, the new template (`/app/views/exercises/new.rhtml`) can be used to create a new exercise or to display errors when creating a new exercise fails validation:

```
<h1>New exercise</h1>

<%= error_messages_for :exercise %>

<% form_for(:exercise, :url => exercises_path) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', exercises_path %>
```

For editing the name of an exercise once it's been added to the system, we'll modify the edit template found in `/app/views/exercises/edit.rhtml` to look like this:

```
<h1>Editing exercise</h1>

<%= error_messages_for :exercise %>

<% form_for(:exercise, :url => exercise_path(@exercise),
                              :html => { :method => :put }) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Show', exercise_path(@exercise) %> |
<%= link_to 'Back', exercises_path %>
```

We'll leave the show template (`/app/views/exercises/show.rhtml`) as it for now, but in the next chapter, we'll look at ways of adding additional value to that view when we add graphing capabilities to our application.

That just leaves us with the task of building our index template (`/app/views/exercises/index.rhtml`), which will be the main page for our users to interact with the exercises. First, we'll want to add some introductory text:

```
<h1>Exercises</h1>
<p>On this page you can create and manage the exercises that you use in your ➥
workouts.</p>
<p>You can also view reports on your progress for each exercises</p>
```

After our introductory text, we're going to want to iterate over the user's list of exercises (stored in the `@exercises` instance variable). We could write that loop directly in our index template with something like this:

```
<% for exercise in @exercises %>
  <tr><td><%=h exercise.name %></td></tr>
<% end %>
```

But a cleaner way to do it is to move the row content into a partial and pass that partial the collection. Let's create a new partial named `_exercise.rhtml` in `/app/views/exercises`, and we'll place the content that we want to display for every exercise in it. We'll want to display the exercise name and links to the view the exercise (show template), edit the exercise (edit template) and a link that will allow a user to delete the exercise from their list:

```
<tr>
   <td><%=h exercise.name %></td>
   <td><%=link_to image_tag("display.gif", {:title => "View Exercise Details"}),
                                              exercise_path(exercise) %></td>
   <td><%=link_to image_tag("edit_photo.gif", {:title => "Edit Exercise"}),
                                              edit_exercise_path(exercise) %></td>
   <td><%= link_to image_tag("delete_photo.gif", {:title => "Delete Exercise"}),
                                         :url => exercise_path(exercise),
                                         :confirm => 'Are you sure?',
                                         :method => :delete %></td>
  </tr>
```

Within our index template, we can render this partial for every exercise in our `@exercises` instance variable by calling it like this:

```
<%= render :partial => 'exercise', :collection => @exercises %>
```

Finally, to make things easier for the end user, we'll also include a form to create exercises on the index page, so users can instantly add a new exercise without having to go to yet another page:

```
<div id="add_exercise">
  <% form_for(:exercise, :url => exercises_path,
                                       :html => {:id => 'new_exercise'}) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
  <% end %>
</div>
```

Once we put all of that together, we'll have an index template that looks like this:

```
<h1>Exercises</h1>
<p>On this page you can create and manage the exercises that you use in ➥
your workouts.</p>
<p>You can also view reports on your progress for each exercises</p>

<table id="exercise_details">
  <tr><th>Name</th></tr>
    <%= render :partial => 'exercise', :collection => @exercises %>
</table>

<br /><br />
<h1>Add a New Exercise</h1>

<div id="add_exercise">
  <% form_for(:exercise, :url => exercises_path,
                                    :html => {:id => 'new_exercise'}) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
  <% end %>
</div>
```

The template renders in a browser as shown in Figure 6-5.

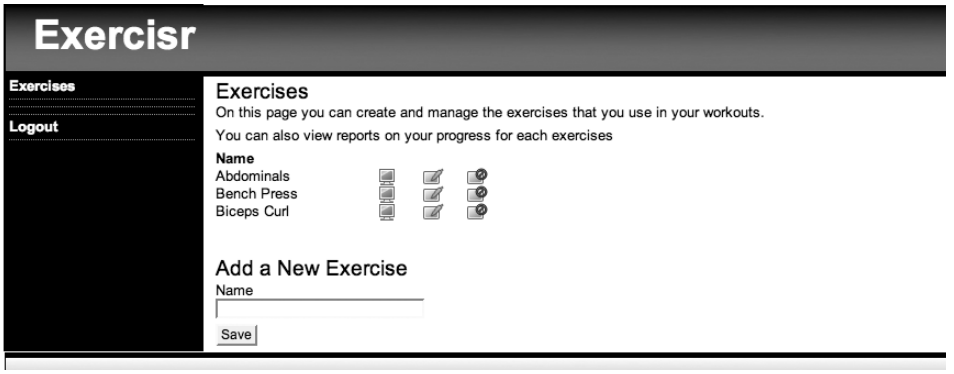

Figure 6-5. *The main index page for our exercises resource*

Now that we have an exercise resource, users need to build a way to record the fact that they worked out on a specific day and capture which exercises they did in each workout, so let's build those processes out now.

The Workout Resource

The next resource we'll build will handle capturing the fact that a user worked out on a specific day. To keep things simple, we'll call this new resource a workout resource. All we'll need to

capture at this level is simply the date of the workout and an optional text description of what type of workout it was (e.g, upper body, abdominals, or arms). We'll create this resource like this:

```
ruby script/generate scaffold_resource Workout date:date label:string ➥
user_id:integer
```

```
output omitted for brevity
```

The first thing we need to do with our new scaffold is to remove the automatically generated layout (`workout.rhtml`) from `/app/views/layouts` so that it won't override our application layout. Once the layout is removed, we'll run our new migration to add the workouts table to the database:

```
rake db:migrate
```

```
== CreateWorkouts: migrating ======================================
-- create_table(:workouts)
   -> 0.0780s
== CreateWorkouts: migrated (0.0930s) =============================
```

The Workout Model and Associations

With the table added to the database, let's round out the models and associations. Since we built the workout model to capture the `user_id`, we'll add in a `belongs_to :user` and basic validation to ensure that we receive a date. Edit our workout model (`/app/models/workout.rb`) to look like this:

```
class Workout < ActiveRecord::Base
  belongs_to :user
  validates_presence_of :date
end
```

We'll also build out the reciprocal association from the user model (`/app/models/user.rb`) to add in a `has_many :workouts` relationship. We'll also pass that association a `:dependent => :destroy` option to ensure that Rails will delete all associated workout objects in the event that we ever delete a user (so we can avoid leaving orphaned data in the database):

```
has_many :workouts, :dependent => :destroy
```

