

Extracted from:

# Agile Web Development with Rails

---

Second Edition

This PDF file contains pages extracted from Agile Web Development with Rails, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragmaticprogrammer.com>.

**Note:** This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2007 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

In this chapter, we'll see

- using partial templates
- rendering into the page layout
- updating pages dynamically with AJAX and rjs
- highlighting changes with Script.aculo.us
- hiding and revealing DOM elements
- working when JavaScript is disabled

## Chapter 9

# Task D: Add a Dash of AJAX

---

Our customer wants us to add AJAX support to the store. But just what *is* AJAX?

In the old days (up until a year or two ago), browsers were treated as really dumb devices. When you wrote a browser-based application, you'd send stuff down to the browser and then forget about that session. At some point, the user would fill in some form fields or click a hyperlink, and your application would get woken up by an incoming request. It would render a complete page back to the user, and the whole tedious process would start afresh. That's exactly how our Depot application behaves so far.

But it turns out that browsers aren't really that dumb (who knew?). They can run code. Almost all browsers can run JavaScript (and the vast majority also support Adobe's Flash). And it turns out that the JavaScript in the browser can interact behind the scenes with the application on the server, updating the stuff the user sees as a result. Jesse James Garrett named this style of interaction *AJAX* (which once stood for *Asynchronous JavaScript and XML* but now just means *Making Browsers Suck Less*).

So, let's AJAXify our shopping cart. Rather than having a separate shopping cart page, let's put the current cart display into the catalog's sidebar. Then, we'll add the AJAX magic that updates the cart in the sidebar without redisplaying the whole page.

Whenever you work with AJAX, it's good to start with the non-AJAX version of the application and then gradually introduce AJAX features. That's what we'll do here. For starters, let's move the cart from its own page and put it in the sidebar.

## 9.1 Iteration D1: Moving the Cart

Currently, our cart is rendered by the `add_to_cart` action and the corresponding `.html` template. What we'd like to do is to move that rendering into the layout that displays the overall catalog. And that's easy, using *partial templates*.<sup>1</sup>

### Partial Templates

Programming languages let you define *methods*. A method is a chunk of code with a name: invoke the method by name, and the corresponding chunk of code gets run. And, of course, you can pass parameters to a method, which lets you write one piece of code that can be used in many different circumstances.

You can think of Rails partial templates (*partials* for short) as a kind of method for views. A partial is simply a chunk of a view in its own separate file. You can invoke (render) a partial from another template or from a controller, and the partial will render itself and return the results of that rendering. And, just as with methods, you can pass parameters to a partial, so the same partial can render different results.

We'll use partials twice in this iteration. First, let's look at the cart display itself.

[Download](#) `depot_i/app/views/store/add_to_cart.html`

```
<div class="cart-title">Your Cart</div>
<table>
  <% for cart_item in @cart.items %>
    <tr>
      <td><%= cart_item.quantity %>&times;</td>
      <td><%= h(cart_item.title) %></td>
      <td class="item-price"><%= number_to_currency(cart_item.price) %></td>
    </tr>
  <% end %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= number_to_currency(@cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Empty cart", :action => :empty_cart %>
```

1. Another way would be to use *components*. A component is a way of packaging some work done by a controller and the corresponding rendering. In our case, we could have a component called `display_cart`, where the controller action fetches the cart information from the session and the view renders the HTML for the cart. The layout would then insert this rendered HTML into the sidebar. However, there are indications that components are falling out of favor in the Rails community, so we won't use one here. (For a discussion of why components are déclassé, see Section 22.9, *The Case against Components*, on page 514.)

It creates a list of table rows, one for each item in the cart. Whenever you find yourself iterating like this, you might want to stop and ask yourself, is this too much logic in a template? It turns out we can abstract away the loop using partials (and, as we'll see, this also sets the stage for some AJAX magic later). To do this, we'll make use of the fact that you can pass a collection to the method that renders partial templates, and that method will automatically invoke the partial once for each item in the collection. Let's rewrite our cart view to use this feature.

