

```

ajaxCaller.get("allPatterns.phtml", null,
               onServerResponse, false, "allPatterns");
setTimeout("ajaxCaller.get('ad.html',null,onServerResponse,false,'ad')",
           5000
        )
}

```

Alternatives

All-In-One

As was mentioned, the alternative—and the de facto choice—is to download everything in one go.

Related Patterns

Portlet

Portlets (Chapter 15) are good candidates for content-specific downloads. A portal's overall structure can be downloaded initially, and each Portlet's content then be downloaded (and refreshed) in parallel to the rest of the page.

Guesstimate

While a block is waiting to be loaded, you might populate it temporarily with a *Guesstimate* (see earlier).

Progress Indicator

While a block is waiting to be loaded, you might populate it temporarily with a *Progress Indicator* (Chapter 14).

On-Demand JavaScript

Like Multi-Stage Download, *On-Demand JavaScript* (Chapter 6) involves an initial download followed by further downloads later on. That pattern focuses specifically on JavaScript content rather than display and semantic content. It's also about downloading only when needed. The emphasis here is on downloading according to an initial schedule. Other patterns, such as *Microlink* and *Live Search*, cover on-demand downloading of display and semantic content.

Metaphor

Multi-Stage Download mirrors agile project management. If you can deliver some really useful things early on, why hold off until everything else is ready?

Fat Client



BrowserSide, Decoupled, DHTML, Fast, Responsive, Rich, Thick

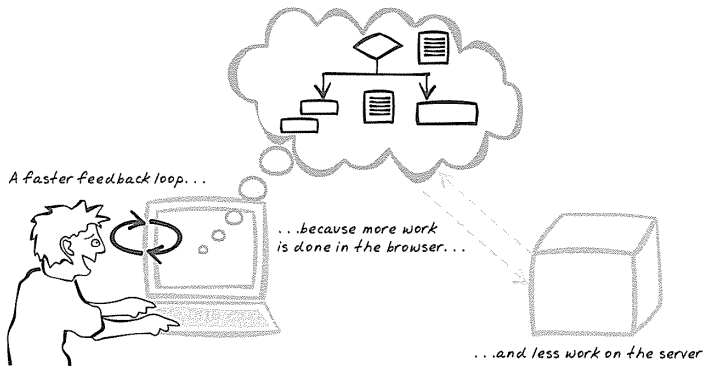


FIGURE 13-11. *Fat Client*

Developer Story

Dave's new mortgage application is very quick as it avoids server calls wherever possible. After some benchmarking, he discovered that optimized JavaScript is actually fast enough to run the financial algorithms in the browser, so that users now receive almost immediate feedback as they tweak parameters.

Problem

How can you architect the system to feel truly rich and interactive?

Forces

- The application should respond to user actions quickly—ideally, instantaneously.
- Responses from the server can be noticeably latent due to data transfer and server processing overheads.

Solution

Create a responsive Ajax App by performing remote calls only when there is no way to achieve the same effect in the browser. Whereas the server is traditionally the realm of more complex processing, this pattern suggests harnessing the browser for all it's worth. In the extreme case, it means running just about everything inside the browser, but that's not the main message here. The main message is to reconsider the traditional server-centric view of web architecture; for developers to be more conscious about decisions on matters regarding where the various components live. Placing more logic inside the browser gives you the opportunity to make the application more responsive.

Web apps usually have several layers:

- User interface
- Application logic
- Business logic
- Data access
- External system communication

The user interface is what the user sees and interacts with. The application logic concerns the dialogue between the user and the computer, the flow of information back and forth—for example, given a particular user action, what should the computer do next? The business logic concerns knowledge about the domain and the web site owner's practices and policies. The data access layer concerns reading and writing data to a persistent store. External system communication is also necessary in many enterprise systems, where outside systems play the role of information sources and sinks.

Conventional web apps concentrate UI code in the browser, using a mix of HTML and CSS, with some JavaScript used to perform a few basic UI tricks like placing keyboard focus or showing some animation or offering a dynamic menu. Everything else is usually managed on the server. This pattern reconsiders the balance by suggesting that some applications are better served pushing application and business logic into the browser (Figure 13-12).

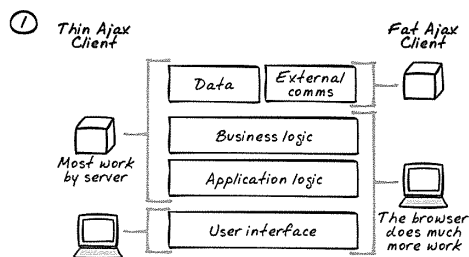


FIGURE 13-12. Responsibilities of a Fat Client

With application logic in the browser, an inversion of control occurs in which the browser now controls the flow of activity. When the user performs some action, it is the browser that decides how to respond, and it calls the server only if necessary, and for very specific services.

The advice to hold business logic in the browser is anathema to most literature on serious web site development and should not be taken lightly. Business logic in the browser has numerous problems:

- Programming in JavaScript can be difficult because many features familiar to mainstream developers are unavailable. The standard API is minimal and object-oriented facilities are based on prototypes rather than classes. Indeed, the usage of dynamic typing is also uncomfortable for many developers in the Java and C# camps.
- Portability is a further complication due to the number of platforms that must be supported, and their subtly inconsistent behaviors.
- Development is also a challenge. Coding, refactoring, debugging, automated testing, and monitoring, are standard in modern IDEs like IntelliJ Idea, but support at this level is minimal for JavaScript.
- Security is another constraint that forces logic server side. Savvy users can peruse code and data in the browser and can even tamper with what's sent back to the server. Obfuscation—making the code difficult to read—is a very weak defense.
- Business and application logic should be held on the server, where they can be reused by other clients.
- Web data generally can't be saved on the local machine.

