Practical REST on Rails 2 Projects

ISBN-13 (pbk): 978-1-59059-994-5

ISBN-10 (pbk): 1-59059-994-2

ISBN-13 (electronic): 978-1-4302-0655-2

ISBN-10 (electronic): 1-4302-0655-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Steve Anglin, Ben Renow-Clarke
Technical Reviewer: Bruce Williams
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: James A. Compton
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Dina Quan
Proofreader: Liz Welch
Indexer: Carol Burbo
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 4

Your First Clients: JavaScript and JSON

The preceding chapters have each served a different purpose: grounding you in the philosophy of RESTful system design, showing you the aspects of Rails that help you apply that philosophy, or leading you through the development of a functioning RESTful application. In this chapter, you'll begin your hands-on exploration of the virtues of an open, RESTful application by building JavaScript clients. You'll start by creating a simple, read-only widget to display information from MovieList, and then you'll move on to a more complicated client capable of both displaying and updating information. While these projects are simple, they suggest many of the techniques you'll be using in the more complex clients in the following chapters—so without further ado, on to the code!

The Widget Approach

The easiest way to get data from one web application to another is probably the simple JavaScript widget. These were first made possible when browser makers added support for the scripting language, and they were enhanced with the addition of support for the Document Object Model (DOM). Widgets are simply scripts embedded on a web page; when the page is loaded, the script calls out to a remote server to retrieve some data. Once the data is received, the script then inserts it into the page—either pushing it into an existing container or writing it directly into the page.

This is a powerful approach, allowing an application to request information from one or more remote servers and incorporate it directly into the pages that end users see. There is a significant limitation, however: these widgets are basically read-only. While there are various techniques you can use to achieve some limited interactivity with scripts of this sort, once the script has loaded, continued interaction with the data source is much more difficult.

Despite this problem, widgets are still a good first step toward accessing data from an open application. They are extremely easy to set up and can serve to familiarize you with issues you will see again later.

Planning

When building any client—from a simple JavaScript widget to a full-fledged ActiveResource Rails application—you should start by determining what data and functionality you want to

expose through it; when your server is RESTful, you can reframe that decision to address the *resources* and *methods* you want to allow the client to access.

Many factors may influence your decision, but two stand out: the nature of the technology used to build the client, and the sites on which the client will appear.

In our example, the major technical limitation at play is that these sorts of widgets are effectively read-only, unable to manipulate the data on the server. This means that of the standard RESTful actions, you'll only be able to provide `index` and `show`—no `create`, `update`, or `delete`.

As for the sites on which your MovieList widgets might appear, the most probable are general movie information sites (such as IMDB or Rotten Tomatoes) and the personal blog sites of people who love film. Because most of the host sites will already display abundant movie information, it doesn't make much sense to use the widget to display that. Instead, the most useful role for the widget would seem to be as an easy way to display upcoming releases. Specifically, you can easily build a widget that will show a comprehensive list of upcoming movie releases—or a list of only those upcoming releases in which a given MovieList user is interested. The widget, then, can become a way in which movie lovers can share their interests with visitors to their blogs.

Now that you have some idea of what you're going to build, you can take a few moments to check that the data and functionality (that is, the resources and methods) you'll be exposing are already available within your server's infrastructure. In general, you should be careful when building clients that don't map cleanly onto your existing RESTful interface, since they will require extra development effort on the server side. For the general-purpose widget, this is not a concern—it is essentially another view of the `ReleasesController#index` action. The user-specific widget, however, has no corresponding action on the MovieList application—unless you're willing to limit it to a user with an active MovieList session—so there'll be some work to get it running when you reach that point.

All Upcoming Releases

Once you know what functionality you'll be providing through the widgets, you can update the server application to support the plan. For the general-purpose version, you won't need to add any business logic; instead, you'll just need to create a new view and add code to the controller to ensure that the appropriate view is returned for a widget request.

On the view side, it's generally best to keep the markup simple and semantically correct. In this case, the widget will be providing a list of movies and release dates—so a standard ordered list should suffice (you could also make the argument that a definition list is most appropriate, but that's a topic for another book). Instead of creating this index view of new releases in HTML, however, you need to build it in JavaScript. Listing 4-1 shows the code.

Listing 4-1. *Listing releases in app/views/releases/index.js.erb*

```
var markup = '<ol id="movielist-releases">';

 <% @releases.each do |release| %>
  markup += '<li><%= h release.released_on.strftime('%m/%d/%Y') %> - ';
  markup += '<%= h release.movie.title %></li>';
<% end %>
```

```
markup += '</ol>';

document.write(markup);
```

Notice first that the filename for this new view is index.js.erb—as you saw back in Chapter 2, the "js" indicates that this view is JavaScript, while the "erb" ending triggers standard ERb processing—meaning that you can embed Ruby here just as you would on a standard Rails template. When this file is processed and sent back to the browser, then, it is executed as JavaScript, building a string representing an ordered list of releases and writing that out to the page.

With the view completed, you just need to make sure that any requests for the widget return this file instead of the more normal index.html.erb. Remembering Chapter 2 again, the filename should serve as a clue to the method you'll be using: if you can force the widget to request the index of releases via the JavaScript format, the application will automatically use your new view instead of the HTML one. The solution, then, is obviously `respond_to`, with the line added in Listing 4-2.

Listing 4-2. *Updating app/controllers/releases_controller.rb to return the JavaScript view*

```
class ReleasesController < ApplicationController
  # ...
  def index
    @releases = Release.paginate(:all, :page => params[:page])

    respond_to do |format|
      format.html # index.html.erb
      format.js
      format.xml  { render :xml => @releases }
    end
  end

  # ...
end
```

The only addition here is a `format.js` declaration within the `respond_to` block, which—since there is no block for it (unlike for `format.xml`)—will force the application to render your new index.js.erb view whenever a JavaScript-formatted request comes in.