[Download](#) depot\_j/app/views/store/add\_to\_cart.html

```
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => @cart.items) %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= number_to_currency(@cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Empty cart", :action => :empty_cart %>
```

That's a lot simpler. The render method takes the name of the partial and the collection object as parameters. The partial template itself is simply another template file (by default in the same directory as the template that invokes it). However, to keep the names of partials distinct from regular templates, Rails automatically prepends an underscore to the partial name when looking for the file. That means our partial will be stored in the file `_cart_item.html` in the `app/views/store` directory.

[Download](#) depot\_j/app/views/store/\_cart\_item.html

```
<tr>
  <td><%= cart_item.quantity %>&times;</td>
  <td><%= h(cart_item.title) %></td>
  <td class="item-price"><%= number_to_currency(cart_item.price) %></td>
</tr>
```

There's something subtle going on here. Inside the partial template, we refer to the current cart item using the variable `cart_item`. That's because the render method in the main template arranges to set a variable with the same name as the partial template to the current item each time around the loop. The partial is called `cart_item`, so inside the partial we expect to have a variable called `cart_item`.

So now we've tidied up the cart display, but that hasn't moved it into the sidebar. To do that, let's revisit our layout. If we had a partial template that could display the cart, we could simply embed a call to

```
render(:partial => "cart")
```

within the sidebar. But how would the partial know where to find the cart object? One way would be for it to make an assumption. In the layout, we have access to the `@cart` instance variable that was set by the controller. It turns out that this is also available inside partials called from the layout. However, this is a bit like calling a method and passing it some value in a global variable. It works, but it's ugly coding, and it increases coupling (which in turn makes your programs brittle and hard to maintain).

Remember using `render` with the `collection` option inside the `add_to_cart` template? It set the variable `cart_item` inside the partial. It turns out we can do the same when we invoke a partial directly. The `:object` parameter to `render` takes an object that is assigned to a local variable with the same name as the partial. So, in the layout we could call

```
<%= render(:partial => "cart", :object => @cart) %>
```

and in the `_cart.html` template, we can refer to the cart via the variable `cart`.

Let's do that wiring now. First, we'll create the `_cart.html` template. This is basically our `add_to_cart` template but using `cart` instead of `@cart`. (Note that it's OK for a partial to invoke other partials.)

[Download](#) depot\_j/app/views/store/\_cart.html

```
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => cart.items) %>

  <tr class="total-line">
    <td colspan="2">Total</td>
    <td class="total-cell"><%= number_to_currency(cart.total_price) %></td>
  </tr>
</table>

<%= button_to "Empty cart", :action => :empty_cart %>
```

Now we'll change the store layout to include this new partial in the sidebar.

[Download](#) depot\_j/app/views/layouts/store.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
</head>
<body id="store">
  <div id="banner">
    <%= image_tag("logo.png") %>
    <%= @page_title || "Pragmatic Bookshelf" %>
  </div>
  <div id="columns">
```

```

<div id="side">
▶ <div id="cart">
▶   <%= render(:partial => "cart", :object => @cart) %>
▶ </div>

  <a href="http://www....">Home</a><br />
  <a href="http://www..../faq">Questions</a><br />
  <a href="http://www..../news">News</a><br />
  <a href="http://www..../contact">Contact</a><br />
</div>
<div id="main">
<% if flash[:notice] -%>
  <div id="notice"><%= flash[:notice] %></div>
<% end -%>
  <%= yield :layout %>
</div>
</body>
</html>

```

Now we have to make a small change to the store controller. We're invoking the layout while looking at the store's index action, and that action doesn't currently set @cart. That's easy enough to remedy.

[Download](#) depot\_j/app/controllers/store\_controller.rb

```

def index
  @products = Product.find_products_for_sale
  @cart = find_cart
end

```

If you display the catalog after adding something to your cart, you should see something like Figure 9.1, on the next page.<sup>2</sup> Let's just wait for the Webby Award nomination.

## Changing the Flow

Now that we're displaying the cart in the sidebar, we can change the way that the `Add to Cart` button works. Rather than displaying a separate cart page, all it has to do is refresh the main index page. The change is pretty simple: at the end of the `add_to_cart` action, we simply redirect the browser back to the index.

[Download](#) depot\_k/app/controllers/store\_controller.rb