Are any of these showstoppers? Let's look at how we can cope with each of these problems:

- JavaScript may be different from server-side languages, but many programmers are discovering it's a lot more powerful than previously assumed (<http://www.crockford.com/javascript/javascript.html>). Until recently, the main resources on JavaScript were “cut-and-paste this code” web sites, but there are an increasing number of books and resources that take the language seriously. The language has undergone a renaissance in parallel with Ajax. New idioms and patterns are emerging, along with a pile of cross-browser frameworks to augment the relatively bare native API. The recent introduction of JSAN (<http://openjsan.org>)—a repository of JavaScript libraries like Perl's CPAN—will only further reuse. Object-oriented concepts, including inheritance and aggregation, are indeed possible, if not in a manner familiar to all server-side developers. Dynamic versus static typing is a matter of taste that shouldn't affect a language's credibility; the dynamic approach of JavaScript may not be everyone's preference, but JavaScript is hardly an outsider here.
- Browsers are more standard nowadays, and many of the new libraries abstract away browser dependencies (see *Cross-Browser Component* [Chapter 12]). For example, over a dozen libraries offer a browser-independent factory method to retrieve XMLHttpRequest.
- Recent tools are making life a lot easier; some are discussed in *Development Patterns* (Part V).
- Security remains an issue, but it's possible to deploy the application so that security-critical functionality can be exposed server side in well-defined services, with the bulk of the application logic held locally and calling on those services as required. There might be duplication because browser-side validation is also important for quick feedback, using JavaScript to create a screening test before passing a request to the server.

However, there are ways to deal with such duplication if it's a serious issue. For instance, with server-side JavaScript, you can have the same script running in the browser and the server, or you can use some custom, declarative notation to capture business rules and have processors for it on both sides.

- It's true that certain business logic needs to be made available to multiple clients and therefore needs to be exposed as *Web Services*, in which case there's a good argument for keeping it inside the server. (It could also be executed in the server as well as the browser, though that's a fairly unusual practice.) Often, though, business logic is application-specific, and that's even more the case for application logic (notwithstanding grandiose plans for running the same app on mobile phones and desktops).
- Browsers cannot normally persist data locally, but a background thread can be used to periodically synchronize with the server. In the 1990s, many desktop applications forced users to explicitly save their work, whereas today, many desktop applications now save periodically and upon the user quitting. In the same way, it's now feasible for web apps to persist data transparently, without affecting workflow. If local persistence is critical enough to break away from "pure Ajax," some easily deployed Flash libraries are available (e.g., Brad Neuberg's AMASS; see <http://codinginparadise.org/projects/storage/README.html>).

None of this says that programming in JavaScript is more productive than programming in Java or C# or Ruby. In most cases, it isn't and never will be. Serious JavaScript programming is nonetheless a very real proposition. Professional software development involves choosing the language that fits the task at hand. Given the many benefits of browser-side coding, there are at least some circumstances that make JavaScript the language of choice for implementation of business and application logic.

Decisions

How will business and application logic be implemented in the browser?

Conventional applications use JavaScript for a little decoration. Even when a lot of JavaScript is used, it's often packed into tightly scoped library routines, to perform little tricks like validating a form's fields match regular expressions or popping up a calendar. These are just little detours on the well-travelled road between browser and server. In contrast, a Fat Client responds to many events itself, delegating to the server only if and when it deems necessary.

All this implies a very different usage of JavaScript, and there are some techniques that can help. It's beyond the scope of this chapter to cover them in detail, but here are a few pointers:

- It's easy to fall into the trap of "paving the cow paths"* by applying design styles from other languages to JavaScript. It's usually more productive to embrace JavaScript as a unique language with its own idiosyncratic features and usage patterns, and to be wary of applying architectural concepts from your favorite server-side language if they don't seem to fit in.
- JavaScript uses a prototype-based paradigm, which is worth learning about. Object-oriented concepts like inheritance are certainly possible, but it's important to appreciate how they fit in to the prototype model.
- Development and deployment practices apply as much to a rich browser application, and are as important as testing the server code, as discussed in Development Patterns (Part V).
- JavaScript runs slower than equivalent desktop applications, so optimization is important. Also, consider issues memory usage and be aware of browser-specific bugs that might lead to memory leaks.
- Break complex applications into multiple JavaScript files and consider managing them with *On-Demand JavaScript*.
- Reuse, Reuse, Reuse. As Appendix A shows, there's a proliferation of JavaScript frameworks and libraries becoming available.

Real-World Examples

NumSum

NumSum (<http://numsum.com>) is an Ajax spreadsheet that provides all the basic spreadsheet functionality in JavaScript: insertion and deletion of rows and columns; various formulas; the ability of a user to enter values into any cell; text-formatting (bold, italic, or underlining); text-justification; and embedded links and images.

The basic spreadsheet is implemented as a table. Table cells aren't actually editable, so the script needs to simulate an editable table. An event handler ensures that a cell becomes active when the user clicks on it, and that cell is then morphed to reflect subsequent user input.

Gmail

Gmail (<http://gmail.com>), like several of its competitors, presents a rich, Ajax interface to a mail system. Conventional web mail offerings rely heavily on form submissions and page reloads—a new page must be opened each time the user wants to start composing an email, finish composing an email, or open up an email message. With Gmail, the list of emails is always present and regularly refreshed. Partial edits are periodically saved as well. All of these activities are handled within the browser using a combination of Display Manipulation (Chapter 5) and Web Remoting (Chapter 6).

* "Paving the cow paths" refers to the way cows mindlessly follow an old path, continuing to walk around trees that aren't actually there anymore.

DHTML Lemmings

DHTML Lemmings (<http://www.oldgames.dk/freeflashgames/arcadegames/playlemmings.php>) shows how much can be achieved with some clever JavaScript hacking (Figure 13-13). It's a full-featured implementation of the Lemmings PC game utilizing a *Richer Plugin* only for sound—the entire game runs on DHTML, DOM, and CSS. The gameplay consists of a static image for the current level, with *Sprites* used to render the lemmings and other objects in the game. There is a periodic timer continuously checking the current state of each object, and DOM manipulation of the objects' properties is used to render their states at any time.

