O'REILLY
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
http://www.onlamp.com/pub/a/onlamp/2005/06/09/rails_ajax.html
See this if you're having trouble printing code examples

Ajax on Rails

by Curt Hibbs
06/09/2005

In a few short months, Ajax has moved from an obscure and rarely used technology to the hottest thing since sliced bread. This article introduces the incredibly easy-to-use Ajax support that is part of the Ruby on Rails web application framework. This is not a step-by-step tutorial, and I assume that you know a little bit about how to organize and construct a Rails web application. If you need a quick refresher, check out Rolling with Ruby on Rails, Part 1 and Part 2.

Just in case you've been stranded on a faraway island for most of the year, here's the history of Ajax in 60 seconds or less.

In the beginning, there was the World Wide Web. Compared with desktop applications, web applications were slow and clunky. People liked web applications anyway because they were conveniently available from anywhere, on any computer that had a browser. Then Microsoft created XMLHttpRequest in Internet Explorer 5, which let browser-side JavaScript communicate with the web server in the background without requiring the browser to display a new web page. That made it possible to develop more fluid and responsive web applications. Mozilla soon implemented XMLHttpRequest in its browsers, as did Apple (in the Safari browser) and Opera.

XMLHttpRequest must have been one of the Web's best kept secrets. Since its debut in 1998, few sites have used it at all, and most developers, if they even knew about it, never used it. Google started to change that when it released a series of high-profile web applications with sleek new UIs powered by XMLHttpRequest. The most visually impressive of these is Google Maps, which gives you the illusion of being able to drag around an infinitely sizable map in its little map window.

While Google's prominent use of XMLHttpRequest dramatically demonstrated that vastly improved UIs for web apps were possible, it was Jesse James Garrett's February 18 essay that finally gave this technique a usable name: Ajax (Asynchronous JavaScript and XML). That was the tipping point. Without knowing it, we as an industry had been waiting for this, and the new Ajax name spread like wildfire. I have never seen such rapid and near universal adoption of a new technology moniker!

Traditional Web App vs. an Ajax App

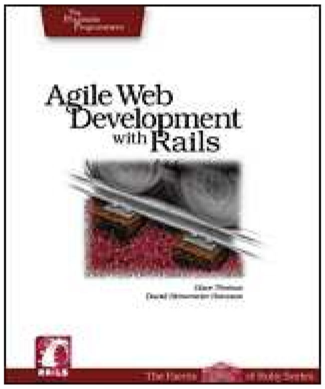
Let me distill the essence of an Ajax web application by examining a use case: inserting a new item into a list.

A typical user interface displays the current list on a web page followed by an input field in which the user can type the text of a new item. When the user clicks on a Create New Item button, the app actually creates and inserts the new item into the list.

At this point, a traditional web application sends the value of the input field to the server. The server then acts upon the data (usually by updating a database) and responds by sending back a new web page that displays an updated list that now contains the new item. This uses a lot of bandwidth, because most of the new page is exactly the same as the old one. The performance of this web app degrades as the list gets longer.

In contrast, an Ajax web application sends the input field to the server in the background and updates the affected portion of the current web page in place. This dramatically increases the responsiveness of the user interface and makes it feel much more like a desktop application.

Related Reading



Agile Web Development with Rails
A Pragmatic Guide
By Dave Thomas, David Heinemeier

Hansson

You can see this for yourself. Below are links to two different weblogs, one that uses Ajax to post comments and another that does not. Try posting some comments to each one:

Traditional Web App

Ajax Web App

Ajax is all about usability--but like any technology or technique, you can use it well or use it poorly. After showing how to use Ajax, I'll give some guidelines on when to use Ajax and when not to.

How to Use Ajax in Your Web Application

The hard way to use Ajax in your web app is to write your own custom JavaScript that directly uses the XMLHttpRequest object's API. By doing this, you have to deal with the idiosyncrasies of each browser.

An easier way is to use one of several JavaScript libraries that provide higher-level Ajax services and hide the differences between browsers. Libraries such as DWR, Prototype, Sajax, and Ajax.NET are all good choices.