```

def add_to_cart
  begin
    product = Product.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  end
end

```

2. And if you've updated your CSS appropriately.... See the listing on page 680 for our CSS.

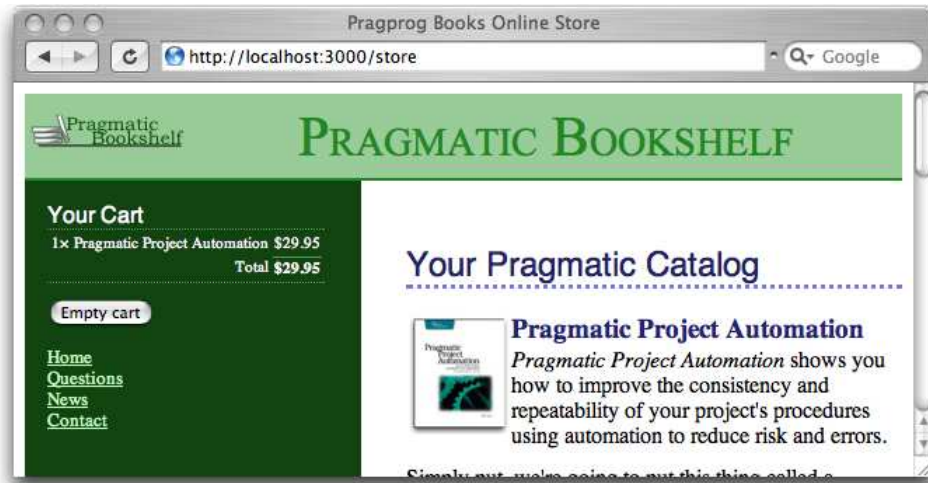


Figure 9.1: The Cart Is in the Sidebar

```

else
  @cart = find_cart
  @cart.add_product(product)
  redirect_to_index
end
end
end

```

For this to work, we need to change the definition of `redirect_to_index` to make the message parameter optional.

[Download](#) depot\_k/app/controllers/store\_controller.rb

```

def redirect_to_index(msg = nil)
  flash[:notice] = msg if msg
  redirect_to :action => :index
end

```

We should now get rid of the `add_to_cart.rhtml` template—it's no longer needed. (What's more, leaving it lying around might confuse us later in this chapter.)

So, now we have a store with a cart in the sidebar. When you click to add an item to the cart, the page is redisplayed with an updated cart. However, if our catalog is large, that redisplay might take a while. It uses bandwidth, and it uses server resources. Fortunately, we can use AJAX to make this better.

## 9.2 Iteration D2: An AJAX-Based Cart

AJAX lets us write code that runs in the browser that interacts with our server-based application. In our case, we'd like to make the `Add to Cart` buttons invoke the server `add_to_cart` action in the background. The server can then send down just the HTML for the cart, and we can replace the cart in the sidebar with the server's updates.

Now, normally you'd do this by writing JavaScript that runs in the browser and by writing server-side code that communicated with this JavaScript (possibly using a technology such as JSON). The good news is that, with Rails, all this is hidden from you. We can do everything we need to do using Ruby (and with a whole lot of support from some Rails helper methods).

The trick when adding AJAX to an application is to take small steps. So, let's start with the most basic one. Let's change the catalog page to send an AJAX request to our server application, and have the application respond with the HTML fragment containing the updated cart.

On the index page, we're using `button_to` to create the link to the `add_to_cart` action. Underneath the covers, `button_to` generates an HTML form. The helper

```
<%= button_to "Add to Cart", :action => :add_to_cart, :id => product %>
```

generates HTML that looks something like