To test this, create the HTML page shown in Listing 4-3, in your application's public directory.

Listing 4-3. *Testing the widget in public/general_widget.html*

```
<html>
<head><title>Test Widget</title></head>

<body>
<h1>Widget Test</h1>
```

```
<script type="text/javascript" src="/releases.js"></script>
</body>
</html>
```

■ **Note** In general, you should reference your server via an absolute path in your widget code—for example, `http://localhost:3000/releases.js` instead of the relative path `/releases.js`. Since the examples in this chapter are run out of your application's public directory, however, this is not needed.

You should now be able to visit `http://localhost:3000/general_widget` to see a display like Figure 4-1.

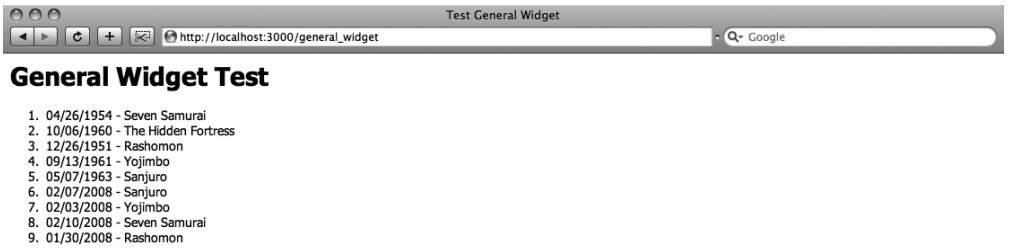

Figure 4-1. *Testing the JavaScript widget*

There are several problems with this as a list of upcoming releases, however. First and most obviously, many of the releases shown took place in the past (the page was accessed on February 1st, 2008). Second, there is no apparent order to the dates display—ideally, the widget would show them in descending date order. In addition to those noticeable issues, though, there is also a hidden problem. If you look at the MovieList logs for this request, you'll see that each movie was loaded in a separate call to the database. For such a small dataset, this isn't too limiting; if you were working with a much larger list, however, that inefficiency would quickly become painful.

The resolution of each of these issues lies in the controller, so open it up and make the changes shown in Listing 4-4.

Listing 4-4. *Updating app/controllers/releases_controller.rb*

```
class ReleasesController < ApplicationController
  # ...

  def index
    respond_to do |format|
      format.html {
        paginate_releases
      }
      format.js {
        @releases = Release.find(:all,
```

```
          :include => :movie,
          :order => 'released_on DESC',
          :conditions => ['released_on >= ?', Date.today]
        )
      }
      format.xml  {
        paginate_releases
        render :xml => @releases
      }
    end
  end

  # ...

  private
  def paginate_releases
    @releases = Release.paginate(:all, :page => params[:page])
  end
end
```

With these updates, the widget response uses an entirely different query to get its release list—instead of paginating over the entire set of releases as the HTML view does, the JavaScript view gets a list of upcoming releases sorted by descending release date. Furthermore, the query also loads each release's related movie record in the initial call to the database, so the multiplication of queries in the earlier version of the widget is avoided. Figure 4-2 shows the result of these updates.

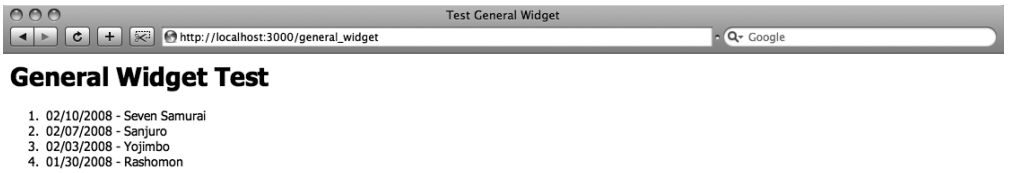

Figure 4-2. *The revised widget test*

The general-purpose widget, then, works. The next step is to get the user-specific version up and running—and for that, you'll have to do a little more work.

Releases for a User

The specific-user version of the widget requires you to add something new to MovieList itself. At present, a user can view the upcoming releases she is interested in by browsing to `/user/notifications`; you set this up in the previous chapter by creating a singleton user resource that `has_many` notifications nested within it.

To get to the point where any user's notifications are visible, you need to make a URL like `/users/[user_id]/notifications` work—and for that, you need to go back to your routes, as shown in Listing 4-5.

Listing 4-5. *Adding routes by hand to config/routes.rb*

```
ActionController::Routing::Routes.draw do |map|
  map.resource  :session
  map.resource  :user, :has_many => [:interests, :notifications]
  map.connect   'user/:user_id/notifications', ➥
    :controller => 'notifications', :action => 'index'
  map.connect   'user/:user_id/notifications.:format', ➥
    :controller => 'notifications', :action => 'index'

  # ...
end
```

These new routes allows you to access the `NotificationsController#index` action both through the user singleton resource and through a custom, unnamed route with a `user_id` parameter—with or without a format specified.

■ **Caution** This custom route looks much like the result of nesting notifications under the standard user resource, with

```
map.resources :users, :has_many => :notifications
```

If you try this, though, your routes will throw an exception, since both nestings will attempt to register the same named routes.

Of course, the route comprises only part of the required changes. You also need to update the notifications controller to behave appropriately when it's accessed through the new method, as shown in Listing 4-6.