The Workout Controller

The scaffolding added a new route to our routes file as `map.resources :workouts`, which we'll leave as is for now, so we can now turn our attention to modifying our workout controller. For the most part, we'll be able to keep the scaffold-generated code; we merely need to add in the `before_filter :login_required` call to limit access to the page, and by making the same types of modifications that we did in our exercises controller, we can scope the results based on the currently logged in user:

```
class WorkoutsController < ApplicationController
  before_filter :login_required

  # GET /workouts
  # GET /workouts.xml
  def index
    @workouts = current_user.workouts.find(:all, :order => 'date desc',
                                           :limit => 10)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @workouts.to_xml }
    end
  end

  # GET /workouts/1
  # GET /workouts/1.xml
  def show
    @workout = current_user.workouts.find(params[:id])

    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @workout.to_xml }
    end
  end

  # GET /workouts/new
  def new
    @workout = current_user.workouts.build
  end
```

```
  # GET /workouts/1;edit
  def edit
    @workout = current_user.workouts.find(params[:id])
  end

  # POST /workouts
  # POST /workouts.xml
  def create
    @workout = current_user.workouts.build(params[:workout])

    respond_to do |format|
      if @workout.save
        flash[:notice] = 'Workout was successfully created.'
        format.html { redirect_to workout_url(@workout) }
        format.xml  { head :created, :location => workout_url(@workout) }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @workout.errors.to_xml }
      end
    end
  end

  # PUT /workouts/1
  # PUT /workouts/1.xml
  def update
    @workout = current_user.workouts.find(params[:id])

    respond_to do |format|
      if @workout.update_attributes(params[:workout])
        flash[:notice] = 'Workout was successfully updated.'
        format.html { redirect_to workout_url(@workout) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @workout.errors.to_xml }
      end
    end
  end

  # DELETE /workouts/1
  # DELETE /workouts/1.xml
  def destroy
    @workout = current_user.workouts.find(params[:id])
    @workout.destroy
```

```
    respond_to do |format|
      format.html { redirect_to workouts_url }
      format.xml  { head :ok }
    end
  end
end
```

Modifying the Views

Our modifications to the workout views will also be very similar to the modifications we did for exercises. We'll move the forms for creating and editing a workout into a partial, which we'll also include on the index page. For iterating over our list of workouts, we'll also use another partial.

So our index template (`/app/views/workouts/index.rhtml`) will look like this:

```
<h1>Listing workouts</h1>

<table>
  <tr><th>Date</th><th>Label</th></tr>
  <%= render :partial => 'workout', :collection => @workouts %>
</table>
<br />

<h1>Add a New Workout</h1>
<div id="add_workout">
  <% form_for(:workout, :url => workouts_path,
                                  :html => {:id => 'new_workout'}) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
  <% end %>
</div>
```

Of course, before this template will work, we'll need to build those workout and form partials, so let's build those now. The first one that we'll look at is the workout partial, which we'll use to iterate over each workout in the `@workouts` instance variable.

Create a new file in `/app/views/workouts` named `_workout.rhtml` and place the following content in it to display the basic information for the workout and provide links to the `show`, `edit`, and `delete` methods:

```
<tr>
  <td><%= workout.date.to_s(:long) %></td>
  <td><%= workout.label %></td>
  <td>
    <%=link_to image_tag("display.gif", {:title => "View Workout Details"}),
                                                    workout_path(workout) %>
  </td>
```

```
  <td>
    <%=link_to image_tag("edit_photo.gif", {:title => "Edit Workout Date/Label"}),
                                                edit_workout_path(workout) %>
  </td>
  <td>
    <%= link_to image_tag("delete_photo.gif", {:title => "Delete Workout"}),
                                               workout_path(workout),
                                              :confirm => 'Are you sure?',
                                              :method => :delete %>
  </td>
</tr>
```

The second partial that we're including on the index page is the form that we use to create a new workout. Create a new file in `/app/views/workouts/` named `_form.rhtml` and place the following form content in it:

```
<p>
    <b>Date</b><br />
    <%= f.date_select :date %>
</p>

<p>
    <b>Label</b><br />
    <%= f.text_field :label %>
</p>

<p>
    <%= submit_tag "Save" %>
</p>
```

Now that we have the workout form partial created, we can also utilize it in our edit and create templates in `/app/views/workouts`. So our `show.rhtml` template will look like this:

```
<h1>Editing workout</h1>
<%= error_messages_for :workout %>

<% form_for(:workout, :url => workout_path(@workout),
                                        :html => { :method => :put }) do |f| %>
   <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Show', workout_path(@workout) %> |
<%= link_to 'Back', workouts_path %>
```

And our `new.rhtml` template will look like this:

```
<h1>New workout</h1>
<%= error_messages_for :workout %>

<% form_for(:workout, :url => workouts_path) do |f| %>
   <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', workouts_path %>
```

Finally, now that we have our workout resource built, we can go back to our application layout (`/app/views/layouts/application.rhtml`) and uncomment the line from the sidebar area that will display the link to the workouts index page:

```
<% if logged_in? %>
  <div class="yui-b sidebar">
    <ul>
      <li><%= link_to 'Exercises', exercises_path %></li>
      <li><%= link_to 'Workouts', workouts_path %></li>
      <li><%# link_to 'Goals', goals_path %></li>
      <li><%= link_to 'Logout', logout_path %></li>
    </ul>
  </div>
<% end %>
```

Capturing Our Workouts

Now that we've built out the process for capturing when a user works out, we need to turn our attention to capturing what the user did at each workout. So after creating a workout, we will need to collect the following information:

- The exercises performed
- The number of sets of each exercise
- The weight or resistance used in each set
- The number of repetitions (i.e., how many times the user was able to perform the exercise) in each set

With those goals clearly in mind, determining our database structure should be fairly obvious. Each row of our table will represent one set of an exercise and capture the workout ID as a foreign key (so it can be associated back to the workout), the exercise ID as a foreign key (to associate it back to an exercise from our exercises resource), the amount of resistance used for the set, and the number of repetitions performed during that set.