```
<form method="post" action="/store/add_to_cart/1" class="button-to">
  <input type="submit" value="Add to Cart" />
</form>
```

This is a standard HTML form, so a POST request will be generated when the user clicks the submit button. We want to change this to send an AJAX request instead. To do this, we'll have to code the form explicitly, using a Rails helper called `form_remote_tag`. The `form..._tag` parts of the name tell you it's generating an HTML form, and the `remote` part tells you it will use AJAX to create a remote procedure call to your application. So, edit `index.rhtml` in the `app/views/store` directory, replacing the `button_to` call with something like this.

[Download](#) `depot_1/app/views/store/index.rhtml`

```
<% form_remote_tag :url => { :action => :add_to_cart, :id => product } do %>
  <%= submit_tag "Add to Cart" %>
<% end %>
```

You tell `form_remote_tag` how to invoke your server application using the `:url` parameter. This takes a hash of values that are the same as the trailing parameters we passed to `button_to`. The code inside the Ruby block (between the `do` and `end` keywords) is the body of the form. In this case, we have a simple submit button. From the user's perspective, this page looks identical to the previous one.

While we're dealing with the views, we also need to arrange for our application to send the JavaScript libraries used by Rails to the user's browser. We'll talk more about this in Chapter 23, *The Web, V2.0*, on page 523, but for now let's just add a call to `javascript_include_tag` to the `<head>` section of the store layout.

Download [depot\\_1/app/views/layouts/store.rhtml](#)

```
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag :defaults %>
</head>
```

So far, we've arranged for the browser to send an AJAX request to our application. The next step is to have the application return a response. The plan is to create the updated HTML fragment that represents the cart and to have the browser stick that HTML into the DOM<sup>3</sup> as a replacement for the cart that's already there. The first change is to stop the `add_to_cart` action redirecting to the index display. (I know, we just added that only a few pages back. Now we're taking it out again. We're agile, right?)

Download [depot\\_1/app/controllers/store\\_controller.rb](#)

```
def add_to_cart
  begin
    product = Product.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @cart = find_cart
    @cart.add_product(product)
  end
end
```

Because of this change, when `add_to_cart` finishes handling the AJAX request, Rails will look for an `add_to_cart` template to render. We deleted the old `.rhtml` template back on page 129, so it looks like we'll need to add something back in. Let's do something a little bit different.

Rails supports RJS templates—the *JS* stands for JavaScript. An `.rjs` template is a way of getting JavaScript on the browser to do what you want, all by writing server-side Ruby code. Let's write our first: `add_to_cart.rjs`. It goes in the `app/views/store` directory, just like any other template.

Download [depot\\_1/app/views/store/add\\_to\\_cart.rjs](#)