Listing 4-6. *Updating app/controllers/notifications_controller.rb for JavaScript access*

```
class NotificationsController < ApplicationController
  before_filter :require_login_or_user

  def index
    @releases = @user.releases(true)
    respond_to do |format|
      format.html
      format.js { render :template => 'releases/index' }
    end
 end

  private
  def require_login_or_user
    if params[:user_id]
      @user = User.find_by_id(params[:user_id])
```

```
    elsif logged_in?
      @user = current_user
    else
      access_denied
    end
  end
end
```

Most of this controller is new; first, the `require_login` filter has been replaced to permit public access to the `index` action—though if you aren't logged in, you will have to specify a `user_id` (that's taken care of by the conditional in the `require_login_or_user` method). The instance variable `@user` will be set based on whether you pass in a `user_id` parameter or are logged in, and it will be used to retrieve the release list for eventual display. There's also a new `respond_to` block that sends HTML responses to `app/views/notifications/index.html.erb`; JavaScript requests, however, get `app/views/releases/index.js.erb`, instead.

The only other change here is the value `true` being passed to `@user.releases`, which means you'll have to update your user model, too (Listing 4-7).

Listing 4-7. *Updating the releases method in app/models/user.rb*

```
class User < ActiveRecord::Base
  def releases(upcoming_only = false)
    movie_ids = movies.map(&:id)
    if upcoming_only
      conditions = [
        "released_on >= ? AND movie_id IN (?)",
        Date.today,
        movie_ids
      ]
    else
      conditions = [
        "movie_id IN (?)",
        movie_ids
      ]
    end

    Release.find(:all,
      :include => :movie,
      :conditions => conditions,
      :order => 'released_on DESC')
  end

  # …
end
```

Here, you're changing the conditions on the query based on the parameter passed into the `releases` method. If you pass in `true`, the system will only pull back upcoming releases; if you send `false` instead, it will return all releases.

All of this together enables you to access the upcoming releases for a given user's interests either through the web interface (at `/users/[user_id]/notifications`) or via JavaScript, just as you did for the general-purpose widget before. To see this in action, create another file in your public folder, as shown in Listing 4-8.

Listing 4-8. *Testing the user-specific widget in public/user_widget.html*

```
<html>
<head><title>Test User Widget</title></head>

<body>
<h1>User Widget Test</h1>
<script type="text/javascript" src=" /users/1/notifications.js">
</script>
</body>
</html>
```

And that (assuming you have a user with the ID 1) should result in something like the page shown in Figure 4-3.

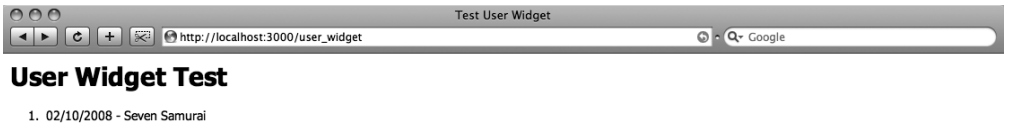

Figure 4-3. *The user-specific widget*

That just about does it for the widgets. As I said earlier, these sorts of projects are easy; to distribute them, you just embed the script tags from general_widget.html and user_widget.html in whatever pages you'd like to have the data—it's that simple.

Nothing this easy is ever perfect, however—and indeed, there are some noticeable problems with the widget approach, as we'll see.

Widget Problems

The most significant problem you'll see with this approach is accessibility; the widgets depend entirely on JavaScript to function, and so they will not work if a user without JavaScript visits the page. There is a wide range of reasons people browse without JavaScript—some by choice, others by necessity. If your audience has a significant number of non-JavaScript users, you may want to reconsider using widgets for any necessary information.

Beyond the accessibility concerns, however, there are other potential problems. In particular, some issues may arise because your widgets are injecting markup into a page over which you may have little or no control. This means that the markup you send may end up rendering in a way that you never expected. For instance, the page on which your widget lives might have a stylesheet that already has declarations related to the ordered list markup you created earlier, resulting in something like Figure 4-4.

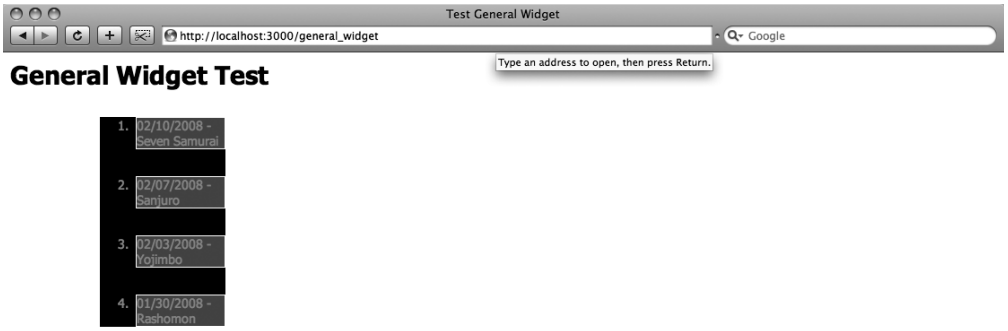

Figure 4-4. *CSS conflicts with the widget approach*

This is an extreme example, of course; in most cases, having your widget inherit the local styles for the pages on which it appears is a benefit, since they will help it to match the look and feel of its surroundings. If you tend to complicate your markup, however, you run a very real risk of conflict.

The best way to minimize these potential difficulties is to keep your widget's markup as simple as possible. In the widgets you just built, you're injecting a simple ordered list. If by some chance even that simple markup breaks a page, keeping to the basics will make it easier to fix for the developers on the receiving side—and you can make it even easier for them by providing appropriate IDs (such as `movielist-releases` on the OL tag), which they can use to create custom CSS rules to handle your widget's markup directly.

Aside from the layout issues that may come up, you might also see more general styling problems. In general, you have two main means of specifying the styles your widget uses: first, you can send them inline, as in Listing 4-9.