The easiest way of all is to use the built-in Ajax facilities of Ruby on Rails. In fact, Rails makes Ajax so easy that for typical cases it's no harder to use Ajax than it is not to!

How Rails Implements Ajax

Rails has a simple, consistent model for how it implements Ajax operations.

Once the browser has rendered and displayed the initial web page, different user actions cause it to display a new web page (like any traditional web app) or trigger an Ajax operation:

1. A trigger action occurs. This could be the user clicking on a button or link, the user making changes to the data on a form or in a field, or just a periodic trigger (based on a timer).
2. Data associated with the trigger (a field or an entire form) is sent asynchronously to an action handler on the server via XMLHttpRequest.
3. The server-side action handler takes some action (that's why it is an *action handler*) based on the data, and returns an HTML fragment as its response.
4. The client-side JavaScript (created automatically by Rails) receives the HTML fragment and uses it to update a specified part of the current page's HTML, often the content of a `<div>` tag.

An Ajax request to the server can also return any arbitrary data, but I'll talk only about HTML fragments. The real beauty is how easy Rails makes it to implement all of this in your web application.

Using `link_to_remote`

Rails has several helper methods for implementing Ajax in your view's templates. One of the simplest yet very versatile methods is `link_to_remote()`. Consider a simple web page that asks for the time and has a link on which the user clicks to obtain the current time. The app uses Ajax via `link_to_remote()` to retrieve the time and display it on the web page.

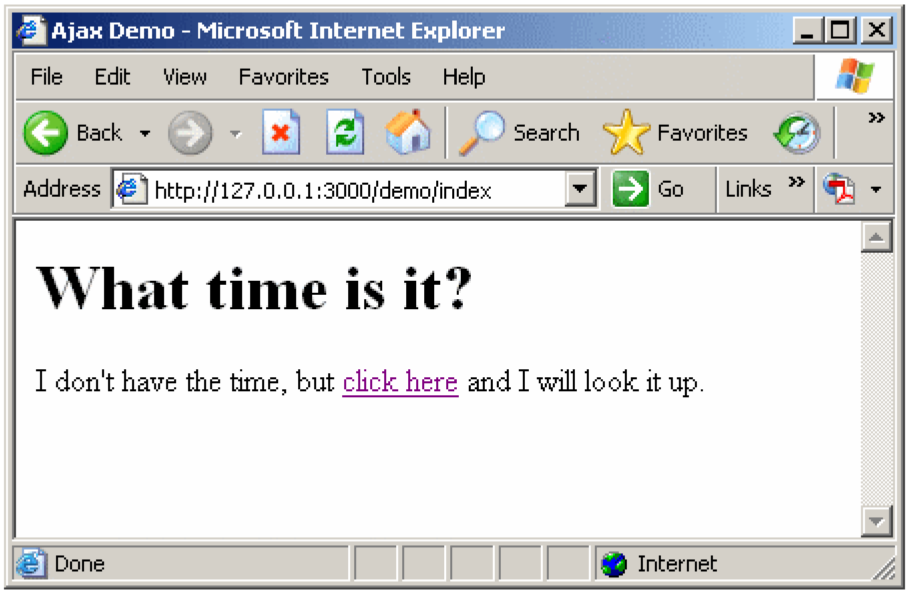
My view template (*index.rhtml*) looks like:

```
<html>
  <head>
    <title>Ajax Demo</title>
    <%= javascript_include_tag "prototype" %>
  </head>
  <body>
    <h1>What time is it?</h1>
    <div id="time_div">
      I don't have the time, but
      <%= link_to_remote( "click here",
                          :update => "time_div",
                          :url =>{ :action => :say_when }) %>
      and I will look it up.
    </div>
  </body>
</html>
```

There are two helper methods of interest in the above template, marked in bold. `javascript_include_tag()` includes the Prototype JavaScript library. All of the Rails Ajax features use this JavaScript library, which the Rails distribution helpfully includes.

The `link_to_remote()` call here uses its simplest form, with three parameters:

1. The text to display for the link--in this case, "click here".
2. The id of the DOM element containing content to replace with the results of executing the action--in this case, `time_div`.
3. The URL of the server-side action to call--in this case, an action called `say_when`.