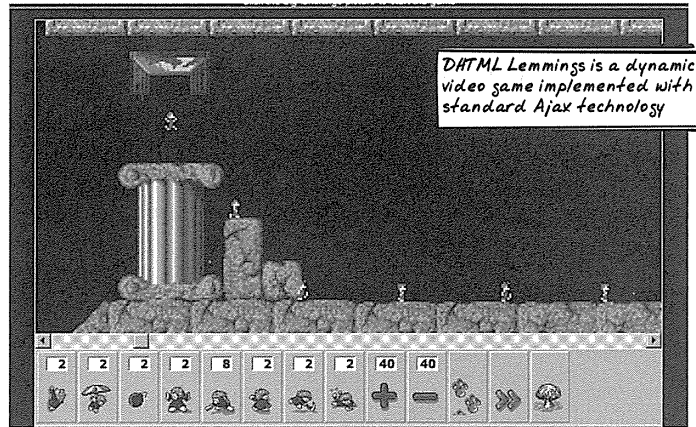


FIGURE 13-13. DHTML Lemmings

JS/UIX shell

JS/UIX shell (<http://www.masswerk.at/jsuix/>) is a demo of an in-browser Unix shell (Figure 13-14). In its current form, it's a demo only. But it's capable of all the basic operating systems commands and even includes a vi editor! The approach stands in contrast to browser-side terminal emulators, which are thin applications that pass characters back and forth. With appropriate server-side integration, the application could be made into a functional multiuser application. Command line remains the mode of choice for many power users, and this demo shows how an Ajax Fat Client can achieve it.

Code Example: AjaxPatterns Basic Wiki

The Wiki Demo (<http://ajaxify.com/run/wiki>) is based on Fat Client principles, though its functionality is fairly limited. All of the interaction and rendering occurs locally. The only interaction with the server is a periodic synchronization event, when the following occurs:

- Pending messages are uploaded, an example of *Submission Throttling* (Chapter 10).
- All current messages are retrieved from the server, so that messages from other users are shown. This is an example of *Periodic Refresh* (Chapter 10).

```

[vi@www.masswerk.at:2]$ date
Tue, 27 Dec 2005 15:47:13
[vi@www.masswerk.at:2]$ pwd
/home/vi
[vi@www.masswerk.at:2]$ ls

[vi@www.masswerk.at:2]$ touch test.txt
[vi@www.masswerk.at:2]$ ls
test.txt
[vi@www.masswerk.at:2]$ ls -la
drwxr-x--- 2 vi wheel ----- 2005/12/27 15:47:22 .
drwxrwxrwx 2 root wheel ----- 2005/12/27 15:43:18 ..
-rw----- 1 vi users 122 2005/12/27 15:43:18 .history
-rw-rw---- 1 vi users 0 2005/12/27 15:47:22 test.txt
[vi@www.masswerk.at:2]$ echo "Does this really work???"
Does this really work???
[vi@www.masswerk.at:2]$ echo "Does this really work???" > test.txt
[vi@www.masswerk.at:2]$ cat test.txt
Does this really work???
[vi@www.masswerk.at:2]$

```

JS/UNIX implements Unix shell functionality in pure browser-side JavaScript

FIGURE 13-14. JS/UNIX shell

There's a timer to ensure synchronization occurs after five seconds of inactivity:

```

var syncTimer = null;

function startPeriodicSync() {
  stopPeriodicSync();
  syncTimer = setInterval(synchronise, 5000);
}

function stopPeriodicSync() {
  clearInterval(syncTimer);
}

```

The timer starts on page load and is suspended each time the user begins to modify some text:

```

window.onload = function() {
  synchronise();
  startPeriodicSync();
}

function onMessageFocus(event) {
  ...
  stopPeriodicSync();
}

function onMessageBlur(event) {
  ...
  startPeriodicSync();
}

```

The script also contains a variable to accumulate pending messages whenever the user changes something:

```

var pendingMessages = new Array();
...
function onMessageBlur(event) {

```

```

...
if (message.value != initialMessageContent) {
    pendingMessages[message.id] = true;
...
}

```

The synchronization event builds up some XML to post to the server, based on the `pendingMessages` array that has been accumulated. If no changes are pending, it simply requests the current server state. If there are changes pending, it posts those changes as an XML message. Either way, the server will respond with an XML specification of its current state.

```

function synchronise() {

    var messageTags = "";
    for (messageId in pendingMessages) {
        var initialServerMessage = $(messageId).serverMessage;
        messageTags += "<message>";
        messageTags += "<id>" + messageId + "</id>";
        messageTags += "<lastAuthor>" + $("authorIdSpec").value + "</lastAuthor>";
        messageTags += "<ranking>" + initialServerMessage.ranking + "</ranking>";
        messageTags += "<content>" + escape($(messageId).value) + "</content>";
        messageTags += "</message>";

        $(messageId).style.backgroundColor = "#cccccc";
    }

    var changesOccurred = (messageTags!="");
    if (!changesOccurred) {
        ajaxCaller.getXML("content.php?messages", onMessagesLoaded);
        return;
    }

    var changeSpec = "<messages>" + messageTags + "</messages>";
    ajaxCaller.postBody
        ("content.php", null, onMessagesLoaded, true, null, "text/xml", changeSpec);
    pendingMessages = new Array();
}

```

The main application logic is virtually orthogonal to server synchronization. The only concession is the accumulation of pending messages, to save uploading the entire browser state each time. The timer and the synchronization process know about the core wiki logic, but the core wiki knows nothing about them and can therefore be developed independently. It's easy to imagine how the wiki could evolve into something with much richer browser-side features, like formatted text and draggable messages and so on.

Alternatives

Thin client

A thin client contains only basic JavaScript and frequently refers back to the server for user interaction. This is likely to remain the dominant style for Ajax Apps and is useful in the following circumstances:

- It's important to support legacy browsers. Graceful degradation is still possible with Fat Clients, but requires more work, and there's the potential for a lot of redundancy.
- Business logic requires sophisticated programming techniques and libraries that can only be implemented server side.
- Business logic involves substantial number crunching or other processing that can only be performed server side.
- Due to bandwidth and memory constraints, the application cannot be held locally.
- There is complex dialogue required with external services, and due to cross-domain security policies, those services cannot be accessed from the browser.