Listing 4-9. *Setting inline styles in app/views/releases/index.js.erb*

```
var markup = '<ul id="movielist-releases" style="font-size:14px;">';

<% @releases.each do |release| %>
  markup += '<li style="color:blue;font-weight: bold;">';
  markup += '<%= h release.released_on.to_s(:short) %> - ';
  markup += '<%= h release.movie.title %></li>';
<% end %>

markup += '</ul>';

document.write(markup);
```

Alternatively, you can include a stylesheet reference with the markup your widget generates, as in Listing 4-10.

Listing 4-10. *Specifying a stylesheet in app/views/releases/index.js.erb*

```
var markup = '<link href="[stylesheet URL]" media="screen" ';
markup +=   'rel="stylesheet" type="text/css" />';
markup += '<ul id="movielist-releases">';

<% @releases.each do |release| %>
  markup += '<li><%= h release.released_on.to_s(:short) %> - ';
  markup += '<%= h release.movie.title %></li>';
<% end %>

markup += '</ul>';

document.write(markup);
```

Both approaches, however, have problems. Either may conflict with or interact unpredictably with the local styles defined on the surrounding page, and the stylesheet approach in particular may override styles outside the scope of the widget. If you have the option, you're generally better off sending as little style information with the widget as possible, relying instead on the receiving developer to style the output as she likes.

The widget approach, then, has both advantages and disadvantages. It is easy to set up and to distribute, requiring very little effort on the end developer's part to get running, but it has some noticeable shortcomings in accessibility and interactivity. Widgets are not the only option available with JavaScript, however, as you'll see in the next section.

A JSON Client

I've mentioned briefly that you can provide some interactivity through a widget; in general, however, widgets are best for providing read-only views into your data. When you want to give end users the ability to manipulate that data, you need to turn elsewhere. JavaScript (which is, after all, the J in Ajax) does provide other capabilities for that sort of project. In this section, you'll be building an Ajax client to interact with MovieList, and you'll be using JSON to pass data between the server and the client.

JSON stands for JavaScript Object Notation; it provides a simple, human-readable representation for structured data, including objects. Over the last few years, it has gained prominence as an alternative to XML for transferring data between applications. As you'll see shortly, JSON is built into Rails, making it an excellent choice for the JavaScript client you'll be building shortly.

MORE ABOUT JSON

JSON provides the standard data types:

- `null`
- `boolean`
- `numeric`
- `string`
- `array`
- `object`

Objects in JSON are simply key/value pairs enclosed in curly braces. The following is a possible JSON representation of a movie object from MovieList:

```
{
  "id": 18,
  "title": "Rashomon",
  "description": "Rashomon (Rashÿmon) is a 1951 Japanese film…",
  "rating": "",
  "releases": [
    {
      "format": "theater",
      "released_on": "Wed, Dec 26 1951"
    }
  ]
}
```

Since JSON is a subset of JavaScript, this representation is valid JavaScript code. As a result, you can instantiate a native JavaScript object by simply passing it through `eval`, and once you have decoded the objects you can manipulate them however you like. Of course, `eval` can be a security concern, but there are ways to avoid problems—the JSON library available at `http://www.json.org/json2.js`, for instance, includes a safer alternative in the `parseJSON` method.

In addition, JSON is a valid subset of YAML, making it a simple matter for many programming languages (including Ruby) to translate JSON-formatted strings into native objects. ActiveSupport::JSON holds the key methods for working with JSON in Rails, including `#to_json` and `#from_json`.

Planning

The planning phase for the JSON client is a bit more open that it was for the widgets. The client, after all, allows for interactions that weren't possible with the read-only scripts. For instance, you can actually update, send data to, and manipulate data on the server. It may not make sense to allow full administrative privileges through the client—no adding or updating movie records—but it would certainly be feasible to allow users to manage their interests.

What's more, with interest management in particular the general security issues are minimized; if you recall, interests (as notifications were before you tweaked them in the previous project) are scoped to the current user via the user singleton resource. This means that any user accessing the client will be unable to affect anything outside of her own interests—and she won't even be able to do that unless she has an active MovieList session. Of course, this means that you'll have to handle unauthenticated users accessing the client, but that will come up later.

Interest management as it currently stands consists of the `index`, `create`, and `destroy` actions. Basically, you can view your interests, add a new one, or remove an existing one. Those functions map to the following URIs and HTTP request methods:

- `GET /user/interests` to list a user's interests
- `POST /user/interests` to add a new interest
- `DELETE /user/interests/[interest_id]` to remove an interest

This framework can be reused directly for the client you'll be building, and (as in the main web interface) you won't need separate `new`, `edit`, or `update` actions—there's no sensible way to update an interest, after all, and you can incorporate the new interest form into the index view.

Implementation

At first glance, creating the JSON interface may be intimidating. Upon a closer look, however, it turns out that the JSON API is very similar to the XML API automatically generated by the scaffolding, so you can use that as a basic template. Of course, you built the interests functionality by hand back in Chapter 3, so there isn't any scaffolded code in that part of the application currently—but it's easy enough to generalize the necessary work from those sections that you did generate via scaffolding.

The first step is to update the `InterestsController#index` action to handle JSON requests. Just as with the widget, you do this by adding a `respond_to` block, as shown in Listing 4-11.

Listing 4-11. *Updating app/views/controllers/interests_controller.rb for the JSON client*

```
class InterestsController < ApplicationController
  # ...

  def index
    @interests = current_user.interests
    @interest = Interest.new

    respond_to do |format|
      format.html
      format.json { render :json => @interests }
    end
  end

  # ...
end
```

This is fairly standard code, exactly parallel to what you might see for the XML API in your movies controller; the only exception is the `render :json` call. Basically, `render :json` returns a JSON representation of the Ruby object that it is given. In this case, the application calls `@interests.to_json` implicitly, returning a JSON string containing an array of Interest objects, each with all its attributes. (You get the same behavior in the generated scaffold code with `render :xml`.)