So now all we need to do is determine a good name for this resource. It would be good to name this resource "exercises," but we've already used that name to maintain our master list of possible exercises. Another good name might be "sets," since each row in the database should match to a single set that we performed. Unfortunately, that would be a path fraught with pain as the word "set" is a reserved word within Ruby. That eliminates our two most obvious names for this resource. After much mental strain, I came up with the name of "activities," as in each workout we did many activities. With our name and our database structure ready, we'll create the resource like this:

```
ruby script/generate scaffold_resource Activity workout_id:integer ➥
exercise_id:integer resistance:integer repetitions:integer
```

```
output omitted for brevity
```

Once again, we'll need to delete the scaffold-generated layout (`/app/views/layouts/activity.rhtml`) and run our new migration to add the table to our database:

```
rake db:migrate
```

```
== CreateActivities: migrating ========================
-- create_table(:activities)
   -> 0.0780s
== CreateActivities: migrated (0.0780s) ===============
```

Building Our Activities Model and Associations

The activity model contains foreign keys to our exercise and workout models, so we'll need to include a pair of `belongs_to` methods in our model, and we'll want to have some basic validations. Our activity model (`/app/models/activity.rb`) should look like this:

```
class Activity < ActiveRecord::Base
  belongs_to :exercise
  belongs_to :workout
  validates_presence_of :resistance, :repetitions
end
```

We can also modify our workouts model (`/app/models/workout.rb`) to both recognize that a workout has many activities and to serve as a `:through` bridge to exercises:

```
class Workout < ActiveRecord::Base
  belongs_to :user
  has_many :activities, :dependent => :destroy
  has_many :exercises, :through => :activities
  validates_presence_of :date
end
```

With those associations we can now pull down the lists of associated activities or exercises for a given workout like this:

```
workout = Workout.find 5
```

```
=> #<Workout:0x14b709c @attributes={"date"=>"2007-01-17", "id"=>"5", ➥
"user_id"=>"2", "label"=>"Back / Biceps"}
```

```
workout.activities
```

```
=> [#<Activity:0x137db04 @attributes={"exercise_id"=>"9", "id"=>"10", ➥
"repetitions"=>"12", "workout_id"=>"5", "resistance"=>"150"}, ➥
#<Activity:0x137d758  @attributes={"exercise_id"=>"9", "id"=>"11", ➥
"repetitions"=>"12", "workout_id"=>"5", "resistance"=>"175"}, ➥
#<Activity:0x137d730 @attributes={"exercise_id"=>"9", "id"=>"12", ➥
"repetitions"=>"12", "workout_id"=>"5", "resistance"=>"180"}]
```

```
workout.exercises
```

```
=> [#<Exercise:0x27615a0 @attributes={"name"=>"Lat Pulldowns", ➥
"exercise_type"=>nil, "id"=>"9", "user_id"=>"2"}, #<Exercise:0x2761578 ➥
@attributes={"name"=>"Lat Pulldowns", "exercise_type"=>nil, "id"=>"9", ➥
"user_id"=>"2"}, #<Exercise:0x2761550 @attributes={"name"=>"Lat Pulldowns", ➥
 "exercise_type"=>nil, "id"=>"9", "user_id"=>"2"}]
```

We can also add an association back to our activities model from within the user model (`/app/models/user.rb`) by passing through the workouts model:

```
require 'digest/sha1'
class User < ActiveRecord::Base
  # Virtual attribute for the unencrypted password
  attr_accessor :password

  validates_presence_of     :login, :email
  validates_presence_of     :password,                   :if => :password_required?
  validates_presence_of     :password_confirmation,      :if => :password_required?
  validates_length_of       :password, :within => 4..40, :if => :password_required?
  validates_confirmation_of :password,                   :if => :password_required?
  validates_length_of       :login,    :within => 3..40
  validates_length_of       :email,    :within => 3..100
  validates_uniqueness_of   :login, :email, :case_sensitive => false
  before_save :encrypt_password
```

```
  has_many :workouts, :dependent => :destroy
  has_many :exercises, :dependent => :destroy, :order => 'name asc'
  has_many :activities, :through => :workouts
 (….remainder of User model ommitted for brevity)
```

Modifying the Activities Routes

Our activities resource is going to require some custom routing rules, though, so edit `/config/routes.rb` to look like this:

```
ActionController::Routing::Routes.draw do |map|

  map.resources :workouts do |workout|
    workout.resources :activities
  end

  map.resources :exercises

  map.home '', :controller => 'sessions', :action => 'new'
  map.resources :users, :sessions
  map.welcome '/welcome', :controller => 'sessions', :action => 'welcome'
  map.signup '/signup', :controller => 'users', :action => 'new'
  map.login '/login', :controller => 'sessions', :action => 'new'
  map.logout '/logout', :controller => 'sessions', :action => 'destroy'
end
```

We've modified our workouts and exercises resources to place activities as a nested resource underneath workouts. This is a way of declaring that a particular resource is only valid within the subcontext of another resource. Let's use an analogy of parents and children to hopefully make this clearer.

In a RESTful application, if I wanted to read a list of all parents I could simply issue a `GET` request to `/parents`. While if I wanted to see the details on a specific parent, I might issue a `GET` request to `/parents/1`.

Similarly, if I wanted to see a list of all children, I could use a `GET /children` request, and to view the details of a specific child, I would use `GET /children/1`.

The problem comes in when it doesn't really make sense for me to access the list of children by themselves. What if I don't have any use for the children outside of their relation to their parents? In that case, what I need is the ability to make my RESTful routing calls on a child resource in relation to their parent, and that's exactly what nested routing provides us.

When we declare children as a nested resource underneath parents, we can avoid having to pass additional parameters to our `GET /children` request to specify the specific parent to whom we want to limit our request. We can now send a `GET` request to `/parents/1/children` to see a list of all the children that belong to a particular parent or a `GET` request to `/parents/1/children/2` to see the details on a specific child.

Some other ideas of where a nested route might be used can be found in Table 6-4.

Table 6-4. *Nested Route Examples*

Parent	Child
Goal	Results
Forum	Posts
Article	Comments
Book	Chapters
State	Cities

Obviously, the resource routing capabilities of Rails are pretty exciting stuff, and the ability to easily nest resources like this is extra cool. In our application, we could make a `GET` request to `/workout/1` to view the information on a specific workout and to pull in a list of all the exercises that in that workout would be a simple `GET` request to `/workout/1/activities`.

Modifying the Activities Controller

As usual, the first thing we'll need to add is a `before_filter :login_required`, so we can ensure that only a user who's logged in is able to access the methods in this controller. However, after that, those powerful nested resources changes do come with a small cost for us, as using them means we'll have to make a few more changes to the activities controller (`/app/controllers/activities_controller.rb`) than we have in previous controllers, primarily in terms of how we scope our finds. In previous controllers, we scoped all of our finds based on the currently logged in user, but in a nested resource, we'll also need to look up the parent resource.