Desktop client

A desktop client runs as a standard application in the user's operating system, and connects to one or more servers using HTTP or any other protocol. Compared to a Fat Client, a desktop client can be richer and faster. However, the benefits of a web-enabled application are lost as a result. You can compensate for some of these benefits:

- Portability can be achieved with a platform-neutral programming platform.
- Centralized data can be achieved by ensuring all data is retained server side.
- Familiar user interfaces can be achieved by tailoring to each individual platform's look-and-feel.

In some contexts, a desktop client represents the best of both worlds: a rich client backed by a powerful server. In other situations, a Fat Client based on Ajax principles is a more appropriate sweet spot: a powerful server, a rich-enough client, and easy access through any modern web browser.

Related Patterns

Periodic Refresh

Periodic Refresh (Chapter 10) is a relatively unobtrusive way for the client state to be enriched with any new server-side information.

Submission Throttling

Submission Throttling (Chapter 10) is a relatively unobtrusive way for the server to be notified of any significant changes that have occurred in the browser.

Widgets

If the user is going to spend a long time in the browser, the various widgets (Chapter 14) should be adopted to improve the experience.

On-Demand JavaScript

Since Fat Clients tend to contain bulkier scripts, *On-Demand JavaScript* (Chapter 6) is useful for managing them.

Drag-And-Drop

Drag-And-Drop (Chapter 15) is familiar to many desktop users and helps to make a large application feel more natural. Whereas more conventional web clients rely on form-style interaction, Fat Clients can feel more rich if they support Drag-And-Drop.

Host-Proof Hosting

Where a Fat Client is seen as a replacement for a desktop equivalent, *Host-Proof Hosting* (Chapter 17) might be considered to avoid compromising on the superior security a localized solution sometimes offers.

Metaphor

Agile development methodologies such as Scrum aim for an “inversion-of-control.” Workers (like the Fat Client) are empowered to plan their own activities, and management’s role (the Server) is to offer whatever services workers request in order to conduct those activities.

Want to Know More?

- Google Groups Discussion: “Applications in the Browser or the Server?” (http://groups-beta.google.com/group/ajax-web-technology/browse_thread/thread/e40317a830ad841d/06483bf77587a62c#06483bf77587a62c)
- “JavaScript: The World’s Most Misunderstood Programming Language” by Douglas Crockford (<http://www.crockford.com/javascript/javascript.html>)
- “JavaScript is not the devil’s plaything” by Cameron Adams (<http://www.themaninblue.com/writing/perspective/2005/04/12/>)

Functionality and Usability Patterns

AJAX IS FUNDAMENTALLY ABOUT DELIVERING VALUE TO USERS AND THEIR ORGANIZATIONS. THE PREVIOUS parts have mostly covered technical issues that support developers, and now it's time to consider the user experience: how Ajax user interfaces will look, and what users will be able to do with them.

Many of the patterns combine display manipulation with the web remoting capability of Ajax. Others, such as *Slider* and *Rich Text Editor*, can be used in the absence of remoting, and still make an important contribution to the richness of Ajax interfaces. Most of the patterns here have been seen before in a desktop context and even on the Web in Flash or Java form; the point of documenting them here is to explore what works with standard browser technology and explain how these ideas can be implemented the Ajax way.

The first three chapters are all about user interface. Widgets (Chapter 14) introduces a number of widgets that are being woven into many Ajax interfaces. A higher-level perspective is taken by Page Architecture (Chapter 15), where the focus is on page layout, content breakdown, and techniques for exposing server-side content. With the popularity of Ajax, the Web is undergoing a rise in visual effects somewhat unprecedented on conventional desktop systems; the most common effects are described in Visual Effects (Chapter 16).

The stack of Ajax technologies opens up several possibilities that haven't been fully explored in a web context. The patterns in Functionality (Chapter 17) are a little more speculative than those in other sections, but more and more real-world usages for them are emerging. They are worth looking at as they might let you build things in a way you hadn't considered before.

CHAPTER FOURTEEN

Widgets

THE WIDGETS PROVIDED BY STANDARD HTML—TEXT INPUTS, RADIOBUTTONS, AND SO ON—HAVEN'T changed much for ten years and are getting a little tired. So many people have been building their own widgets on top of the standard HTML offerings, and a few patterns have emerged. It's unlikely you'll need to build these widgets yourself, because library implementations are available for almost all of them. Nevertheless, it's worth it to consider when to use them and how to configure them.

The chapter starts with smaller, isolated widgets and builds up to more complex components. *Slider* is “the widget HTML forgot”—a typical slider widget so useful it's surprising it hasn't yet become standard HTML. *Progress Indicator* is a simple message or animation shown while waiting for a web remoting call. *Drilldown* is a control for selecting an element within a hierarchy.

Traditional HTML widgets are turbo-charged in the next couple of patterns. *Data Grid* is a “table on steroids,” with tools for querying and restructuring that will be familiar to users of spreadsheets and database report tools. And *Rich Text Editor* is a “textarea on steroids,” with options such as font size and color similar to those of a modern word processor.

Suggestion is similar to the traditional combo-box, a mixture between browsing and searching that relies on web remoting to locate a set of options against a partial text input. *Live Search* works similarly but shows search results rather than helping to complete a text field. It also works with nontext controls.

Finally, *Live Command-Line* and *Live Form* are higher-level patterns that tie together various widgets and patterns. The former is largely speculative and explores techniques to help people learn and use command-line interfaces within the Web. The latter is a more proven Ajax Pattern—a form that keeps changing in response to user input.

Slider



Continuous, Lever, Multiple, Range, Slider

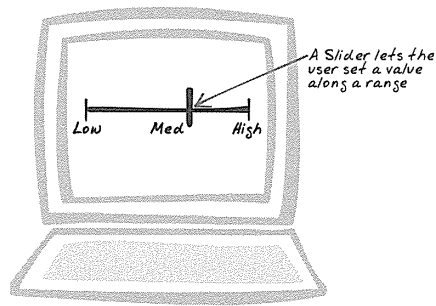


FIGURE 14-1. Slider

Goal Story

Stuart is rating his lecturer online. Each answer lies somewhere between “Strongly Disagree” and “Strongly Agree.” Unlike conventional surveys, the scale is continuous, because Stuart uses a Slider to set a precise value along the spectrum.

Problem

How can users specify a value within a range?