Ajax Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:3000/demo/index Go Links
Done Internet

What time is it?

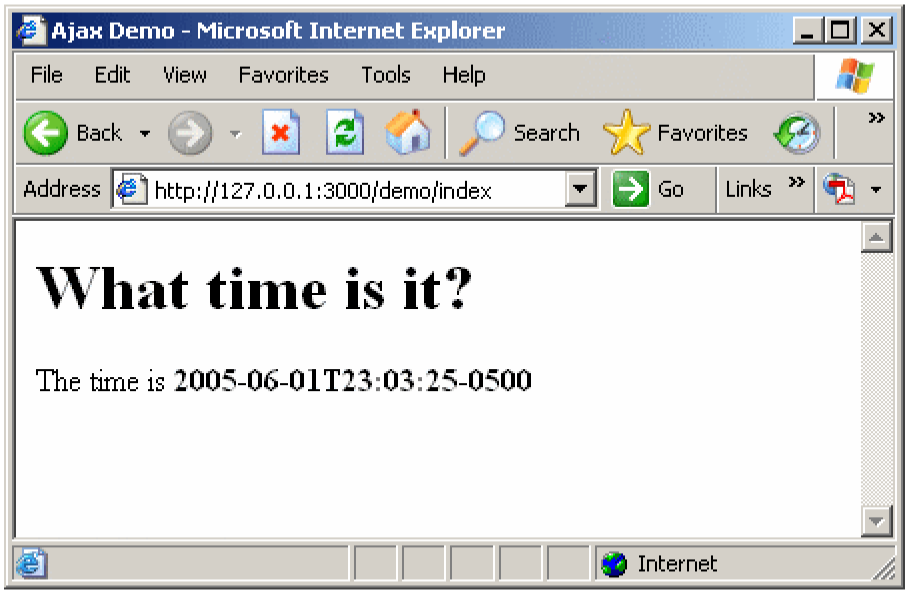
I don't have the time, but click here and I will look it up.

Figure 1. Before clicking on the link

My controller class looks like:

```
class DemoController < ApplicationController
  def index
  end

  def say_when
    render_text "<p>The time is <b>" + DateTime.now.to_s + "</b></p>"
  end
end
```


Ajax Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:3000/demo/index Go Links
Internet

What time is it?

The time is **2005-06-01T23:03:25-0500**

Figure 2. After clicking on the link

The `index` action handler doesn't do anything except letting Rails recognize that there is an index action and causing the rendering of the *index.rhtml* template. The `say_when` action handler constructs an HTML fragment that contains the current date and time. Figures 1 and 2 show how the index page appears both before and after clicking on the "click here" link.

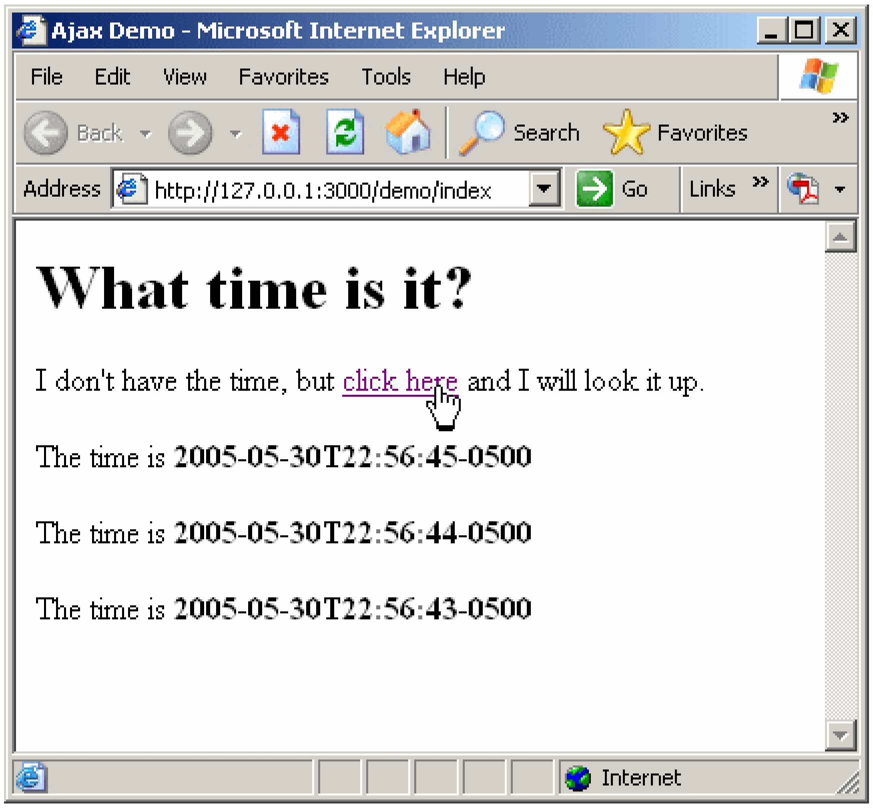
When the user clicks on the "click here" link, the browser constructs an XMLHttpRequest, sending it to the server with a URL that will invoke the `say_when` action handler, which returns an HTML response fragment containing the current time. The client-side JavaScript receives this response and uses it to replace the contents of the `<div>` with an id of `time_div`.