In the case of an activity, we also will want to look up the associated workout to use in scoping our activities. We'll do this by adding a new `before_filter` and a new protected method to our activities controller. At the top of our controller, we'll add a `before_filter` like this:

```
before_filter :find_workout
```

And down at the bottom, we'll add a protected method named `find_workout`, which will look up the workout object and place it in an `@workout` instance variable:

```
protected
def find_workout
  @workout = current_user.workouts.find(params[:workout_id])
end
```

Now that we have the current user's workout available in the `@workout` instance variable, we can use it in all of our finds within our seven RESTful methods in the activities controller. So our activities controller will look like this:

```
class ActivitiesController < ApplicationController
  before_filter :login_required
  before_filter :find_workout
```

```
  # GET /activities
  # GET /activities.xml
  def index
    @activities = @workout.activities.find(:all)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @activities.to_xml }
    end
  end

  # GET /activities/1
  # GET /activities/1.xml
  def show
    @activity = @workout.activities.find(params[:id])

    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @activity.to_xml }
    end
  end

  # GET /activities/new
  def new
    @activity = @workout.activities.build
  end

  # GET /activities/1;edit
  def edit
    @activity = @workout.activities.find(params[:id])
  end

  # POST /activities
  # POST /activities.xml
  def create
    @activity = @workout.activities.build(params[:activity])

    respond_to do |format|
      if @activity.save
        flash[:notice] = 'Activity was successfully created.'
        format.html { redirect_to workout_url(@workout) }
        format.xml  { head :created, :location => activity_url(@workout, @activity)}
```

```
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @activity.errors.to_xml }
      end
    end
  end

  # PUT /activities/1
  # PUT /activities/1.xml
  def update
    @activity = @workout.activities.find(params[:id])

    respond_to do |format|
      if @activity.update_attributes(params[:activity])
        flash[:notice] = 'Activity was successfully updated.'
        format.html { redirect_to workout_url(@workout) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @activity.errors.to_xml }
      end
    end
  end

  # DELETE /activities/1
  # DELETE /activities/1.xml
  def destroy
    @activity = @workout.activities.find(params[:id])
    @activity.destroy

    respond_to do |format|
      format.html { redirect_to activities_url }
      format.xml  { head :ok }
    end
  end

  protected
  def find_workout
    @workout = current_user.workouts.find(params[:workout_id])
  end
end
```

Modifying Activities View Templates

We will modify the view templates for our activities following the pattern we established with previous resources—with one important difference. Recall that because we've established activities as a nested resource, we've essentially said that activities are only valid in the context of their relationship to a workout. This means that whenever we link to an activity resource, we must always provide the `workout_id` as well. So we need to pass in the `@workout` instance variable as an additional parameter to all of our activities named route methods, such as

```
new_activity_path(@workout)
edit_activity_path(@workout, @activity)
```

With that knowledge, we can go ahead and modify our view templates for activities. We'll start out by building our two standard partials, the first being the activity partial that will be used to iterate over our list of activities. Create a new file named `_activity.rhtml` in `/app/views/activities` with the following links in it:

```
<tr>
  <td><%= activity.exercise.name %></td>
  <td><%= activity.repetitions %></td>
  <td><%= activity.resistance %></td>
  <td>
    <%=link_to image_tag("edit_photo.gif", {:title => "Edit Exercise"}),
                                        edit_activity_path(@workout, activity) %>
  </td>
  <td>
    <%= link_to image_tag("delete_photo.gif", {:title => "Delete Exercise"}),
                                       activity_path(@workout, activity),
                                       :confirm => 'Are you sure?',
                                       :method => :delete %>
  </td>
</tr>
```

Next, we'll create the form partial that we'll use to create a new activity. Create a new file named `_form.rhtml` in `/app/views/activities` with the following content:

```
<p>
  <%= f.collection_select :exercise_id, current_user.exercises.find(:all), :id,
                                       :name, :prompt => "Select an Exercise" %>
</p>
<p>One set of <%= f.text_field :repetitions %> with ➥
                     <%= f.text_field :resistance %> pounds of resistance</p>
<p>
  <%= submit_tag "Save" %>
</p>
```

We'll modify the edit template (`/app/views/activities/edit.rhtml`) to utilize our form partial like this:

```
<h1>Editing activity</h1>
<%= error_messages_for :activity %>

<% form_for(:activity, :url => activity_path(@workout, @activity),
                              :html => { :method => :put }) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', workout_path(@workout) %>
```

We'll also modify the `index` method for activities (`/app/views/activities/index.rhtml`) like this:

```
<h1>Listing activities</h1>

<table>
   <tr><th>Exercise</th><th>Reps</th><th>Resistance</th></tr>
   <%= render :partial => 'activity', :collection => @activities %>
</table>
<br />

<%= link_to 'New activity', new_activity_path(@workout) %>
```

We'll modify the new template (`/app/views/activities/new.rhtml`) like this:

```
<h1>New activity</h1>
<%= error_messages_for :activity %>

<% form_for(:activity, :url => activities_path(@workout)) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', workout_path(@workout) %>
```

Finally, we'll change the show template (`/app/views/activities/show.rhtml`) to look like this:

```
<p>
  <b>Exercise:</b>
  <%=h @activity.exercise.name %>
</p>
```

```
<p>
  <b>Resistance:</b>
  <%=h @activity.resistance %>
</p>

<p>
  <b>Repetitions:</b>
  <%=h @activity.repetitions %>
</p>

<%= link_to 'Edit', edit_activity_path(@workout, @activity) %> |
<%= link_to 'Back', activities_path(@workout) %>
```

Modifying the Show Method for a Workout

Even though we set up all the templates for the activities resource, in reality, we don't want a user to have to navigate to those pages—especially since we've nested activities underneath workouts. What we'll do instead is modify the show template in our workouts resource to serve as the primary interface for an end user to add activities to a workout.

Our first step in doing that is to modify the show method in our workouts controller (`/app/controllers/workouts_controller.rb`) to also generate an `@activities` instance variable containing a list of that workout's activities. To avoid doing *N*+1 queries to pull in the associated exercise name for each activity, we'll load the associated exercise object for each activity using `:include => :exercise` (you can see the result of this template in Figure 6-6):

```
def show
    @workout = current_user.workouts.find(params[:id])
    @activities = @workout.activities.find(:all, :include => :exercise)

    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @workout.to_xml }
    end
  end
```

With the `@activities` instance variable set, we modify the show template for our workouts (`/app/views/workouts/show.rhtml`) to render our activity partial to show the exercises that we performed in that workout and the activity form partial to be able to add a new exercise activity to the workout:

```
<h1><%= h @workout.label %> Workout on <%= h @workout.date.to_s(:long) %> </h1>
<table>
   <tr><th>Exercise</th><th>Reps</th><th>Resistance</th></tr>
   <%= render :partial => 'activities/activity', :collection => @activities %>
</table>
```

```
<h3>Add Exercise to this Workout</h3>
<% form_for(:activity, :url => activities_path(@workout)) do |f| %>
   <%= render :partial => 'activities/form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', workouts_path %>
```