Forces

- Most data is constrained to reside within a range.
- Prevention is better than cure; the user interface should be designed such that the user is forced to choose a value within the range instead of the interface checking and providing an error message later on.

Solution

Provide a Slider to let the user choose a value within a range. The user drags a handle along one dimension to set the value. As a variant, two handles on the same Slider can allow the user to choose a range instead of a single value.

Standard HTML contains two related input controls: radiobuttons and selectors. Each of these lets you specify a value from a fixed list of choices. The biggest advantage of a Slider is that the data range can be continuous as well as discrete. Of course, “continuous” is an approximation, since you can only have as many choices as there are pixels in the Slider range. But with most Sliders, that means you have hundreds of unique values, which is continuous for most intents and purposes.

A Slider can also show discrete data by identifying several points along the range. When the user releases the handle, the Slider jumps to the nearest point. Why use a Slider when radio buttons and selectors already support discrete data? A Slider provides a better visual indication that the data resides in a spectrum. Also, it’s often faster because the user can click anywhere in the region as opposed to aiming precisely at a radiobutton or opening up a dialog box.

A further benefit of Sliders is their excellent support for comparing similar data. When several Sliders share the same range, they can be placed in parallel to show how the variables differ. For example, a product survey could ask questions such as “Were you happy with the price?” and “How easy was it to start using?” The answers lie on a different scale but ultimately map to the same range from “Unhappy” to “Happy.” Placing these horizontal rows in parallel helps the user stick to the same scale and compare each factor to the others.

Because Sliders aren’t standard controls, you’ll need to either use a library or roll your own. Typical control mechanisms include the following:

- Dragging the handle moves it directly.
- Clicking on either side of the handle moves it a little in that direction.
- While the Slider has keyboard focus, pressing left and right arrows move it in either direction. It’s sometimes useful to offer an accelerator key such as Shift, which, held down at the same time as the arrow, speeds up the handle movement. Keyboard controls are particularly important when the control is part of a form.
- Receiving notifications from external sources. The Slider is sometimes synchronized with another object, so if that object changes, the Slider must update too.

A typical implementation separates the scale from the handle. The scale consists of a line, possibly with some notches and labels. The handle is usually an image, with the `zIndex` property set to place it “in front of” the main Slider. There are several event handlers required to support all of the mechanisms above, and the Slider handle itself is manipulated using techniques discussed in *Drag-And-Drop* (Chapter 15). In addition, movements will often trigger other activity, such as an *XMLHttpRequest Call* or a change to another page element.

Decisions

How will you orient the Slider?

There are two options for orientation: horizontal or vertical. Following are a few considerations:

Data type

Sometimes the nature of the data dictates which option is more logical. For instance, Google Maps uses a vertical Slider for zoom, which corresponds to a feeling of flying toward and away from the map as you zoom in and out.

Layout

Aesthetic appearance and space conservation are important. Many forms will feature horizontal Sliders because they fit well underneath questions. Vertical Sliders would lead to a lot of whitespace.

Proximity

Where the Slider controls something else on the page, you'll probably want to place them near each other, which might dictate orientation.

Comparison

As mentioned earlier in the "Solution," Sliders work well when placed in parallel, which means a common orientation.

What scale will you use?

There are quite a few ways to present a variable, and the choice will depend on the nature of the data and what users are comfortable with. Examples include:

- Qualitative descriptions ("Low", "High"). This might seem suited only to discrete ranges, but the labels can also be used as markers within a continuous range.
- Absolute values.
- Percentages (typically ranging from 0 to 100).

How will you present the scale?

There are various strategies for presenting the scale:

- Provide no labels; rely on context and conventions. For instance, users—at least in western countries—usually assume Sliders increase in value to the right.
- Provide just a label at either end.
- Provide labels at several points along the range.

The labels are usually shown alongside the Slider, but to conserve space, you can sometimes show them inside it.

Real-World Examples

Yahoo! Mindset

A product of Yahoo Labs, Yahoo! Mindset (<http://mindset.research.yahoo.com/>) lets you tweak search results with an unusual type of Slider that ranges from “shopping” on one end to “researching” on the other (Figure 14-2). Pull the “Ajax” results towards “shopping” and you’ll see links to cleaning products and football. Pull it to the right and you’ll see some information about web design (and, it must be said, more football). Also of note: the Slider is “Live”—each time you change it, the results are automatically updated via an *XMLHttpRequest Call*.

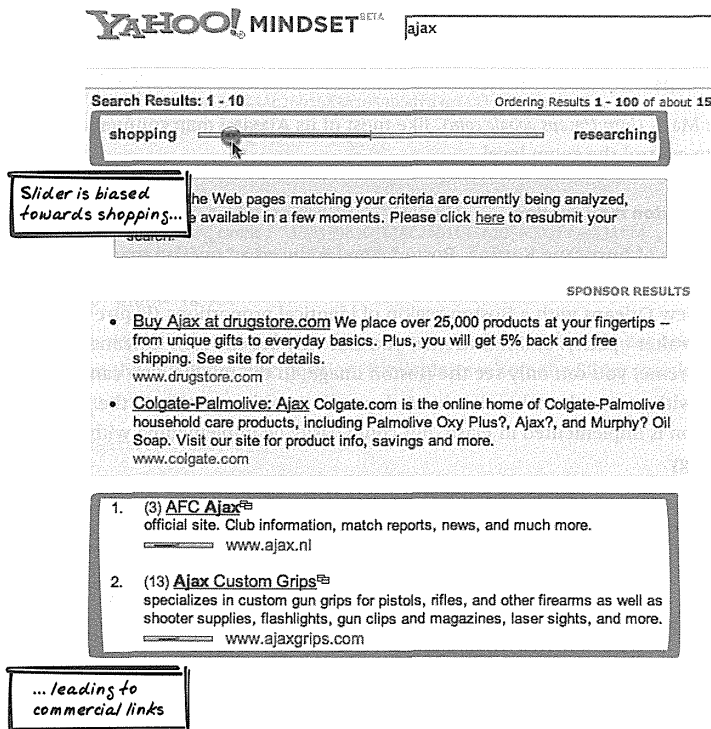


FIGURE 14-2. Yahoo! Mindset

Amazon Diamond Search

Amazon Diamond Search (<http://www.amazon.com/gp/gsl/search/finder/002-1527640-2908837?%5Fencoding=UTF8&productGroupID=loose%5Fdiamonds>) presents several elaborate Sliders (see Figure 14-5). Each lets you specify a range, which acts as a filter for the diamond search. There are several innovative aspects of the presentation, discussed next.

