Extracted from:

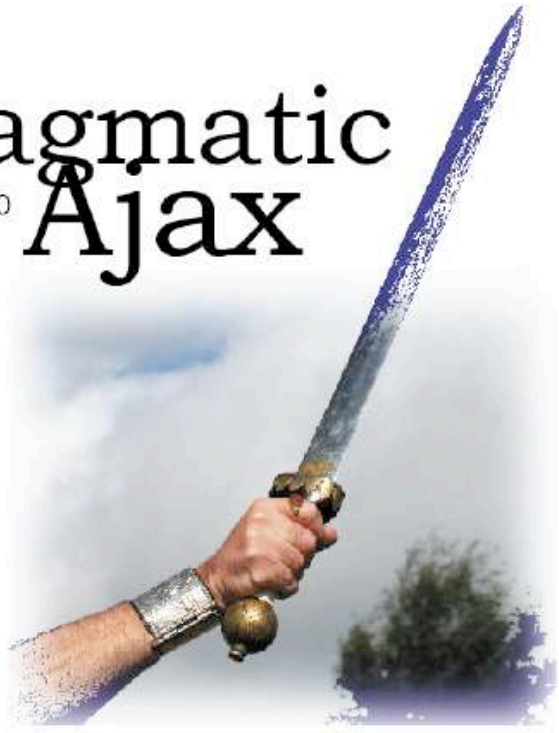
Pragmatic Ajax

A Web 2.0 Primer

The Pragmatic Programmers

Pragmatic Ajax

A Web 2.0 Primer



Justin Gehtland
Ben Galbraith
Dion Almaer

Chapter 5

Ajax Frameworks

Until now, we've looked at Ajax either at an abstract architectural level or from down in the tunnels underneath the structure. The DOM API and JavaScript's sometimes tortured interactions with it form the basis of all other Ajaxian techniques. Though it is vital to understand these things for when you run into trouble, it is also likely that you've been left scratching your head from time to time. Maybe you wondered who decided to use magic numbers for all the readyState() values. Or why the industry-standard way to create an XHR instance is in a try/catch block that will encounter an exception ~70% of the time. In fact, if you are anything like us, it probably occurred to you that you could write a fairly simple wrapper around this stuff to make it more usable in production code. These wrappers are fairly common; the Internet is littered with their corpses.

A few library wrappers have survived and flourished to become full-fledged toolkits. They provide us with much better leverage for using these Ajaxian techniques to make real applications. In this chapter, we will look at several of these frameworks at our disposal and will rewrite Hector's CRM application using the most mature and popular versions.

5.1 Frameworks, Toolkits, and Libraries

As Ajax has taken off, we've been inundated with projects claiming to have Ajax support. Since the term itself has such a broad meaning in the popular consciousness, it's often hard to know exactly what this means. Does the site perform asynchronous callbacks to the server? Does it re-render fresh data in-page? Or does it just manipulate the properties of existing DOM nodes? Figure 5.1, on the following page, clarifies the distinct layers of Ajax proper.