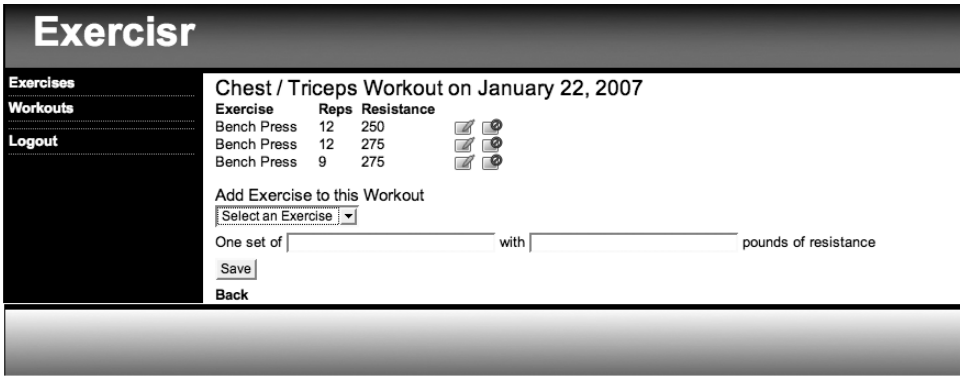

Figure 6-6. *Adding Exercises to a Workout*

Improving the Add Activity Form

The one thing that I really don't like about the current process is that if I realize that I'm missing an exercise in my drop-down list while I'm adding activities to a workout, my only option is to leave the workouts pages and go add my missing exercise back in the exercises section. Now, granted, the more I use the application, the less likely that situation will become, but it is still an irritation that I'd like to avoid if I can.

Fortunately, it is one we can solve with a fairly minor amount of additional code. To start with, let's add a new form field to our activity form partial (`/app/views/activities/_form.rhtml`) that can be used to capture a new exercise name:

```
<p>
  <%= f.collection_select :exercise_id, current_user.exercises.find(:all), :id,
                                        :name, :prompt => "Select an Exercise" %>
  or add a new exercise:
  <%= f.text_field :new_exercise_name %>
</p>
<p>One set of <%= f.text_field :repetitions %> with ➥
                          <%= f.text_field :resistance %> pounds of resistance</p>
<p>
  <%= submit_tag "Save" %>
</p>
```

Because the fields in this form are supposed to tie back to attributes in an activity model, this would currently generate an error if we tried to render it, since there is no corresponding field in the activities table named `new_exercise_name`. What we need to do is create `new_exercise_name` as a virtual attribute by modifying our activity model (`/app/models/activity.rb`) with the following line:

```
attr_accessor :new_exercise_name
```

Now that we have a way to capture the submitted value in memory, we can simply create a new method in our activity model named `create_exercise_if_submitted`. In this method, we'll call the `create_exercise` method (this method is added to our model by our `belongs_to` association) from a `before_save` callback. We'll pass it the appropriate `user.id` and the `new_exercise_name` virtual attribute to create a new exercise object directly from the activity model. So our activity model will look like this:

```
class Activity < ActiveRecord::Base
  belongs_to :exercise
  belongs_to :workout
  validates_presence_of :resistance, :repetitions

  attr_accessor :new_exercise_name
  before_save :create_exercise_if_submitted

  def create_exercise_if_submitted
    create_exercise(:user_id => workout.user_id, :name => new_exercise_name) ➥
unless new_exercise_name.blank?
  end
end
```

Our new workout view will look like the one shown in Figure 6-7 and will allow a user to either select an exercise from the drop-down list or create a new exercise directly.

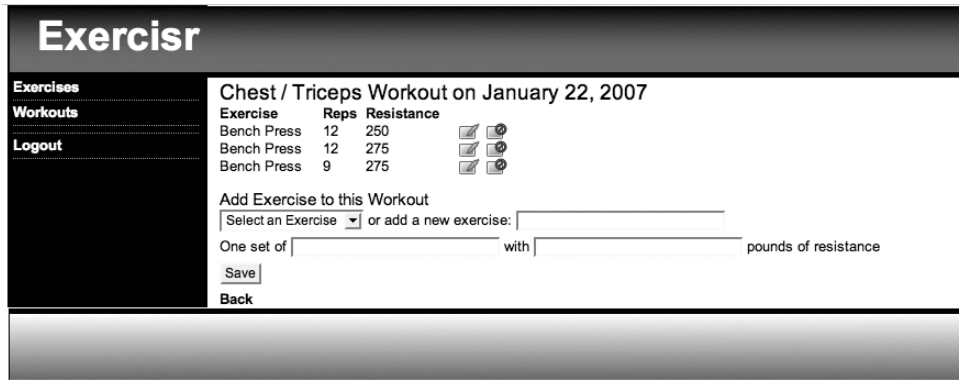

Figure 6-7. *Allowing users to create a new exercise while adding an activity*

Tracking Fitness Goals

The initial planning for this project also included supporting the ability to track general health goals such as weight, blood sugar, and so on. To accomplish this, we'll add yet another resource that we'll name "goals." Attributes for a goals resource would be the name of the goal that we're tracking and the target goal that we're trying to achieve, and we'll give ourselves a little cheat by storing the last result for this goal in the object as well (that way, we can very simply do things like calculate the difference between where we are currently and how much further until we reach our goal). We'll create our goal resource with this command:

```
ruby script/generate scaffold_resource Goal name:string value:decimal ➥
last:decimal user_id:integer
```

```
output omitted for brevity
```

Finally, we need a way to capture our individual results toward reaching our goals. For example, assuming that our goal is tracking our current weight, we'd want a place to capture weekly weigh-in results. We'll name this resource "results." The attributes for a results object would be to know which goal it's associated with, the date of this result, and the value that we're recording. We'll create this resource with this command:

```
ruby script/generate scaffold_resource Result goal_id:integer ➥
date:date value:decimal
```

```
output omitted for brevity
```

And with those few commands, we've just generated a significant portion of our application code to support tracking fitness goals. We can now go manipulate and massage the generated code into working the way that we want it to work. Before we do that though, let's run the migration files that our scaffolding generated to create our database tables.

```
rake db:migrate
```

```
== CreateGoals: migrating ======================================
-- create_table(:goals)
   -> 0.0780s
== CreateGoals: migrated (0.0780s) =============================
== CreateResults: migrating ====================================
-- create_table(:results)
   -> 0.0780s
== CreateResults: migrated (0.0780s) ===========================
```

With our database prepped for use, go ahead and remove the scaffold-generated `goals.rhtml` and `results.rhtml` layout files from `/app/views/layouts/`, and then we can direct our attention to configuring our models with our necessary associations and validations for our application.