```
page.replace_html("cart", :partial => "cart", :object => @cart)
```

3. The Document Object Model. This is the browser's internal representation of the structure and content of the document being displayed. By manipulating the DOM, we cause the display to change in front of the user's eyes.

Let's analyze that template. The `page` variable is an instance of something called a JavaScript generator—a Rails class that knows how to create JavaScript on the server and have it executed by the browser. Here, we tell it to replace the content of the element on the current page with the `id cart` with...something. The remaining parameters to `replace_html` look familiar. They should—they're the same ones we used to render the partial in the store layout. This simple `.rjs` template renders the HTML that represents the cart. It then tells the browser to replace the content of `<div>` whose `id="cart"` with that HTML.

Does it work? It's hard to show in a book, but it sure does. Make sure you reload the index page in order to get the `form_remote_tag` and the JavaScript libraries loaded into your browser. Then, click one of the `Add to Cart` buttons. You should see the cart in the sidebar update. And you *shouldn't* see your browser show any indication of reloading the page. You've just created an AJAX application.

### Troubleshooting

Although Rails makes AJAX incredibly simple, it can't make it foolproof. And, because you're dealing with the loose integration of a number of technologies, it can be hard to work out why your AJAX doesn't work. That's one of the reasons you should always add AJAX functionality one step at a time.

Here are a few hints if your Depot application didn't show any AJAX magic.

- Did you delete the old `add_to_cart.rhtml` file?
- Did you remember to include the JavaScript libraries in the store layout (using `javascript_include_tag`)?
- Does your browser have any special incantation to force it to reload everything on a page? Sometimes browsers hold local cached versions of page assets, and this can mess up testing. Now would be a good time to do a full reload.
- Did you have any errors reported? Look in `development.log` in the logs directory.
- Still looking at the log file, do you see incoming requests to the action `add_to_cart`? If not, it means your browser isn't making AJAX requests. If the JavaScript libraries have been loaded (using View → Source in your browser will show you the HTML), perhaps your browser has JavaScript execution disabled?
- Some readers have reported that they have to stop and start their application to get the AJAX-based cart to work.

- If you're using Internet Explorer, it might be running in what Microsoft call *quirks mode*, which is backward compatible with old IE releases but is also broken. IE switches into *standards mode*, which works better with the AJAX stuff, if the first line of the downloaded page is an appropriate DOCTYPE header. Our layouts use

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

### The Customer Is Never Satisfied

We're feeling pretty pleased with ourselves. We changed a handful of lines of code, and our boring old Web 1.0 application now sports Web 2.0 AJAX speed stripes. We breathlessly call the client over. Without saying anything, we proudly press  and look at her, eager for the praise we know will come. Instead, she looks surprised. "You called me over to show me a bug?" she asks. "You click that button, and nothing happens."

We patiently explain that, in fact, quite a lot happened. Just look at the cart in the sidebar. See? When we add something, the quantity changes from 4 to 5.

"Oh," she says, "I didn't notice that." And, if she didn't notice the page update, it's likely our customers won't either. Time for some user-interface hacking.

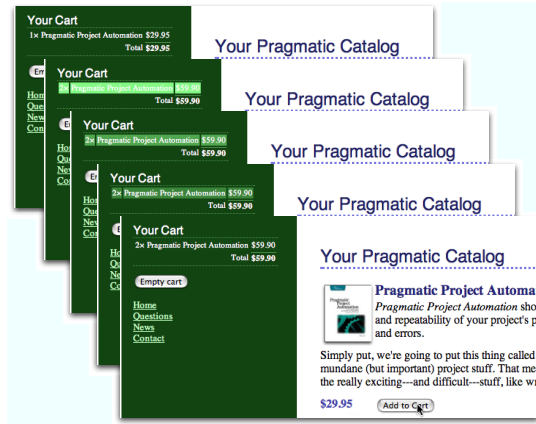
## 9.3 Iteration D3: Highlighting Changes

We said earlier that the `javascript_include_tag` helper downloads a number of JavaScript libraries to the browser. One of those libraries, `effects.js`, lets you decorate your web pages with a number of visually interesting effects.<sup>4</sup> One of these effects is the (now) infamous Yellow Fade Technique. This highlights an element in a browser: by default it flashes the background yellow and then gradually fades it back to white. Figure 9.2, on the next page, shows the Yellow Fade Technique being applied to our cart: the image at the back shows the original cart. The user clicks the  button, and the count updates to 2 as the line flares brighter. It then fades back to the background color over a short period of time.

Let's add this kind of highlight to our cart. Whenever an item in the cart is updated (either when it is added or when we change the quantity), let's flash its background. That will make it clearer to our users that something has changed, even though the whole page hasn't been refreshed.

The first problem we have is identifying the most recently updated item in the cart. Right now, each item is simply a `<tr>` element. We need to find a way to flag the most recently changed one. The work starts in the `Cart` model. Let's

4. `effects.js` is part of the `Script.aculo.us` library. Have a look at the visual effects page at <http://wiki.script.aculo.us/scriptaculous/show/VisualEffects> to see the cool things you can do with it.




---

Figure 9.2: Our Cart with the Yellow Fade Technique

---

have the `add_product` method return the `CartItem` object that was either added to the cart or had its quantity updated.

[Download](#) `depot_m/app/models/cart.rb`

```
def add_product(product)
  current_item = @items.find {|item| item.product == product}
  if current_item
    current_item.increment_quantity
  else
    current_item = CartItem.new(product)
    @items << current_item
  end
  current_item
end
```

Over in `store_controller.rb`, we'll take that information and pass it down to the template by assigning it to an instance variable.

[Download](#) `depot_m/app/controllers/store_controller.rb`