As you saw in Chapter 2, Rails 2 allows you to add custom response formats if you need to; luckily, however, Rails already knows the JSON format and maps it to `text/x-json`. This means that you're done updating the MovieList code—there's no need to create a view, since you're rendering a JSON string from the action. The next step, then, is to look at the client. The first thing to do is build a simple page (Listing 4-12) so that you can test your code.

Listing 4-12. *Creating public/json_client.html*

```
<html>
<head>
<title>JSON Client</title>
<script type="text/javascript" src="/javascripts/prototype.js"></script>
<script type="text/javascript">
// JSON processing will go here
</script>
</head>
<body>

<div id="target">
nothing yet!
</div>

</body>
</html>
```

Notice that this page includes the Prototype JavaScript library. Prototype itself isn't required; you could do everything here with straight JavaScript, or with another library like jQuery—but given the library's familiarity to Rails developers and the helper methods it includes for working with JSON, it is a good choice to make the examples cleaner and clearer. In addition, you have easy access to it by running your client from within your Rails application.

Now that you have a test page, you can add the code to retrieve your MovieList interests. Start by creating a new Ajax request and capturing the JSON it returns. Then, you can instantiate JavaScript objects and build whatever markup you (as the client developer) deem appropriate, as shown in Listing 4-13.

Listing 4-13. *Requesting data in public/json_client.html*

```
...
<script type="text/javascript">
// JSON processing will go here
new Ajax.Request('/user/interests.json', {
```

```
  method:'get',
  onSuccess: function(data){
    var interests = data.responseText.evalJSON();
    var markup = '<ul>';
    interests.each(function(interest) {
      markup += '<li>' + interest.movie_title + '</li>';
    })
    markup += '</ul>';
    $('target').update(markup)
  }
});
</script>
</body>
</html>
```

Notice that this code doesn't use `eval` directly. The `evalJSON()` method is built into Prototype, and it provides some small measure of security beyond that of `eval` directly (much like the `parseJSON` method in the library mentioned earlier)—in particular, you can call it with the argument `true` (e.g., `evalJSON(true)`) to detect and defuse potentially dangerous code.

If you're following along with the code and try this out, you may be confused when nothing happens. It turns out that `interest.movie_title` doesn't actually work yet. Remember that `render :json` calls `#to_json` behind the scenes, translating ActiveRecord objects into JSON-formatted strings. By default, however, `#to_json` only includes *attributes* in the resulting string—and in this case, we need the output of a method (`movie_title`). Luckily, it's a quick fix. First we add the method to the interest model, in Listing 4-14.

Listing 4-14. *Adding a convenience method to app/models/interest.rb*

```
class Interest < ActiveRecord::Base
  # …

 def movie_title
    movie.title
  end
end
```

Then we update the controller so that the results of that method are included in the JSON string, in Listing 4-15.

Listing 4-15. *Updating app/views/controllers/interests_controller.rb*

```
class InterestsController < ApplicationController
  # ...

  def index
    @interests = current_user.interests

    respond_to do |format|
      format.html
```

```
      format.json { render :json => @interests.to_json(
        :methods => [:movie_title]
      ) }
    end
  end

  # ...
end
```

Here, the implicit call to `#to_json` has been abandoned in favor of an explicit call with a customized options hash, through which you can specify the attributes and methods you want serialized (check the Rails documentation for the full suite of options). With this change, the JSON client as written works—or at least, the index view does, as illustrated in Figure 4.5.

Figure 4-5. *The JSON client in action*

Unauthenticated Users

At this point, however, you may encounter another problem. Remember that the interests controller scopes all of its actions to the current user. This means that your JSON client will only work if the user visiting the client has already logged in to MovieList. This by itself isn't an issue (actually, it serves as an extra security check—preventing just anyone from editing a MovieList user's interests), but it does mean that you need to write some code to handle the case where someone who *isn't* logged in visits the client.

In the web interface for MovieList, unauthenticated users who visit the `Interests-Controller#index` action hit the `login_required` filter and are redirected to the login page. When the request comes in via Ajax, however, they don't get redirected. Instead, such visitors get a somewhat obscure HTTP status code: 406 Not Acceptable.

There are two general approaches to handling this error. The first is to ignore it; you can set the initial contents of the target div to a message suitable for the condition, such as "Please log in at MovieList to continue." When the client receives a 406 error, the target div will retain its default contents, and the specified message will remain in place unless an authenticated user visits.

The alternative (and more responsible) approach is to make use of one of the more interesting features of Prototype: status code-specific callback functions. With both `Ajax.Request` and `Ajax.Updater`, you can specify callback functions for any HTTP status code you wish. Most relevant for this case, you can specify the callback for 406 responses, as shown in Listing 4-16.

Listing 4-16. *Handling unauthenticated users in public/json_client.html*

```
...
<script type="text/javascript">
// JSON processing will go here
new Ajax.Request('/user/interests.json', {
```

```
  method:'get',
  onSuccess: function(data){
    var interests = data.responseText.evalJSON();
    var markup = '<ul>';
    interests.each(function(interest) {
      markup += '<li>' + interest.movie_title + '</li>';
    })
    markup += '</ul>';
    $('target').update(markup)
  },
  on406: function(data){
    $('target').update('Please log in at MovieList to continue');
  }
});
</script>
</body>
</html>
```

This code simply displays the specified error message when a 406 is returned from MovieList, but in a more elaborate client application you could easily use that callback to present a login form, or to handle the situation in some other way. This technique also allows you to distinguish between users who haven't logged in to MovieList and those who have JavaScript turned off; the latter will always see the default contents of the target div, so you can put a message specific to them in it on page load.