- The labels are dynamic. As you drag the handles, the labels show exactly the value that's been specified.
- There's a graphic inside the Slider, which represents the variable in each case. Price, for example, is shown as an increasing histogram, and Cut shows several diamonds of decreasing cut quality.
- The Slider graphic is faded outside the selection region.
- Each Slider has a *Microlink* (Chapter 15) that opens up an explanation about that variable.
- The Sliders are horizontal and parallel to each other. Unfortunately, the directions aren't aligned—price and carat increase from left to right, but cut quality increases from right to left.

Google Maps

Google Maps (<http://maps.google.com>), like most of its Ajaxian map counterparts, uses a Slider to control the zoom level.

Katrina-Boston map overlay

In the wake of hurricane Katrina, Boston.com produced an overlay map (http://www.boston.com/news/weather/hurricanes/multimedia/globe_map/) combining the flood-affected area of New Orleans with a Boston region of identical proportions (Figure 14-3). A Slider is used to alter opacity: at one extreme, you can only see the New Orleans image; at the other extreme, you can only see the Boston image. In the middle, you can see both images, with one or the other dominant depending on the position of the Slider. The application is implemented in Flash, but could easily be implemented with standard Ajax technology.

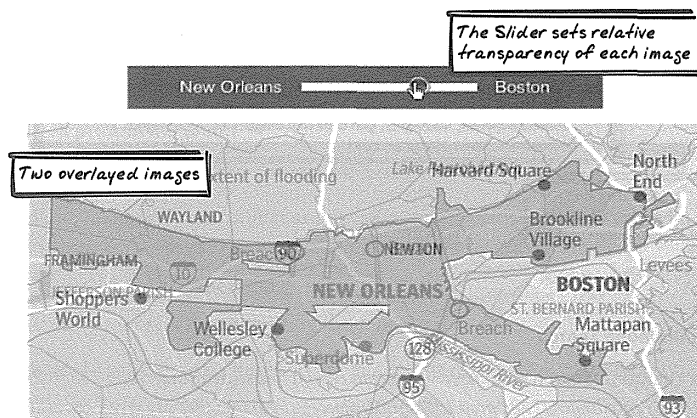


FIGURE 14-3. Katrina-Boston overlay map

WebFX Slider Demo

The WebFX Slider Demo (<http://webfx.eae.net/dhtml/slider/slider.html>) shows a few different Sliders, in both horizontal and vertical orientations. You can set the Slider directly and also update it with some text inputs.

Code Example: Yahoo! Mindset

The Yahoo! Mindset Slider (<http://mindset.research.yahoo.com/>) is created with several images. There are images for the vertical lines at the center and extremes. The main horizontal bar is created with a single vertical line; it's 1 pixel wide, but the image width is set to 150 pixels, thus creating a horizontal bar. There's a handler registered for the click event (which pulls the handle in that direction):

```

```

The Slider handle, called sliderball, is also an image:

```

```

A drag function is registered to handle moving the mouse after the handle has been selected. Based on the current mouse position, it calculates where the Slider should be and calls a function to move it:

```
function dragStart(e, sID, q) {
    ...
    document.onmousemove = function(e) { drag(e); };
    document.onmouseup = function(e) { endDrag(e); };
    ...
}

function drag(e) {
    ...
    var relativePos = e.clientX - sliderOffset;
    ...
    moveSlider(relativePos);
}
```

The moveSlider function redraws the handle based on its relative position (positive or negative offset from the center):

```
function moveSlider (relativePos) {

    var centerPoint = (maxRight - minLeft) / 2;
    var centerBuffer = 5;

    //the ball position is relative
    var ballPos = (-(maxRight - minLeft)) + (relativePos-(ballWidth/2));
    document.getElementById('sliderball').style.left = ballPos+'px';
```



```
...
}
```

Finally, when the Slider is released, the handle's position is finalized and an *XMLHttpRequest* is issued to bring the results in line with the new value:

```
function endDrag(e) {
    ...
    var relativePos = e.clientX - sliderOffset;
    drag(e);
    ...
    var sliderURI = "/searchify/slider?UserSession="+sessionID+
                    "&SliderValue="+threshold+"&query="+query;
    (Sends XMLHttpRequest to SliderURI)
}
```

Alternatives

Conventional controls: radiobuttons and selector field

As mentioned in the solution, a Slider performs similar functionality to radiobuttons and selectors.

Related Patterns

Drag-And-Drop

The Slider handle is usually manipulated with a *Drag-And-Drop* (Chapter 15) action.

Metaphor

Sliders are a UI metaphor based on physical sliders in control devices such as audio-visual consoles.

Progress Indicator



Activity, Feedback, Hourglass, Meter, Progress, Waiting

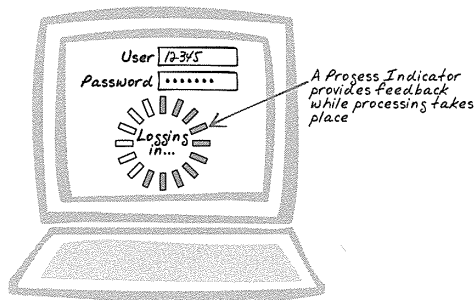


FIGURE 14-4. Progress Indicator

Goal Story

Reta has made a sale and is typing in the details. She completes the credit card details and proceeds down the page to delivery dates. Meanwhile, the status area has appeared beside the credit card details. While she completes the rest of the form, the status area continues to update with the current verification stage—initially “submitted,” then “responding,” then “verified.”

Problem

How can you provide feedback while waiting for server responses?