```
def add_to_cart
  begin
    product = Product.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @cart = find_cart
    @current_item = @cart.add_product(product)
  end
end
```

In the `_cart_item.html` partial, we then check to see whether the item we're rendering is the one that just changed. If so, we tag it with an id of `current_item`.

[Download](#) `depot_m/app/views/store/_cart_item.html`

```
<% if cart_item == @current_item %>
  <tr id="current_item">
<% else %>
  <tr>
<% end %>
  <td><%= cart_item.quantity %>&times;</td>
  <td><%= h(cart_item.title) %></td>
  <td class="item-price"><%= number_to_currency(cart_item.price) %></td>
</tr>
```

As a result of these three minor changes, the `<tr>` element of the most recently changed item in the cart will be tagged with `id="current_item"`. Now we just need to tell the JavaScript to invoke the highlight effect on that item. We do this in the existing `add_to_cart.js` template, adding a call to the `visual_effect` method.

[Download](#) `depot_m/app/views/store/add_to_cart.js`

```
page.replace_html("cart", :partial => "cart", :object => @cart)
```

```
page[:current_item].visual_effect :highlight,
                                  :startcolor => "#88ff88",
                                  :endcolor => "#114411"
```

See how we identified the browser element that we wanted to apply the effect to by passing `:current_item` to the page? We then asked for the *highlight* visual effect and overrode the default yellow/white transition with colors that work better with our design. Click to add an item to the cart, and you'll see the changed item in the cart glow a light green before fading back to merge with the background.

## 9.4 Iteration D4: Hide an Empty Cart

One last request from the customer: right now, even carts with nothing in them are still displayed in the sidebar. Can we arrange for the cart to appear only when it has some content? But of course!

In fact, we have a number of options. The simplest is probably to include the HTML for the cart only if the cart has something in it. We can do this totally within the `_cart` partial.

```
<% unless cart.items.empty? %>
<div class="cart-title">Your Cart</div>
<table>
  <%= render(:partial => "cart_item", :collection => cart.items) %>

  <tr class="total-line">
```

```

      <td colspan="2">Total</td>
      <td class="total-cell"><%= number_to_currency(cart.total_price) %></td>
    </tr>
  </table>

  <%= button_to "Empty cart", :action => :empty_cart %>
  <% end %>

```

Although this works, the user interface is somewhat brutal: the whole sidebar redraws on the transition between a cart that's empty and a cart with something in it. So let's not use this code. Instead, let's smooth it out a little.

The Script.aculo.us effects library contains a number of nice transitions that make elements appear. Let's use *blind\_down*, which will smoothly reveal the cart, sliding the rest of the sidebar down to make room.

Not surprisingly, we'll use our existing *.rjs* template to call the effect. Because the *add\_to\_cart* template is invoked only when we add something to the cart, then we know that we have to reveal the cart in the sidebar whenever there is exactly one item in the cart (because that means that previously the cart was empty and hence hidden). And, because the cart should be visible before we start the highlight effect, we'll add the code to reveal the cart before the code that triggers the highlight.

The template now looks like this.

```

Download depot_n/app/views/store/add_to_cart.rjs
page.replace_html("cart", :partial => "cart", :object => @cart)
▶ page[:cart].visual_effect :blind_down if @cart.total_items == 1

page[:current_item].visual_effect :highlight,
                                  :startcolor => "#88ff88",
                                  :endcolor => "#114411"

```

This won't yet work, because we don't have a *total\_items* method in our cart model.

```

Download depot_n/app/models/cart.rb
def total_items
  @items.sum { |item| item.quantity }
end

```

We have to arrange to hide the cart when it's empty. There are two basic ways of doing this. One, illustrated by the code at the start of this section, is not to generate any HTML at all. Unfortunately, if we do that, then when we add something to the cart and suddenly create the cart HTML, we see a flicker in the browser as the cart is first displayed and then hidden and slowly revealed by the *blind\_down* effect.

A better way to handle the problem is to create the cart HTML but set the CSS style to `display: none` if the cart is empty. To do that, we need to change the `store.html` layout in `app/views/layouts`. Our first attempt is something like this.