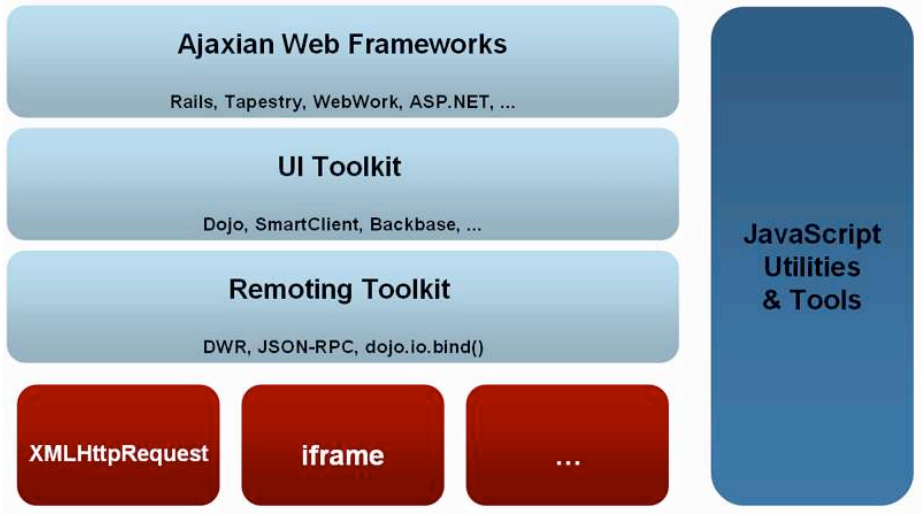


Figure 5.1: Layers of Ajax Frameworks

Remoting Toolkit

The lowest level of Ajax helpers is a remoting toolkit. If you were to create your own toolkit, this would probably be where you'd start out: wrapping XMLHttpRequest with your own API to make life easier. A really good remoting toolkit should be able to do much more than simply hide our ugly try/catch XHR instantiation code. What should happen if your Ajaxian page is loaded into a browser that does not support XMLHttpRequest? It ought to find a way, if possible, to provide all (or at least some) of the page's functionality by other means. For example, some remoting toolkits will use a hidden iframe to provide fake XHR support to the page.

Figure 5.1 lists a handful of such frameworks, and shows what each attempts to provide to developers. The Dojo Toolkit, JSON-RPC, and Prototype are all pure JavaScript frameworks that are agnostic about the world of the server side (although Prototype was built with Ruby on Rails in mind).

Others, such as DWR (Direct Web Remoting), couple a JavaScript client library with a server-side listener piece written for the Java platform. JSON-RPC itself has various bindings for many back-end languages.

iframes

Prior to the broad adoption of the XMLHttpRequest object, many web applications were using a hidden iframe to accomplish in-page round-trips back to the server. An iframe is just like a normal HTML frame (a container that can be targeted at a URL and render the results) except that it is embedded in another page. These applications simply created an iframe of 0px by 0px and then caused it to refresh against a given URL in order to pull more data back from the server.

While the technique is valid and worked for many, there were two inherent problems. The first is, if you wanted multiple asynchronous requests, you had to have multiple iframes. This became a game of guessing how many you would need and embedding that many in the page, which is not a tremendous burden, just somewhat ungainly.

More important is the question of *coding intentionally*: the use of iframe is a quintessential kludge. By that, we mean it's the repurposing of a technology to do something it wasn't quite meant to do. Though it works, it always feels a little like cheating. XMLHttpRequest, however poorly named, is an object specifically designed for initiating, monitoring, and harvesting the results of in-page postbacks. Programming against it feels natural, and lends itself to more readable (and therefore maintainable) code.

A third issue, which affects IE, is that the iframe issues audio feedback to the user whenever it makes a request. This comes in the form of a "click" sound, which can be jarring for the user since they usually have no other indication of ongoing asynchronous behavior.

Toolkit Resources

- Dojo: http://dojotoolkit.com
- Prototype: http://prototype.conio.net/
- Script.aculo.us: http://script.aculo.us
- DWR: https://dwr.dev.java.net/
- Backbase: http://www.backbase.com
- SmartClient: http://www.isomorphic.com
- Ajax.NET: http://ajax.schwarz-interactive.de/
- SAJAX: http://www.modernmethod.com/sajax/
- JSON-RPC: http://json-rpc.org/

DWR, JSON-RPC, Ajax.NET, and SAJAX are all examples of ORB-based Ajax frameworks. They allow you to map JavaScript methods to back-end services, treating the client-side JavaScript as though it could directly access your server-side objects.

UI Toolkit

Above, or potentially alongside, remoting toolkits we find JavaScript UI libraries. These give us the ability to use rich UI components and effects out of the box, but they differ in many ways.

Richer UI Components

Toolkits such as Dojo give us rich widgets like trees, tabbed panes and menus. These are self-contained, instantiable UI components that can be used to compose a rich, though still very "webish," application. The result is still unmistakably an HTML UI.