Modifying Our Models

Our goals model (`/app/models/goal.rb`) should stay fairly simple for now; we'll associate it with our user model with a `belongs_to` association, add a `has_many` relationship to results, and add in some basic validations:

```
class Goal < ActiveRecord::Base
  belongs_to :user
  has_many :results, :dependent => :destroy
  validates_presence_of :name, :value
end
```

Next, we'll modify our results model (`/app/models/result.rb`) to include a `belongs_to` association with our goal, and we'll add in a basic `validates_presence_of` requirement for the date and value elements:

```
class Result < ActiveRecord::Base
  belongs_to :goal
  validates_presence_of :date, :value
end
```

Finally—we need to configure our user model (`/app/models/user.rb`) with an association back to the Goal mode:

```
has_many:goals
```

Setting Up a Nested Route

Open `/config/routes.rb`, and we'll modify our goals and results resources as a nested resource. Afterward, your `routes.rb` configuration file should look exactly like this:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :goals do |goal|
    goal.resources :results
  end

  map.resources :workouts do |workout|
    workout.resources :activities
  end

  map.resources :exercises

  map.home '', :controller => 'sessions', :action => 'new'
  map.resources :users, :sessions
  map.welcome '/welcome', :controller => 'sessions', :action => 'welcome'
```

```
  map.signup '/signup', :controller => 'users', :action => 'new'
  map.login '/login', :controller => 'sessions', :action => 'new'
  map.logout '/logout', :controller => 'sessions', :action => 'destroy'
end
```

Configuring Our Controllers

The modifications that we need to make to the goals controller should be old hat to you by now. First, we'll need to add the `login_required` before filter, so we can control access to the methods. Second, we'll need to limit the scope all of our finds to only the currently logged in users goals. Finally, because we're going to display goal results on the show template, we need to generate a list of results in an `@results` instance variable within the show method.

Edit `/app/controllers/goals_controllers.rb` to look like this:

```
class GoalsController < ApplicationController
  before_filter :login_required

  # GET /goals
  # GET /goals.xml
  def index
    @goals = current_user.goals.find(:all)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @goals.to_xml }
    end
  end

  # GET /goals/1
  # GET /goals/1.xml
  def show
    @goal = current_user.goals.find(params[:id])
    @results = @goal.results.find(:all, :order => 'date desc')

    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @goal.to_xml }
    end
  end

  # GET /goals/new
  def new
    @goal = current_user.goals.build
  end

  # GET /goals/1;edit
  def edit
    @goal = current_user.goals.find(params[:id])
```

```
  end

  # POST /goals
  # POST /goals.xml
  def create
    @goal = current_user.goals.build(params[:goal])

    respond_to do |format|
      if @goal.save
        flash[:notice] = 'Goal was successfully created.'
        format.html { redirect_to goal_url(@goal) }
        format.xml  { head :created, :location => goal_url(@goal) }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @goal.errors.to_xml }
      end
    end
  end

  # PUT /goals/1
  # PUT /goals/1.xml
  def update
    @goal = current_user.goals.find(params[:id])

    respond_to do |format|
      if @goal.update_attributes(params[:goal])
        flash[:notice] = 'Goal was successfully updated.'
        format.html { redirect_to goal_url(@goal) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @goal.errors.to_xml }
      end
    end
  end

  # DELETE /goals/1
  # DELETE /goals/1.xml
  def destroy
    @goal = current_user.goals.find(params[:id])
    @goal.destroy

    respond_to do |format|
      format.html { redirect_to goals_url }
      format.xml  { head :ok }
    end
```

```
  end
end
```

Editing the Results Controller

Next, we'll need to modify the results controller (`/app/controllers/results_controller.rb`); because it's a nested resource, it will require editing in a similar fashion to the activities controller.

We'll start out by limiting access with the `login_required` before filter and then make another `before_filter` call to populate the `@goals` instance variable with the parent goal for these results. We'll then use that `@goals` variable to scope our finders within the standard RESTful methods. Finally, we'll need to modify the redirect destinations for the `create`, `update`, and `destroy` methods to send the user back to the goal detail page.

In the end, your `results_controller.rb` should look like this:

```
class ResultsController < ApplicationController
  before_filter :login_required
  before_filter :find_goal

  # GET /results
  # GET /results.xml
  def index
    @results = @goal.results.find(:all)

    respond_to do |format|
      format.html # index.rhtml
      format.xml  { render :xml => @results.to_xml }
    end
  end

  # GET /results/1
  # GET /results/1.xml
  def show
    @result = @goal.results.find(params[:id])

    respond_to do |format|
      format.html # show.rhtml
      format.xml  { render :xml => @result.to_xml }
    end
  end

  # GET /results/new
  def new
    @result = @goal.results.build
  end
```

```
# GET /results/1;edit
def edit
  @result = @goal.results.find(params[:id])
end

# POST /results
# POST /results.xml
def create
  @result = @goal.results.build(params[:result])

  respond_to do |format|
    if @result.save
      flash[:notice] = 'Result was successfully created.'
      format.html { redirect_to goal_url(@goal) }
      format.xml  { head :created, :location => result_url(@result) }
    else
      format.html { render :action => "new" }
      format.xml  { render :xml => @result.errors.to_xml }
    end
  end
end

# PUT /results/1
# PUT /results/1.xml
def update
  @result = @goal.results.find(params[:id])

  respond_to do |format|
    if @result.update_attributes(params[:result])
      flash[:notice] = 'Result was successfully updated.'
      format.html { redirect_to goal_url(@goal) }
      format.xml  { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml  { render :xml => @result.errors.to_xml }
    end
  end
end

# DELETE /results/1
# DELETE /results/1.xml
def destroy
  @result = @goal.results.find(params[:id])
  @result.destroy
```

```
    respond_to do |format|
      format.html { redirect_to goal_url(@goal) }
      format.xml  { head :ok }
    end
  end

  protected
  def find_goal
    @goal = current_user.goals.find(params[:goal_id])
  end
end
```

Configuring Our Views

The first thing we'll need to do is open our layout (`/app/views/layouts/application.rhtml`) and uncomment the line in the sidebar div that provides a link to our goals index page:

```
<div class="yui-b sidebar">
  <ul>
    <li><%= link_to 'Exercises', exercises_path %></li>
    <li><%= link_to 'Workouts', workouts_path %></li>
    <li><%= link_to 'Goals', goals_path %></li>
    <li><%= link_to 'Logout', logout_path %></li>
  </ul>
</div>
```

The Goals Views

The modifications we made to our goals templates will be nearly identical to the ones that we made to the workout templates. So let's jump right in by extracting out a partial named `_form.rhtml` for the new and edit pages. Create `_form.rhtml` in `/app/views/goals` as follows:

```
<p><label for="goal_name">Name of the Goal:</label><br />
<%= f.text_field :name %></p>

<p><label for="goal_value">Goal to Reach:</label><br />
<%= f.text_field :value %></p>

<p><label for="goal_last">Current Result:</label><br />
<%= f.text_field :last %></p>

<p><%= submit_tag "Save" %></p>
```

Now, we'll modify the edit and new templates to use our new partial. We'll alter `/app/views/goals/edit.rhtml` as follows:

```
<h1>Editing goal</h1>
```

```
<%= error_messages_for :goal %>

<% form_for(:goal, :url => goal_path(@goal), :html => { :method => :put }) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Show', goal_path(@goal) %> |
<%= link_to 'Back', goals_path %>
```

We'll alter /app/view/goals/new.rhtml in a similar fashion:

```
<h1>New goal</h1>

<%= error_messages_for :goal %>

<% form_for(:goal, :url => goals_path) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', goals_path %>
```

Once again, we'll extract out the iteration of our goals from the index page to a partial named _goal.rhtml that we'll use to render the collection. So create a file named _goal.rhtml in /app/views/goals with the following content in it:

```
<tr>
  <td><%=h goal.name %></td>
  <td> </td>
  <td>
    <%= link_to image_tag("display.gif", {:title => "View Report"}),
                                                goal_path(goal) %>
  </td>
  <td>
    <%=link_to image_tag("edit_photo.gif", {:title => "Edit Goal Details"}),
                                                edit_goal_path(goal) %>
  </td>
  <td>
    <%= link_to image_tag("delete_photo.gif", {:title => "Delete Goal"}),
                                                goal_path(goal),
                                                :confirm => 'Are you sure?',
                                                :method => :delete %>
  </td>
</tr>
```

Finally, we'll edit the index page (`/app/views/goals/index.rhtml`) to utilize our two new partials:

```
<h1>Listing goals</h1>
<table>
  <tr><th>Name</th></tr>
   <%= render :partial => 'goal', :collection => @goals %>
</table>

<br />

<h1>Add a New Goal</h1>

<div id="add_goal">
  <% form_for(:goal, :url => goals_path, :html => {:id => 'new_goal'}) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
  <% end %>
</div>
```

The Results Views

Editing the templates for the results resource will also be old hat to you at this point, so we'll whip through these fairly quickly. First, we'll generate the same two partials that we have for previous resources and modify the standard templates to use them. Second, because this is a nested resource, we'll also modify any of our named routes to pass the `@goals` variable as well.

Create a new partial named `/app/views/results/_form.rhtml`:

```
<p><label for="">Date</label><br />
<%= f.date_select :date %></p>

<p><label for="">Value</label><br />
<%= f.text_field :value %></p>

<p><%= submit_tag "Save" %></p>
```

Next, we'll create the results partial to display our collection of results. Create `_result.rhtml` in `/app/views/results/`, and place the following content in it:

```
<tr>
   <td><%=h result.date.to_s(:long) %></td>
   <td><%=h result.value %></td>
   <td><%=link_to image_tag("edit_photo.gif", {:title => "Edit Result Details"}),
                                          edit_result_path(@goal, result) %></td>
   <td><%= link_to image_tag("delete_photo.gif", {:title => "Delete Result"}),
                                              result_path(@goal, result),
                                              :confirm => 'Are you sure?',
                                              :method => :delete %></td>
</tr>
```

Now, we'll modify our templates to utilize these partials. Edit /app/views/results/new.rhtml to look like this:

```
<h1>New result</h1>
<%= error_messages_for :result %>

<% form_for(:result, :url => results_path(@goal)) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', results_path(@goal) %>
```

And edit /app/views/results/edit.rhtml to look like this:

```
<h1>Editing result</h1>
<%= error_messages_for :result %>

<% form_for(:result, :url => result_path(@goal, @result),
                                        :html => { :method => :put }) do |f| %>
    <%= render :partial => 'form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back to Goal', goal_path(@goal) %>
```

Meanwhile, the results index page (/app/views/results/index.rhtml) should look like this:

```
<h1>Listing results for <%= h @goal.name %></h1>

<table>
  <tr><th>Date</th><th>Value</th></tr>
      <%= render :partial => 'result', :collection => @results %>
</table>

<br />

<%= link_to 'Back to Goal', goal_path(@goal) %>
```

Finally, we can wrap up these templates by building the goals show template, which will utilize our two new results partials. Edit /app/views/goals/show.rhtml to look like this:

```
<h1>Results for <%= h @goal.name %> </h1>
<table>
   <tr><th>Date</th><th>Value</th></tr>
   <%= render :partial => 'results/result', :collection => @results %>
</table>
```

```
<h3>Record New Result for this Goal</h3>
<% form_for(:result, :url => results_path(@goal)) do |f| %>
   <%= render :partial => 'results/form', :locals => {:f => f} %>
<% end %>

<%= link_to 'Back', workouts_path %>
```

Capturing the Last Result

When we defined our goals resource, we included that we wanted it to break normalization by storing the duplicate data of the most recent result, but we haven't built a way to capture that data yet. Thanks to Rails's powerful callback support this is an easy problem to solve, as we can simply add an `after_create` callback in our results model to magically populate the last attribute for the associated goal. Open `/app/models/results.rb`, and add our functionality like this:

```
class Result < ActiveRecord::Base
  belongs_to :goal
  validates_presence_of :date, :value
  after_create :update_last_result