```
<div id="cart"
  <% if @cart.items.empty? %>
    style="display: none"
  <% end %>
>
  <%= render(:partial => "cart", :object => @cart) %>
</div>
```

This code adds the CSS `style=` attribute to the `<div>` tag, but only if the cart is empty. It works fine, but it's really, really ugly. That dangling `>` character looks misplaced (even though it isn't), and the way logic is interjected into the middle of a tag is the kind of thing that gives templating languages a bad name. Let's not let that kind of ugliness litter our code. Instead, let's create an abstraction that hides it—we'll write a helper method.

## Helper Methods

Whenever we want to abstract some processing out of a view (any kind of view), we want to write a helper method.

If you look in the `app` directory, you'll find four subdirectories.

```
depot> ls -p app
controllers/  helpers/      models/      views/
```

Not surprisingly, our helper methods go in the `helpers` directory. If you look in there, you'll find it already contains some files.

```
depot> ls -p app/helpers
admin_helper.rb      application_helper.rb  store_helper.rb
```

The Rails generators automatically created a helper file for each of our controllers (admin and store). The Rails command itself (the one that created the application initially) created the file `application_helper.rb`. The methods we define in a controller-specific helper are available to views referenced by that controller. Methods in the overall `application_helper` file are available in all the application's views. This gives us a choice for our new helper. Right now, we need it just in the store view, so let's start by putting it there.

Let's have a look at the file `store_helper.rb` in the `helpers` directory.

```
module StoreHelper
end
```

Let's write a helper method called `hidden_div_if`. It takes a condition, an optional set of attributes, and a block. It wraps the output generated by the block in a `<div>` tag, adding the `display: none` style if the condition is true. We'd use it in the store layout like this.

Download depot\_n/app/views/layouts/store.rhtml

```
<% hidden_div_if(@cart.items.empty?, :id => "cart") do %>
  <%= render(:partial => "cart", :object => @cart) %>
<% end %>
```

We'll write our helper so that it is local to the store controller by adding it to `store_helper.rb` in the `app/helpers` directory.

Download depot\_n/app/helpers/store\_helper.rb

```
module StoreHelper

  def hidden_div_if(condition, attributes = {}, &block)
    if condition
      attributes["style"] = "display: none"
    end
    content_tag("div", attributes, &block)
  end
end
```

This code uses the Rails standard helper, `content_tag`, which can be used to wrap the output created by a block in a tag. By using the `&block` notation, we get Ruby to pass the block that was given to `hidden_div_if` down to `content_tag`.

And, finally, we need to stop setting the message in the flash that we used to display when the user empties a cart. It really isn't needed any more, because the cart clearly disappears from the sidebar when the catalog index page is redrawn. But there's another reason to remove it, too. Now that we're using AJAX to add products to the cart, the main page doesn't get redrawn between requests as people shop. That means we'll continue to display the flash message saying the cart is empty even as we display a cart in the sidebar.

Download depot\_n/app/controllers/store\_controller.rb

```
def empty_cart
  session[:cart] = nil
  redirect_to_index
end
```

Although this might seem like a lot of steps, it really isn't. All we did to make the cart hide and reveal itself was to make the CSS display style conditional on the number of items in the cart and to use the `.rjs` template to invoke the `blind_down` effect when the cart went from being empty to having one item.

Everyone is excited to see our fancy new interface. In fact, because our computer is on the office network, our colleagues point their browsers at our test application and try it for themselves. Lots of low whistles follow as folks marvel at the way the cart appears and then updates. Everyone loves it. Everyone, that is, except Bruce. Bruce doesn't trust JavaScript running in his browser and has it turned off. And, with JavaScript disabled, all our fancy AJAX stops working. When Bruce adds something to his cart, he sees something strange.