Web Application Toolkit

Toolkits such as SmartClient aim to give you widgets that build a UI that looks and feels the same as a native application on Windows or Mac OS X.These are useful if you are building an application that happens to be on the Web versus a website that uses a couple of UI effects and components. SmartClient, for example, features widgets that make the page look and feel exactly like a Windows NT application.

Markup Based

Backbase allows you to add rich components through a markup programming API. Your traditional HTML becomes something like this:

File 4

```
<xmp b:backbase="true" style="display:none;"
     xmlns:nav="http://www.backbase.com/site/nav">
  <s:event b:on="construct" b:action="show"/>

  <!-- everything that is never shown - in here -->
  <div style="display:none;">
    <s:include b:url="/chrome/bb3/skin.xml"/>
    <s:include b:url="/data/navigation.xml"/>
    <s:include b:url="/data/forms.xml"/>

    <!-- listeners for links to non-BDOC documents... -->
    <div id="forum">
      <s:event b:on="nav:show-page"
               b:action="select"
               b:target="id('forumBuffer')" />
    </div>
    <div id="/shop/">
      <s:event b:on="nav:show-page"
               b:action="select"
               b:target="id('shop_main_panel')" />
    </div>

    <!-- Contains references to protected buffers -->
    <!-- Trigger 'command' event to issue bufferdirty on them all -->
    <div id="clear_protected_trigger">
      <s:event b:on="command">
        <s:task b:action="trigger"
                b:event="command"
                b:target="*" b:test="*" />
      </s:event>
    </div>
  </div>

  <!-- Include shop -->
  <s:include b:url="/shop/shopIndex.html?cmd=index" />
  <!-- ... -->
</xmp>
```

Such a system could potentially enable a new generation of visual development tools. Part of the problem with such tools is the conflict between markup and code. Traditional JavaScript-based pages have caused problems for such tools because it is difficult to provide visual representations of code resources. An all-markup framework, on the other hand, would provide the right abstractions for these kinds of development environments. See, for example, the markup-based components in ASP.NET, Tapestry, and JavaServer Faces.

Simple JavaScript-Driven Effects

In Chapter 6, *Ajax UI, Part I*, on page 93, and Chapter 7, *Ajax UI, Part II*, on page 122, we'll look at several frameworks that use pure JavaScript and HTML to create extremely complex UI effects. These kinds of frameworks provide high-level abstractions on top of some meaty JavaScript, making the effects simple to implement in your application. The results are often completely cross-browser compatible and fail gracefully to static HTML in legacy browsers.

Ajaxian Web Frameworks

At the top of the tower are the web frameworks that are aware of Ajax. This is a growing group and covers all of the platforms. All the major players are represented: Java, .NET, Ruby, PHP, Python, Perl, etc.

Once again, the various frameworks offer different models for how you can work with them in an Ajaxian world.

Code Generation

The Ruby on Rails community jumped on Ajax like nobody else. They offer high-level Ruby helper functions that generate Prototype-based JavaScript code. WebWork2 is doing the same thing on the Java platform, utilizing the Dojo Toolkit as the base JavaScript framework. Many other frameworks are following suit, from Spring to CherryPy to PHP.

Component-Based

ASP.NET had Ajaxian components before there was Ajax. Other frameworks such as JavaServer Faces and Tapestry on the Java platform join ASP.NET by letting you use components that may happen to use Ajaxian techniques. In this world, you drag your DataTableComponent onto your designer view and start tweaking the property sheet for that component. Here you may see a checkbox for *autoupdate*. Simply checking that box will put this component in Ajax mode, and the rest is history.

5.2 Remoting with the Dojo Toolkit

Now that we've examined the landscape of available helper toolkits, we'll port Hector's CRM application to several of them to see how they work. Hector's CRM system is working OK with our low-level XMLHttpRequest

example from the previous chapter, but we want to move up the stack and utilize a remoting toolkit to abstract away browser compatibility issues and give us more options for controlling the remoting calls.

We will first port our application to use the Dojo Toolkit,[1] explaining choices that you have along the way and finally discussing more advanced features.

What Is the Dojo Toolkit?