Forces

- Ajax Apps often make *XMLHttpRequest Calls* (Chapter 6) to query and update the server.
- To provide good feedback, you should provide an idea of which page elements are affected by any update.
- Users like to feel in control, and that control is taken away when the user is waiting for a response. It’s not just the delay that should be minimized, but the *perceived* delay.

Solution

Indicate the progress of server calls. You can’t always reduce delay, but you can include a Progress Indicator to ease the pain. A Progress Indicator helps maintain the user’s attention, improves the user’s understanding of how the system works, and also communicates that the system is still alive even if a response hasn’t yet occurred.

The Progress Indicator is typically introduced to the DOM once an *XMLHttpRequest Call* begins and removed when the call has returned. The easiest way to detect whether the call has returned is using the *XMLHttpRequest* callback function. An indicator need not relate to a single call—it can show progress for a sequence of related calls.

Sometimes it’s a *Popup* (Chapter 15) element instead of a new element directly on the page. A popular idiom is a small opaque Popup on the corner of the page showing just a word or two (i.e., “Saving...”, “Loading...”).

For shorter delays, typical Progress Indicators include:

- A small message like “Updating document”
- An animated icon (there’s now a library of open source Progress Indicator icons available at <http://mentalized.net/activity-indicators/>)

For longer delays, the following can be used:

- A meter showing how much progress has been made
- An estimate of time remaining

- A sequence of messages indicating what's happening at present
- Content that's engaging but not directly related to the progress, such as "Tip of the Day" or a canned graphical depiction of system activity

Of course, you can combine these approaches. Generally speaking, some form of unobtrusive animation is worthwhile in any Progress Indicator, because it at least tells the user that something's happening, even if progress is temporarily stuck. In addition, longer delays should usually be completed with a visual effect such as *One-Second Spotlight* (Chapter 16), since the user's focus has probably moved elsewhere by that stage.

Note that one form of indicator to *avoid* is changing the cursor. Many traditional GUIs switch over to a "rotating hourglass" or related icon during delays. That's probably inappropriate for Ajax because it's something the actual browser software will do too—e.g., while loading a new page—so it's likely to create confusion.

Decisions

What sort of Progress Indicator will you use?

A well known set of guidelines is summarized in Jakob Nielsen's Usability Engineering (<http://www.useit.com/papers/responsetime.html>). Following is a quick summary:

- If the delay is less than 0.1 second, the user will feel it's instantaneous. No feedback necessary.
- If the delay is between 0.1 second and 1 second, the user will notice it but it won't break their flow of thought. No feedback necessary.
- If the delay is between 1 and 10 seconds, the user's flow of thought is interrupted as he awaits the response. Basic feedback necessary; i.e., an indication *that* a delay is occurring. Ajax examples include animated icons and simple text messages.
- If the delay is greater than 10 seconds, the user will want to proceed to other tasks. Detailed feedback necessary. Ajax examples include progress meters and messages showing current state.

The precise figures may require some adjustment and I suspect a web context requires them to be dropped a bit. For example, users will probably want some feedback for a delay of 0.5 second rather than 1 second, and more detailed information is probably appropriate after 2–3 seconds rather than 10 seconds.

Bruce Tognazzini also offers some useful guidelines (<http://www.asktog.com/basics/firstPrinciples.html#latencyReduction>).

How will you provide feedback during longer delays?

For longer delays, you need to help the user track how much progress has been made, typically using a progress meter that shows percent complete. Sometimes, a long delay can come from a single *XMLHttpRequest Call*, because although the network transfer may be

quick, the backend processing might not be. For example, the call might trigger a major database operation.

You probably won't get any useful information about its progress by monitoring the `responseText` component of `XMLHttpRequest`. The `responseText` tends not to populate in a linear fashion, for two reasons. First, there are usually backend calculations involved, during which no output can occur. Thus, output tends to happen either in bursts or all at the end. Second, the output is often compressed using the standard HTTP content encoding facility, and the compression algorithm will force data to be outputted in bursts. The `XMLHttpRequest`'s `readyState` won't tell you very much either. For reasons described in *XMLHttpRequest Call* and *HTTP Streaming* (Chapter 6), tracking support is inconsistent across browsers.

So if you can't monitor the progress of an `XMLHttpRequest` Call, how can you help the user understand how much progress has been made? One thing you can do is *Guesstimate*: predict the total time, and start running a timer to monitor how long it has been since the call began. The prediction of total duration need not be hardcoded every time; you could have the application track download times and reflect them in future estimates. This sort of thing is quite common; e.g., in the download-time estimates given by a web browser.

If you want more accuracy, introduce a second monitoring channel. While the primary request takes place, a sequence of monitoring requests are issued to ask the server for a progress estimates. For example, the server might be looping through 1,000 records, running a transformation on each and saving it to the database. The loop variable can be exposed in a second *Web Service* so that the browser monitoring can inform the user.

Not all Progress Indicators concern a single `XMLHttpRequest` Call. Indeed, those requiring a progress meter are longer processes, likely incorporating several `XMLHttpRequest` Calls. With those, you have much better opportunity for real-time progress monitoring; each time a call returns, further progress has occurred. In a simple model, you can show that progress is 50 percent complete when two of four calls have returned.

Real-World Examples

Amazon Diamond Search

Amazon Diamond Search (<http://www.amazon.com/gp/gsl/search/finder/002-1527640-2908837?%5Fencoding=UTF8&productGroupID=loose%5Fdiamonds>) is a *Live Search* that shows a Progress Indicator while updating the number of results (Figure 14-5). The indicator is a simple animation depicting a block moving back and forth with a "LOADING RESULTS" message. One nice design feature is the placement of the Progress Indicator on the result status. It replaces the results once searching has begun and remains until the new results are shown. Thus, it serves to invalidate the previous results at the start and focuses the user's attention on the new results at the end.