It's also possible to insert the response, instead of replacing the existing content:

```
<%= link_to_remote( "click here",
                   :update => "time_div",
                   :url => { :action => :say_when },
                   :position => "after" ) %>
```

I added an optional parameter **`:position => "after"`**, which tells Rails to insert the returned HTML fragment after the target element (`time_div`). The position parameter can accept the values `before`, `after`, `top`, and `bottom`. `top` and `bottom` insert inside of the target element, while `before` and `after` insert outside of the target element.

In any case, now that I added the position parameter, my "click here" link won't disappear, so I can click on it repeatedly and watch the addition of new time reports.


Ajax Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:3000/demo/index Go Links
Internet

What time is it?

I don't have the time, but click here and I will look it up.

The time is **2005-05-30T22:56:45-0500**

The time is **2005-05-30T22:56:44-0500**

The time is **2005-05-30T22:56:43-0500**

Figure 3. The position option inserts new content

The web page doesn't change, and neither does the URL being displayed by the browser. In a trivial example like this, there has not been much of a savings over an entire page refresh. The difference becomes more noticeable when you have a more complicated page of which you need to update only a small portion.

Using `form_remote_tag`

The `form_remote_tag()` helper is similar to `link_to_remote()` except that it also sends the contents of an HTML form. This means that the action handler can use user-entered data to formulate the response. This example displays a web page that shows a list and an Ajax-enabled form that lets users add items to the list.

My view template (*index.rhtml*) looks like:

```
<html>
  <head>
    <title>Ajax List Demo</title>
    <%= javascript_include_tag "prototype" %>
  </head>
  <body>
    <h3>Add to list using Ajax</h3>
    <%= form_remote_tag(:update => "my_list",
                       :url => { :action => :add_item },
                       :position => "top" ) %>
      New item text:
      <%= text_field_tag :newitem %>
      <%= submit_tag "Add item with Ajax" %>
    <%= end_form_tag %>
    <ul id="my_list">
      <li>Original item... please add more!</li>
    </ul>
  </body>
</html>
```

Notice the two parts in bold. They define the beginning and end of the form. Because the form started with `form_remote_tag()` instead of `form_tag()`, the application will submit this form using XMLHttpRequest. The parameters to `form_remote_tag()` should look familiar:

- The update parameter specifies the id of the DOM element with content to update by the results of executing the action--in this case, `my_list`.
- The url parameter specifies the server-side action to call--in this case, an action named `add_item`.
- The position parameter says to insert the returned HTML fragment at the top of the content of the `my_list` element--in this case, a `<ul>` tag.