Dojo is a *browser toolkit.* It is an open-source project that (to quote its marketing text) aims to "allow you to easily build dynamic capabilities into web pages and any other environment that supports JavaScript. Dojo provides components that let you make your sites more useable, responsive, and functional. With Dojo you can build degradable user interfaces more easily, prototype interactive widgets quickly, animate transitions, and build Ajax-based requests simply."

It is a full-featured toolkit that has many packages, including the following:

- dojo.io: The core package that we will look at in this chapter, which makes Ajax requests easy
- dojo.event: Browser-compatible event system
- dojo.lang: Support for mixins and object extension
- dojo.graphics: Support for nifty HTML effects such as fadeIn/Out, slideTo/By, explode/implode, etc)
- dojo.dnd: Drag-and-drop support
- dojo.animation: Animation effects
- dojo.hostenv: Support for JavaScript packages (think imports and includes instead of having to create script src="...")

Porting CRM to dojo.io.bind()

This chapter is all about the remoting layer, and in Dojo that means the dojo.io package. We are going to go from where we left off with the CRM application and replace the raw XMLHttpRequest object with a call to dojo.io.bind().

[1] http://dojotoolkit.org

autocomplete="off"

As part of cleanup, we added the HTML attribute autocomplete="off" on the city and state input values. This stops your browser from trying to do its own completion, which gets in the way when the value is being set by a return from Ajax.

Cleaning Up the JavaScript

Before we even get into Dojo, we should clean up the JavaScript a little and encapsulate the acts of assigning the city and state in the form and announcing errors. Until now these acts were hidden in the callback function used by XMLHttpRequest.

First, we create a function that assigns the city and state:

File 11
```
function assignCityAndState(data) {
  var cityState = data.split(',');
  document.getElementById("city").value = cityState[0];
  document.getElementById("state").value = cityState[1];
  document.getElementById("zipError").innerHTML = "";
}
```

Then we have a simple error assignment procedure:

File 11
```
function assignError(error) {
  document.getElementById("zipError").innerHTML = "Error: " + error;
}
```

With this simple abstraction, we will be able to use any remoting solution and reuse these functions.

Migrating to dojo.io.bind()

Now we get to the dojo.io package and in particular, a dojo.io.bind() function that encapsulates remoting. Everything you need to do with remoting can be done with this simple function. dojo.io.bind() takes a hash as input, using the values to initialize the underlying XHR object and register callbacks to other JavaScript functions.

We have to include Dojo in our HTML head element:

```
<script language="JavaScript" type="text/javascript"
        src="../scripts/dojo/dojo.js">
</script>
```

Let's look at the code that now does the Ajax request for the Zip data:

File 11

```
function getZipData(zipCode) {
  dojo.io.bind({
    url: url + "?zip=" + zipCode,
    load: function(type, data, evt){ assignCityAndState(data); },
    error: function(type, error){ assignError(error); },
    mimetype: "text/plain"
  });
}
```

The must-have element in the dojo.io.bind() parameter is the url key. In our example it will become /ajaxian-book-crm/zipService?zip=53711 if you are looking up a Wisconsin city.

The load key takes a function object as a callback. After the Ajax request has loaded a response, this function will be called (think of this as being the callback when the status from an XMLHttpRequest is the magic 4). In your callback you get access to the following:

- type, which tells you whether the response returned normally (load) or from an error condition (error).
- data, the response (harvested from XHR.responseText). This is the payload of the request.
- evt, a DOM event.

The error key handles errors, whereas load handles successful requests. The function callback gets access to the error message itself in its second function parameter.

The mimetype key is important. We have discussed how there are various styles of remoting and how you can choose to return HTML, JavaScript, or your own text. Here, we decided to use text/plain, get back the city/state information as the string Madison,WI, and split up for our usage.

Changing dojo.io.bind() to Use a Return Type of JavaScript

Now we have our Ajax request encapsulated in one simple dojo.io.bind() function call. This is a lot more elegant than using the raw XMLHttpRequest API, and we will soon see how we have access to features above and beyond the simple requesting and retrieving of data.