```

$("cart").update("<h1>Your Cart</h1>\n\n<ul>\n \n <li
id=\"current_item\">\n\n 3 &times; Pragmatic Project
Automation\n</li>\n</ul>\n \n<form method=\"post\"
action=\"/store/empty_cart\" class=\"button-to...

```

Clearly this won't do. We need to have our application work if our users have disabled JavaScript in their browsers. That'll be our next iteration.

## 9.5 Iteration D5: Degrading If Javascript Is Disabled

Remember, back on page 128, we arranged for the cart to appear in the sidebar. We did this before we added a line of AJAX code to the application. If we could fall back to this behavior when JavaScript is disabled in the browser, then the application would work for Bruce as well as for our other co-workers. This basically means that if the incoming request to `add_to_cart` doesn't come from JavaScript, we want to do what the original application did and redirect to the index page. When the index displays, the updated cart will appear in the sidebar.

If a user clicks the button inside a `form_remote_tag`, one of two things happens. If JavaScript is disabled, the target action in the application is invoked using a regular HTTP POST request—it acts just like a regular form. If, however, JavaScript is enabled, it overrides this conventional POST and instead uses a JavaScript object to establish a back channel with the server. This object is an instance of class `XmlHttpRequest`. Because that's a mouthful, most folks (and Rails) abbreviate it to `xhr`.

So, on the server, we can tell that we're talking to a JavaScript-enabled browser by testing to see whether the incoming request was generated by an `xhr` object. And the Rails request object, available inside controllers and views, makes it easy to test for this condition: it provides an `xhr?` method. As a result, making our application work regardless of whether JavaScript is enabled takes just a single line of code in the `add_to_cart` action.

[Download](#) `depot_o/app/controllers/store_controller.rb`

```

def add_to_cart
  begin
    product = Product.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    logger.error("Attempt to access invalid product #{params[:id]}")
    redirect_to_index("Invalid product")
  else
    @cart = find_cart
    @current_item = @cart.add_product(product)
    redirect_to_index unless request.xhr?
  end
end

```

## 9.6 What We Just Did

In this iteration we added AJAX support to our cart.

- We moved the shopping cart into the sidebar. We then arranged for the `add_to_cart` action to redisplay the catalog page.
- We used `form_remote_tag` to invoke the `add_to_cart` action using AJAX.
- We then used an `.rjs` template to update the page with just the cart's HTML.
- To help the user see changes to the cart, we added a highlight effect, again using the `.rjs` template.
- We wrote a helper method that hides the cart when it is empty and used the `.rjs` template to reveal it when an item is added.
- Finally, we made our application work if the user's browser has JavaScript disabled by reverting to the behavior we implemented before starting on the AJAX journey.

The key point to take away is the incremental style of AJAX development. Start with a conventional application, and then add AJAX features, one by one. AJAX can be hard to debug: by adding it slowly to an application, you make it easier to track down what changed if your application stops working. And, as we saw, starting with a conventional application makes it easier to support both AJAX and non-AJAX behavior in the same codebase.

Finally, a couple of hints. First, if you plan to do a lot of AJAX development, you'll probably need to get familiar with your browser's JavaScript debugging facilities and with its DOM inspectors. Chapter 8 of *Pragmatic Ajax: A Web 2.0 Primer* [JG06] has a lot of useful tips. And, second, I find it useful to run two different browsers when I'm developing (I personally use Firefox and Safari on my Mac). I have JavaScript enabled in one, disabled in the other. Then, as I add some new feature, I poke at it with both browsers to make sure it works regardless of the state of JavaScript.

### Playtime

Here's some stuff to try on your own.

- The cart is currently hidden when the user empties it by redrawing the entire catalog. Can you change the application to use the `Script.aculo.us` `blind_up` instead?
- Does the change you made work if the browser has JavaScript disabled?
- Experiment with other visual effects for new cart items. For example, can you set their initial state to hidden and then have them grow into place?

Does this make it problematic to share the cart item partial between the AJAX code and the initial page display?

- Add a link next to each item in the cart. When clicked it should invoke an action to decrement the quantity of the item, deleting it from the cart when the quantity reaches zero. Get it working without using AJAX first, and then add the AJAX goodness.

(You'll find hints at <http://wiki.pragprog.com/cgi-bin/wiki.cgi/RailsPlayTime>)