  def update_last_result
    goal.last = value
    goal.save
  end
end
```

Now, whenever we create a new result, its value will also be populated into the goals last field—incredibly powerful yet seriously easy.

Exploring the RESTful Interface

With that last change, our application is well on its way. We can easily use the HTML interface to create and manage our workouts and goals, but what if we wanted to play around with the XML interface that we gained for free with REST?

The easiest way to do that is to simply append a `.xml` to the end of any the URL strings in our browser, and Rails will serve us back the XML version of any of any of our resources. You can see the result of pulling back the list of activities in a specific workout at `http://localhost:3000/workouts/7/activities.xml` in Figure 6-8, but some other example URLs are

- `http://localhost:3000/exercises.xml`
- `http://localhost:3000/exercises/6.xml`
- `http://localhost:3000/goals.xml`
- `http://localhost:3000/goals/3/results/1.xml`

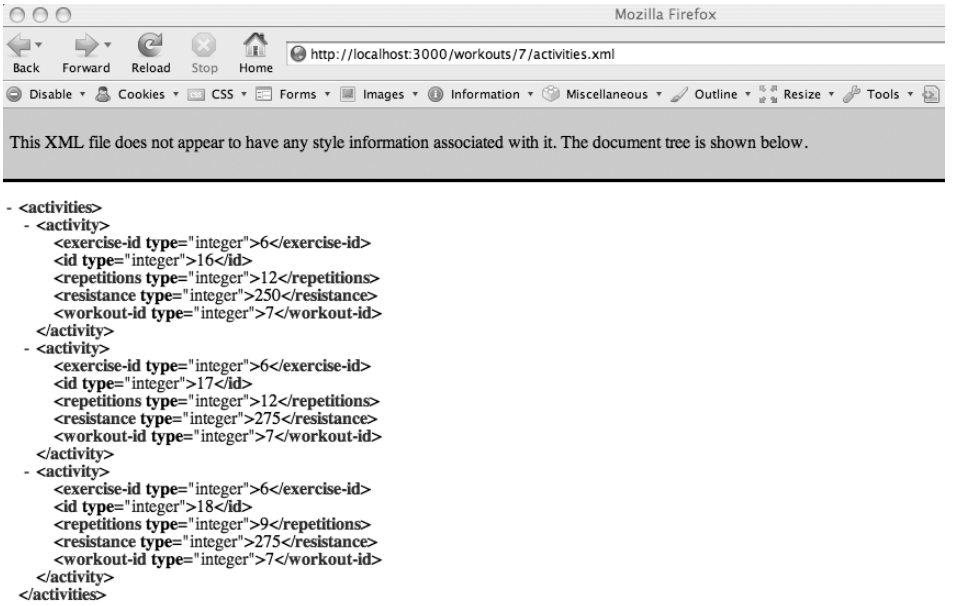

Figure 6-8. *Accessing the XML interface through a browser*

Using CURL to Interact with Our API

The limitation of using a web browser to experiment with your RESTful interface is that you're limited to mainly `GET` requests, so you can look but can't modify. Another alternative for interacting with your application that will allow you full access to read and modify your data is to use the command line utility `curl`.

Some common `curl` options that you should know are

- `-X [ ]`: Specifies the HTTP verb (i.e., `GET`, `POST`, `PUT` or `DELETE`)
- `-d [ ]`: Sets `POST` variables
- `-H [ ]`: Sets the content type
- `-U [ ]`: Sets the `username:password` for HTTP authentication

Using `curl`, you could access a list of all the exercises for a user ealameda (obviously, you'll need to use the username and password from the account that you created earlier) with the following command:

```
curl -X GET --Basic -u ealameda:test -H "Accept: text/xml" ➥
http://localhost:3000/exercises
```

```
<?xml version="1.0" encoding="UTF-8"?>
<exercises>
  <exercise>
    <exercise-type></exercise-type>
    <id type="integer">8</id>
    <name>Abdominals</name>
    <user-id type="integer">2</user-id>
  </exercise>
  <exercise>
    <exercise-type></exercise-type>
    <id type="integer">6</id>
    <name>Bench Press</name>
    <user-id type="integer">2</user-id>
  </exercise>
  <exercise>
    <exercise-type></exercise-type>
    <id type="integer">7</id>
    <name>Biceps Curl</name>
    <user-id type="integer">2</user-id>
  </exercise>
  <exercise>
    <exercise-type></exercise-type>
    <id type="integer">9</id>
    <name>Lat Pulldowns</name>
    <user-id type="integer">2</user-id>
  </exercise>
  <exercise>
    <exercise-type></exercise-type>
    <id type="integer">10</id>
    <name>Leg Press</name>
    <user-id type="integer">2</user-id>
  </exercise>
</exercises>
```

If you wanted to view the details of a specific exercise, you could simply specify it in the URL:

```
curl -X GET --Basic -u ealameda:test -H "Accept: text/xml" ➥
http://localhost:3000/exercises/6
```

```
<?xml version="1.0" encoding="UTF-8"?>
<exercise>
  <exercise-type></exercise-type>
  <id type="integer">6</id>
  <name>Bench Press</name>
  <user-id type="integer">2</user-id>
</exercise>
```

To create a new goal, however, we'll need to specify that we want to use the HTTP verb `POST` instead and use the `-d` parameter to set our post parameters:

```
curl -X POST --Basic -u ealameda:test -d "goal[name]=Daily Calories"
-d "goal[value]=1300" -H "Accept: text/xml" http://localhost:3000/goals
```

Once we've created the new goal, we can verify it's there by viewing the list of goals:

```
curl -X GET --Basic -u ealameda:test -H "Accept: text/xml" ➥
 http://localhost:3000/goals
```

```
<?xml version="1.0" encoding="UTF-8"?>
<goals>
  <goal>
    <id type="integer">3</id>
    <last type="decimal">250.0</last>
    <name>Weight Loss</name>
    <user-id type="integer">2</user-id>
    <value type="decimal">220.0</value>
  </goal>
  <goal>
    <id type="integer">4</id>
    <last type="decimal">110.0</last>
    <name>Blood Sugar (post lunch)</name>
    <user-id type="integer">2</user-id>
    <value type="decimal">100.0</value>
  </goal>
  <goal>
    <id type="integer">5</id>
    <last type="decimal"></last>
    <name>Daily Calories</name>
    <user-id type="integer">2</user-id>
    <value type="decimal">1300.0</value>
  </goal>
</goals>
```

AUTHENTICATION ERRORS?

At this time of this writing, a number of users were complaining of having issues authenticating through RESTful authentication and were proposing a workaround of making the following modifications to the login required method in `/lib/authenticated_system.rb`. Advocates of the modification suggest changing it from this:

```
def login_required
  username, passwd = get_auth_data
  self.current_user ||= User.authenticate(username, passwd) || :false ➥
if username && passwd
  logged_in? && authorized? ? true : access_denied
end
```

to this:

```
def login_required
  username, passwd = get_auth_data
  if self.current_user == :false && username && passwd
    self.current_user = User.authenticate(username, passwd) || :false
  end
  logged_in? && authorized? ? true : access_denied
end
```

You can read more about it at `http://www.railsweenie.com/forums/3/topics/1258` if you encounter similar issues.

Summary

In this chapter, we explored the basics of what RESTful applications are and why there's so much buzz about them within the Rails community. We even built our very first RESTful application in Rails using the `scaffold_resource` generator.

As a result, we now have a web application where we can enter all of our workout results. In the next chapter, we're going to develop another iteration of this application by further enhancing the interface design, integrating reporting capabilities using several popular Ruby graphing libraries, and adding some additional MIME types that can be called through the RESTful interface.