If you aren't using Prototype, you can handle this error by registering a callback function with the more general `onFailure` event—it is less specific than Prototype's status code callbacks, but it works perfectly well for this purpose.

Adding an Interest

At this point, the JSON client is still little better than the earlier widgets you built. With the basic technique familiar, however, you can now start adding some interactivity. The first management function to build is the ability to create a new interest. For that, you need a form on the client page. You could add the form dynamically—by injecting it into the page when the user clicks a link, for instance—but in the interest of keeping things at least somewhat simple, the code shown in Listing 4-17 adds it as a persistent feature of the interest listing.

Listing 4-17. *Updating public/json_client.html with the new form*

```
...
<script type="text/javascript">
// JSON processing will go here
new Ajax.Request('/interests.json', {
  method:'get',
  onSuccess: function(data){
    var interests = data.responseText.evalJSON();
    displayInterestList(interests);
  },
```

```
  on406: function(data){
    $('target').update('Please log in at MovieList to continue');
  }
});

function submitForm() {
  new Ajax.Request('/user/interests.json', {
    asynchronous: true,
    evalScripts: true,
    parameters: Form.serialize('interest-form'),
    onSuccess: function(data){
      var interests = data.responseText.evalJSON();
      displayInterestList(interests);
    }
  });
}

function displayInterestList(interests) {
  var markup = '<ul>';
  interests.each(function(interest) {
    markup += '<li>' + interest.movie_title + '</li>';
  })
  markup += '</ul>';

  markup += '<form method="post" id="interest-form" ';
  markup += 'onsubmit="submitForm(); return false;" ';
  markup += 'action="/user/interests.json">';
  markup += 'Add a Movie: ';
  markup += '<input type="text" name="interest[movie_title]" />';
  markup += '<input type="submit" value="Save" />';
  markup += '</form>';

  $('target').update(markup)
}
</script>
</body>
</html>
```

Figure 4-6 shows the display this code produces on the page in which it is embedded.

Figure 4-6. *The JSON client with the interest creation form*

At the point, then, anything you enter in the form is submitted via Ajax to your MovieList application; the server then replies with the updated list of interests and redisplays them just as it did for the `index` action—in fact, the display is identical, so the script in Listing 4-17 has been refactored to use a single `displayInterestListing` method.

With that, the client is complete; the MovieList code, however, is not yet set up to handle interest creations via Ajax. The next step, then, is to go back to the controller (Listing 4-18).

Listing 4-18. *Updating create to accept JSON requests in app/controllers/interests_controller.rb*

```
class InterestsController < ApplicationController
  # ...

  def create
    current_user.interests.create(params[:interest])
    flash[:notice] = 'You have added an interest in the specified movie'

    respond_to do |format|
      format.html { redirect_to user_interests_path }
      format.json { render({
        :json => current_user.interests.reload.to_json(
          :methods => [:movie_title]
        )
      }) }
    end
  end

  # ...
end
```

The only difference between this and the `index` action code is the addition of `reload` to `current_user.interests`—this is necessary to ensure that the list returned to the client is updated with the newly added interest. As it stands, however, this still doesn't quite work. Note that the client form here submits a field named `interest[movie_title]`—to use that value while creating an interest, you need to add a new method to the interest model (Listing 4-19).

Listing 4-19. *Adding a convenience method to app/models/interest.rb*

```
class Interest < ActiveRecord::Base
  # ...

  def movie_title=(title)
    self.movie = Movie.find_by_title(title)
  end
end
```

And with that, you can easily create new interests via the client, as illustrated in Figure 4-7.

Figure 4-7. *Creating an interest via the JSON client*

Error Handling

Of course, the movie title field in the client form is a free-form text field, making it possible for users to type in *anything* as an interest. This means that you have to deal with entry errors (misspellings, etc.) and with users attempting to add interests in movies that aren't yet in the system. The appropriate response (short of adding such movies on the fly) is to display some message when such errors occur, and for that you'll need to modify both the server and the client, to catch the error and display the message.

■ **Note** There is an easy way to cut down on the number of errors generated by this field; you could add autocompletion to it, so that when users start to type in a movie's title, the system automatically looks up titles that match the text entered and allows the user to select from among them. The autocompletion helper that was built into Rails previously, however, has been extracted to a plugin in Rails 2; you can install it from `http://dev.rubyonrails.org/browser/plugins/auto_complete`.

In your controller, the easiest thing to do is to check whether the interest record was successfully saved by the create attempt, as highlighted in Listing 4-20.

Listing 4-20. *Returning different status codes in app/controllers/interests_controller.rb*

```
class InterestsController < ApplicationController
  # ...

  def create
    interest = current_user.interests.create(params[:interest])

    respond_to do |format|
      format.html { redirect_to user_interests_path }
      format.json {
        status_code = interest.new_record? ? 422 : 201
        render :json => current_user.interests.reload.to_json(
          :methods => [:movie_title]
        ), :status => status_code }
    end
  end

  # ...
end
```

With this update, MovieList will send the HTTP status code 201 ("created") if the interest was created successfully, or 422 ("unprocessable entry") if the creation failed. Your client is already set up to handle the 201—it just has to redisplay the interest list and form; all that remains, then, is to update it to correctly handle 422, as in Listing 4-21.