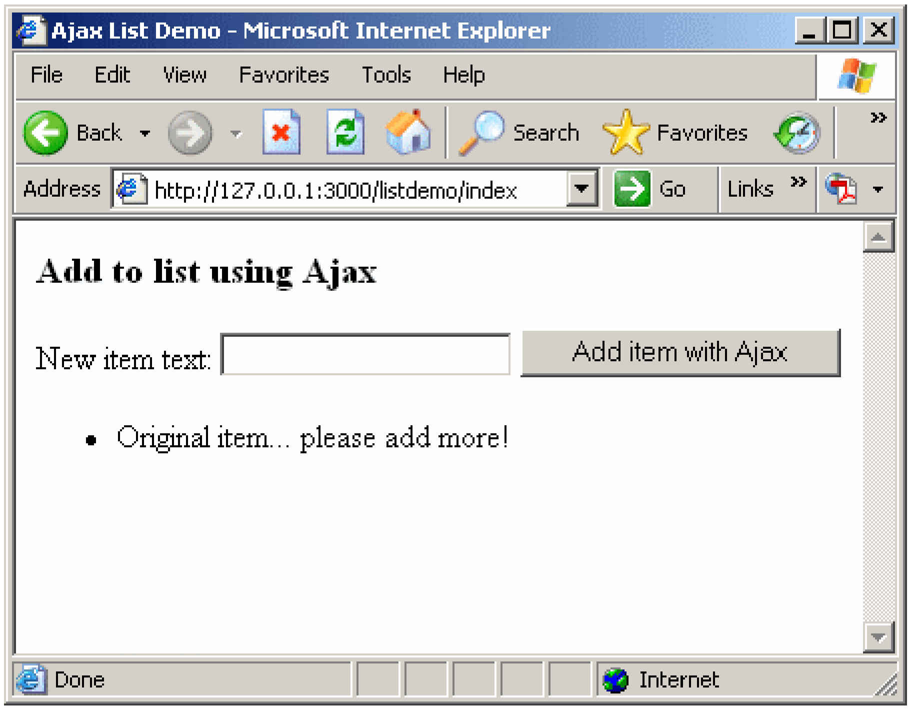

Ajax List Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:3000/listdemo/index Go Links
Add to list using Ajax
New item text:
Add item with Ajax
Original item... please add more!
Done
Internet

Figure 4. Before adding any items

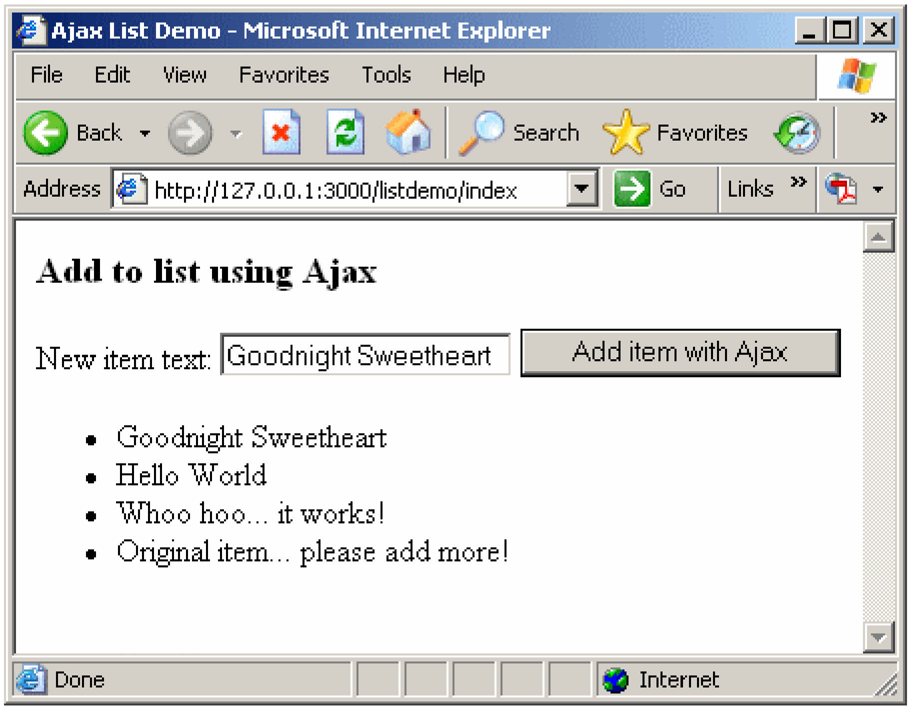
My controller class looks like:

```
class ListdemoController < ApplicationController
  def index
  end

  def add_item
    render_text "<li>" + params[:newitem] + "</li>"
  end
```

```
end
```

The `add_item` action handler constructs an HTML list item fragment containing whatever text the user entered into the `newitem` text field of the form.