What if we wanted to talk to a service that responded directly with JavaScript for us to evaluate, instead of a proprietary string that we needed to parse? For example, instead of returning Madison,WI, the service could return this:

Generic Handle

Rather than separating the load and error handlers, in theory you can use one handler named handle. This is when you would use the type parameter and would probably check against it to see how you were called. We could have written the same example as follows:

```
handle: function(type, data, evt){
          if (type == "load") {
            assignCityAndState(data);
          } else if (type == "error") {
            assignError(error);
          } else {
            // could potentially handle other types!
          }
        },
```

```
document.getElementById('city').value = 'Boulder';
document.getElementById('state').value = 'CO';
```

Making this change is quite trivial with Dojo, and it will simplify our code even more. We can get rid of the assignCityState() call itself, and there is no need for a load() function, because Dojo will automatically load a JavaScript result from the server if we tell it via the MIME type text/javascript:

File 10

```
function getZipData(zipCode) {
        dojo.io.bind({
                url: url + "?zip=" + zipCode + "&type=eval",
                error: function(type, error){ assignError(error); },
                mimetype: "text/javascript"
        });
}
```

Notice that we added &type=eval to the URL to make sure that the server sent us back JavaScript this time.

Advanced Features of dojo.io.bind()

We hope at this point you have seen that it makes little sense to use the low-level API when you have a nice, clean, simple interface that Dojo gives you. It turns out that dojo.io.bind() can do a lot more for you. For one, it is able to do browser detection and makes sure that it finds the right XMLHttpRequest object for your browser. If it can't find one, it can

Transport Enforcement

Sometimes, we don't want graceful, transparent failover. If, for some reason, we must mandate that only certain kinds of post-back transport mechanisms be used, we can pass in our rule on the dojo.io.bind() call. If we want to enforce one transport only, we can do so by setting the following

```
transport: 'XMLHTTPTransport'
```

in the hash that we pass in.

drop back to iframes to do the deed. All of this happens transparently to the developer.

Submitting Forms

Dojo can submit a form asynchronously for you as well as access a given URL. All you need to do to submit your form is tell Dojo about the form element in your HTML via the following:

```
dojo.io.bind({
  url: "http://your.formsub.url",
  load: function(type, obj) { /* use the response */ },
  formNode:  document.getElementById('yourForm')
})
```

What if your form has a file upload as part of it? XMLHttpRequest can't do the job here, because it can't get the file from disk in a reliable way. Dojo has a solution, though, thanks to the pluggable I/O layer.

Browsers know how to send files, and we piggyback on that by selecting the IframeIO transport.

So, the simple solution is to place the following piece of code before you have forms with file uploads:

```
dojo.require("dojo.io.IframeIO");
```

Support for Browser Back/Forward Buttons

This feature is a gem. One of the issues with using XMLHttpRequest versus an iframe is that iframe events are placed in the browser history, while XHR events are not. This can cause an issue if a user clicks something

Uploading a File without a File!

You can actually upload content as though it is a file using the XMLHttpRequest transport.

In your dojo.io.bind(..) call, pass in a file object to the argument object itself:

```
file: {
  name: "upload.txt",
  contentType: "plain/text",
  content: "look ma! no form node!"
}
```

that causes an Ajax request that changes the page, and then they hit the back button assuming that it will take them to the state they were in before that request. Instead, they are taken to the page before the Ajax code (which could be away from your website!).

Dojo allows you to tie into the browser buttons, passing in the work that you want to do when a user clicks back or forward. In our CRM example, you could save the current city and state information and clean it out in the form when the user clicks back. Then, if the user clicks forward you could reset it into the form without having to go back to the server.

```
backButton: function() {
  saveCityState();
  cleanCityState();
},

forwardButton: function() {
  setupCityState();
},
```

How does Dojo do this? Is there a nice API that Firefox and IE give you to hook in? No. The actual implementation differs depending on the browser, but at a high level Dojo creates a hidden iframe, makes it go forward two requests, and then one back. Now, it is set up ready to do your bidding. If you click back, the onload event will call into your backButton callback. Ditto for the forward button.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

Pragmatic Ajax
`pragmaticprogrammer.com/titles/ajax`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/ajax.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com