Listing 4-21. *Handling creation failures in public/json_client.html*

```
<script type="text/javascript">
// ...

function submitForm() {
  new Ajax.Request('/user/interests.json', {
    asynchronous: true,
    evalScripts: true,
    parameters: Form.serialize('interest-form'),
    onSuccess: function(data){
      var interests = data.responseText.evalJSON();
      displayInterestList(interests);
    },
    on422: function(data){
      var interests = data.responseText.evalJSON();
      displayInterestList(interests,
        "The movie title you entered was not found. ➥
Please try again.");
    }
  });
}

function displayInterestList(interests, message) {
  var markup = '<ul>';
  interests.each(function(interest) {
    markup += '<li>' + interest.movie_title + '</li>';
  })
  markup += '</ul>';

  if (message) {
    markup += '<p>' + message + '</p>';
  }

  markup += '<form method="post" id="interest-form" ';
  markup += 'onsubmit="submitForm(); return false;" ';
  markup += 'action="/user/interests.json">';
  markup += 'Add a Movie: ';
  markup += '<input type="text" name="interest[movie_title]" />';
  markup += '<input type="submit" value="Save" />';
  markup += '</form>';
```

```
  $('target').update(markup)
}
</script>
```

Figure 4-8 shows the result of this modification; a title that cannot be used to create an interest will generate a message on the interest list, allowing users to try again (if, for instance, they misspelled the title).

Figure 4-8. *A failed attempt to create an interest*

And with that, this JSON client is at least usable—you can view your MovieList interests and add to them. Of course, if you play around with this long enough, you'll eventually want to remove some of the interests (did you *really* want to be notified any time one of those Keanu Reeves movies is rereleased on DVD?). The natural next step, then, is to add the ability to remove interests through the client.

Removing an Interest

Like adding an interest, deleting an interest will require changes both to the server and the client. Starting with the controller again, the server needs to return the list of interests after an interest is removed. Listing 4-22 shows the code.

Listing 4-22. *Handling JSON deletion requests in app/controllers/interests_controller.rb*

```
class InterestsController < ApplicationController
  # ...

  def destroy
    interest = current_user.interests.find(params[:id])
    interest.destroy if interest

    respond_to do |format|
      format.html { redirect_to user_interests_path }
      format.json {
        render :json => current_user.interests.reload.to_json(
          :methods => [:movie_title]
        )
      }
    end
  end
end
```

This works just like the final version of the `create` action—when a JSON request comes in, the specified interest is removed, and the system returns a JSON-formatted string of the current user's remaining interests.

The next step is to add a mechanism to access this action from the client. In keeping with the traditional scaffolding, the code in Listing 4-23 will add links that submit Ajax requests to delete the associated records.

Listing 4-23. *Adding delete functionality to public/json_client.html*

```
<script type="text/javascript">
// ...

function removeInterest(id) {
  new Ajax.Request('/user/interests/' + id + '.json', {
    method: 'post',
    asynchronous: true,
    evalScripts: true,
    parameters: '_method=delete',
    onSuccess: function(data){
      var interests = data.responseText.evalJSON();
      displayInterestList(interests);
    }
  });
}

function displayInterestList(interests, message) {
  var markup = '<ul>';
  interests.each(function(interest) {
    markup += '<li>' + interest.movie_title;
    markup += ' <a href="#" onclick="removeInterest(' + interest.id;
    markup += ');return false;">remove</a></li>';
  })
  markup += '</ul>';

  if (message) {
    markup += '<p>' + message + '</p>';
  }

  markup += '<form method="post" id="interest-form" ';
  markup += 'onsubmit="submitForm(); return false;" ';
  markup += 'action="/user/interests.json">';
  markup += 'Add a Movie: ';
  markup += '<input type="text" name="interest[movie_title]" />';
  markup += '<input type="submit" value="Save" />';
  markup += '</form>';
```

```
  $('target').update(markup)
}
</script>
```

There are a couple of things to note about this code. First, the removal link gets the ID of the interest object—as you saw in the planning section, the delete link has to submit to the URI `/user/interests/[interest_id]`. The HTTP method being used is also important; the Ajax request itself is being `POST`ed to the server, but look carefully at the parameters. The client is sending `_method=delete`, which (as you saw in Chapter 2) leads the MovieList to treat the request as if it came in as a `DELETE`. This combination of URI and request method routes the request to the appropriate `InterestsController#destroy` action; without either piece, this routing would fail. As it stands, however, it works—try it out on the client you've been working with, and finally get rid of that interest in *Ishtar* you've been hiding.

It is, of course, possible for a delete to fail, and you could write error-handling code to add a message to the page if that happened; the code would look much like that used in the `create` action. This is a fairly rare occurrence, however, and the code doesn't serve any additional instructional purpose at this point, so I'll skip it. Even without that error handling, though, your JSON client now fulfills the goals set out during planning. You can use it to view your MovieList interests, add new ones, and remove old ones. The completed JavaScript code for the JSON client can be seen in Listing 4-24.

Listing 4-24. *The completed public/json_index.html*

```
<html>
<head>
<title>JSON Client</title>
<script type="text/javascript" src="/javascripts/prototype.js"></script>
<script type="text/javascript">
// JSON processing will go here
new Ajax.Request('/user/interests.json', {
  method:'get',
  onSuccess: function(data){
    var interests = data.responseText.evalJSON();
    displayInterestList(interests);
  },
  on406: function(data){
    $('target').update('Please log in at MovieList to continue');
  }
});

function submitForm() {
  new Ajax.Request('/user/interests.json', {
    asynchronous: true,
    evalScripts: true,
    parameters: Form.serialize('interest-form'),
    onSuccess: function(data){
      var interests = data.responseText.evalJSON();
      displayInterestList(interests);
```

```
    },
    on422: function(data){
      var interests = data.responseText.evalJSON();
      displayInterestList(interests,
        "The movie title you entered was not found. Please try again.");
    }
  });
}

function removeInterest(id) {
  new Ajax.Request('/user/interests/' + id + '.json', {
    method: 'post',
    asynchronous: true,
    evalScripts: true,
    parameters: '_method=delete',
    onSuccess: function(data){
      var interests = data.responseText.evalJSON();
      displayInterestList(interests);
    }
  });
}

function displayInterestList(interests, message) {
  var markup = '<ul>';
  interests.each(function(interest) {
    markup += '<li>' + interest.movie_title;
    markup += ' <a href="#" onclick="removeInterest(' + interest.id;
    markup += ');return false;">remove</a></li>';
  })
  markup += '</ul>';

  if (message) {
    markup += '<p>' + message + '</p>';
  }

  markup += '<form method="post" id="interest-form" ';
  markup += 'onsubmit="submitForm(); return false;" ';
  markup += 'action="/user/interests.json">';
  markup += 'Add a Movie: ';
  markup += '<input type="text" name="interest[movie_title]" />';
  markup += '<input type="submit" value="Save" />';
  markup += '</form>';

  $('target').update(markup)
}
</script>
</head>
<body>
```

```
<div id="target">
  nothing yet!
</div>

</body>
</html>
```