Ajax List Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:3000/listdemo/index Go Links
Done Internet

Add to list using Ajax

New item text: Goodnight Sweetheart [Add item with Ajax]

- Goodnight Sweetheart
- Hello World
- Whoo hoo... it works!
- Original item... please add more!

Figure 5. After adding several new list items

Using Observers

Rails lets you monitor the value of a field and make an Ajax call to an action handler whenever the value of the field changes. The current value of the observed field is sent to the action handler in the post data of the call.

A very common use for this is to implement a live search:

```
<label for="searchtext">Live Search:</label>
<%= text_field_tag :searchtext %>
<%= observe_field(:searchtext,
                  :frequency => 0.25,
                  :update => :search_hits,
                  :url => { :action => :live_search }) %>
<p>Search Results:</p>
<div id="search_hits"></div>
```

This code snippet monitors the value of a text field named `searchtext`. Every quarter of a second, Rails checks the field for changes. If the field has changed, the browser will make an Ajax call to the `live_search` action handler, displaying the results in the `search_hits` div.

You can see an actual demonstration of this live search on my weblog. The search box is in the upper-right corner. Try typing `enterprise` or `rails` and see what you get.

To Use or Not Use (Ajax, That Is)

When you use Ajax techniques to update portions of a web page, the user gains responsiveness and fluidity. However, the user also loses the ability to bookmark and to use the browser's back button. Both of these drawbacks stem from the same fact: the URL does not change because the browser has not loaded a new page.

Don't use Ajax just because it's cool. Think about what makes sense in your web app's user interface.

For example, if a web page displays a list of accounts with operations on the displayed list like adding, deleting, and renaming accounts, these are all good candidates for Ajax. If the user clicks on a link to show all invoices that belong to an account, that's when you should display a new page and avoid Ajax.

This means that the user can bookmark the accounts page and invoices page, and use the back and forward buttons to switch between them. The user can't bookmark the operations within one of these lists or use the back button to try to undo an operation on the list (both of which you would probably want to prevent in a traditional web app, as well).

Odds 'n' Ends

I'd like to bring a couple of really cool things to your attention, but I won't be going into any detail.

Web pages that upload files are often frustrating to users because the user receives no feedback on the status of the upload while it progresses. Using Ajax, you can communicate with the server during the upload to retrieve and display the status of the upload. Sean Treadway and Thomas Fuchs have implemented a live demonstration of this using Rails and a video on how to implement it.

The Prototype JavaScript library that Rails uses also implements a large number of visual effects. The Effects Demo Page has a live demonstration of these effects along with JavaScript calls to use them

You can find more detailed information about these and other Ajax features of Rails in Chapter 18 of Agile Web Development with Rails.

Parting Thoughts

The Web has come a long way since the days of isolated web sites serving up static pages. We are slowly moving into a new era where sites are dynamically interconnected, web APIs allow us to easily build on top of existing services, and the web user interface is becoming more fluid and responsive. Ajax not only plays an important role in this emerging Web 2.0 saga, but also raises the bar on what people will consider to be an acceptable web application.

By all rights, adding complex Ajax features to a web application should be a lot of extra work, but Rails makes it dead simple.

Resources

Web sites

- Official Ruby home page
- Official Ruby on Rails home page
- Rails Ajax API
- Download the Rails source code used to create the screenshots in this article.

Mailing lists

- Rails mailing list
- Ruby-talk mailing list

Curt Hibbs is a senior software developer in Saint Louis, Missouri, with more than 30 years' experience in platforms, languages, and technologies too numerous to list.

Return to ONLamp.com.

Copyright © 2005 O'Reilly Media, Inc.