Testing

The last thing to discuss in this chapter is a topic that has been missing (perhaps glaringly so) from these projects so far: testing. In general, you won't see many tests in the projects in this book (though you will, of course, see them in the downloadable application code); the testing techniques used are relatively common and are easy to pick up; in many cases, you can do so simply by reading through the tests generated by Rails 2's scaffolding. There is at least one aspect of testing for the widgets and client in this chapter, however, that *isn't* covered in the generated code. The autogenerated tests are noticeably lacking anything dealing with alternate formats; the generated XML API, for instance, is entirely untested.

Given that you've just written a fair amount of code to deal with a couple of new formats (`js` and `json`), it makes sense to see what some of the tests for those formats should look like. Listing 4-25, then, is an excerpt from the InterestsControllerTest class.

Listing 4-25. *Testing formatted requests in test/functional/interests_controller_test.rb*

```
class InterestsControllerTest < Test::Unit::TestCase
  # ...

  def test_json_index_should_return_json_string
    user = User.find(1)
    user.interests.delete_all
    movie = Movie.find(1)
    user.interests.create(:movie => movie)

    get :index, {:format => 'json'}, {:user => 1}
    raw = @response.body
    decoded = ActiveSupport::JSON.decode(@response.body)

    assert raw.is_a?(String)
    assert decoded.is_a?(Array)
    assert decoded.first.has_key?('movie_id')
    assert_equal 1, decoded.first['movie_id']
  end
end
```

This test case passes when a request to GET `/interests` for the user ID 1 returns a JSON string that contains an array of hashes, each of which describes an Interest object. In this particular case, the resulting array has a single member, and the interest it represents is the Movie object with ID 1.

The most important thing to note in this is that you can test your `respond_to` code by passing a `:format` parameter to your `get/post/put/delete` method—and make sure you specify the format as a string, not as a symbol, or it won't be recognized.

Further Projects

As I mentioned in the last chapter, each of these project chapters will end with suggestions for potential further development. With these JavaScript widgets and clients, you have a multitude of possibilities to build on the things you've done.

For instance, you've already seen an example of how the JSON client can avoid some of the accessibility problems that the JavaScript widget had (by showing a default message when JavaScript is disabled). You could take the time to enhance the widget to use JSON, reaping the accessibility benefit and potentially avoiding many of the styling and layout issues—this approach requires significantly more work on the part of the client developer, since she has to integrate raw JavaScript objects into her page, but it is more effective in the long run and allows for better error-handling and presentation.

Given your experience with the JSON client, it should be simple to see how this revised widget would work. First, you'd delete the `index.js.erb` view. Then, you would replace the `format.js` block in the controller with a block for `format.json`, as in Listing 4-26.

Listing 4-26. *Updating the JavaScript widget to use JSON in app/controllers/releases_controller.rb*

```
class ReleasesController < ApplicationController
  # ...
  def index
    @releases = params[:user_id] ?
      User.find(params[:user_id]).releases.find_upcoming :
      Release.find_upcoming

    respond_to do |format|
      format.html # index.html.erb
      format.json { render :json => ➥
@releases.to_json(:include => :movie) }
      format.xml  { render :xml => @releases }
    end
  end
  # ...
end
```

Finally, you'd have to change how you distributed the widget. Instead of embedding a simple script tag on the page, client developers would have to customize callback functions to generate the appropriate markup for their sites, as in Listing 4-27.

Listing 4-27. *The JSON widget, in public/json_release_widget.html*

```
<script type="text/javascript" src="javascripts/prototype.js"></script>
<script type="text/javascript">
new Ajax.Request('/releases.json', {
```

```
  method:'get',
  onSuccess: function(data){
    var releases = data.responseText.evalJSON();

    var markup = '<ul>';
    releases.each(function(release) {
      markup += '<li>' + release.released_on;
      markup += ' - ' + release.movie.title + '</li>';
    })
    markup += '</ul>';

    document.write(markup);
  }
});
</script>
```

Similarly, you could revise the user-specific version of the widget by changing the URL from `releases.json` to `users/[user id]/notifications.json`, leaving everything else the same.

You could also spend some time expanding the functionality of the JSON client itself. A good first step, for instance, would be to generate a login form in the 406 error handling code, allowing unauthenticated users to log in directly from the client. You might also add notifications to the display, or movies and their releases—really, the sky's the limit. The JSON client is as flexible as MovieList itself is.

Summary

In this chapter you've built your first MovieList clients—from a couple of simple, read-only widgets to an interactive tool through which you can fully manage your MovieList interests from a third-party site. These clients, though, are limited; they run entirely in the browser. In the next chapter, you'll be building a client meant to live on another server, in a language that is (to say the least) dissimilar to Ruby: PHP. When you're ready to tackle that, read on.