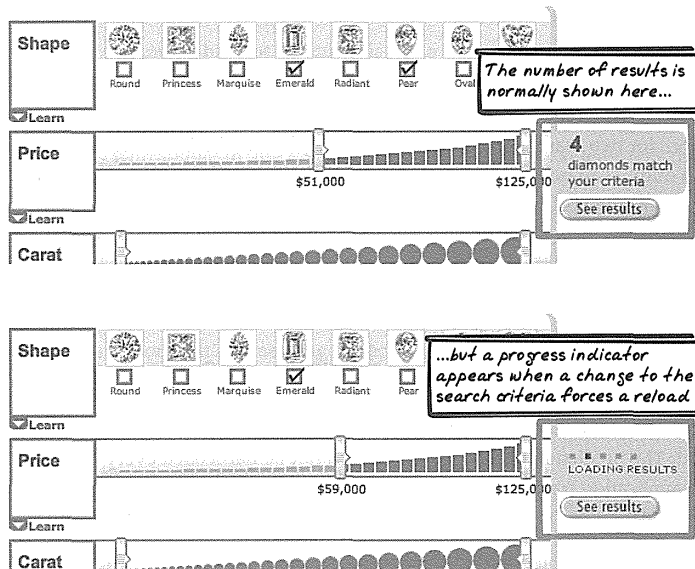


FIGURE 14-5. Amazon Diamond Search

Amazon Zuggest

Like Amazon Diamond Search, Francis Shanahan's Amazon Zuggest (<http://www.francisshanahan.com/zuggest.aspx>) is a *Live Search* that shows a Progress Indicator while searching is underway. In this case, the indicator is a text message such as "Searching... beat." It occupies its own fixed area, and when searching is complete, morphs into another message: "Done!"

Protopage

Protopage (<http://www.protopage.com/>) is an Ajax portal. Each time you make a change, such as dragging a *Portlet* (Chapter 15), an opaque "Saving..." message appears in the bottom-right corner. This is a good model for a *Fat Client* application, in which server synchronization should be unobtrusive.

TalkDigger

TalkDigger (<http://talkdigger.com>) simultaneously calls several search engines, showing a Progress Indicator on each result panel. It's interesting because it shows how to use Progress Indicators in a *Multi-Stage Download* (Chapter 13) situation.

Kayak

Kayak (<http://kayak.com>) illustrates how to handle a longer delay. When you search for a trip, it creates a result page with several forms of progress feedback:

- The number of search results so far.
- A progress meter that fills up from left to right.
- A sampling of web sites that are being searched.
- A nice graphic depicting a retro, pre-electronic airport display board. Initially, each character is random. As the search of a particular web site proceeds, the random characters are replaced with the correct character for that web site. Meanwhile, the characters that remain unpopulated continue to flicker from one random character to another. Once all characters have been corrected and the web site name displays correctly, the display becomes random again and starts revealing another web site. All this is an excellent example of a graphic that is engaging and at the same time indicative of the processing that's occurring.

The Pick'em Game

The Pick'em Game (<http://www.pickemgame.com/welcome/picksheet>) is an office pool game allowing you to predict this week's football winners. It provides a form where you declare a prediction and confidence level for each game. Above the form is a small instruction message, and when data is being saved, it morphs into a Progress Indicator. The indicator is a spinning disk and an "Updating Pick Sheet" message. (The demo page doesn't perform a real remote call.)

Code Refactoring: AjaxPatterns Progress Indicator Demo

This demo (<http://ajaxify.com/run/sum/progress>) introduces a progress display to the Basic Sum Demo (<http://ajaxify.com/run/sum/progress>). It's a simple animated GIF that shows up while waiting for the sum to return.

An `img` tag for the animation is present in the initial HTML:

```

```

The script will toggle the image's CSS class depending on whether you're in waiting mode or not. The stylesheet ensures that it's visible when waiting and invisible when not:

```
.waiting {  
    visibility: visible;  
}  
  
.notWaiting {  
    visibility: hidden;  
}
```

With the styles defined, the script just has to flick the CSS class back and forth as the waiting status changes:

```
function submitSum() {  
    $("progress").className = "waiting";  
    ...  
}  
  
function onSumResponse(text, headers, callingContext) {  
    $("progress").className = "notWaiting";  
    ...  
}
```

Related Patterns

Status Area

A Progress Indicator is usually presented as a *Status Area* (Chapter 15).

Popup

The Progress Indicator can sometimes reside in a *Popup* (Chapter 15).

One-Second Spotlight

Once a long process has completed, use a *One-Second Spotlight* (Chapter 16) to point this out to the user.

Guesstimate

Sometimes you don't know how long a task will take or how much progress has been made so far. A sloppy guess is better than nothing at all, so make a *Guesstimate* (Chapter 13) of the progress.

Distributed Events

When a call comes in, you need to close off the Progress Indicator. There's a risk here that you'll end up with a single function that mixes Progress Indicator stuff with the logic of processing the response. Separate that logic using *Distributed Events* (Chapter 10).

Metaphor

Banks and post offices often use ticket-based queueing systems, showing the number that's currently being served.

Want to Know More?

- Gnome Guidelines, Chapter 7: Feedback (<http://developer.gnome.org/projects/gup/hig/1.0/feedback.html>)
- AskTog.com First Principles of Interaction Design, Latency Reduction (<http://www.asktog.com/basics/firstPrinciples.html>)

Drilldown



Drilldown, Menu, Progressive

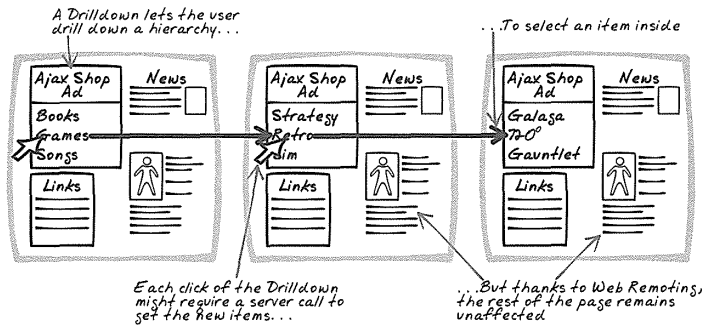


FIGURE 14-6. Drilldown

Goal Story

Pam is booking a trip on the corporate travel planner. She sees a form with the usual fields and clicks on location. Suddenly, a list of cities fades in beside the form, and Pam selects Paris. Beside the city list, a third list appears, this one showing approved hotels. Pam chooses the Hilton, and both lists disappear. The location field now contains “Paris Hilton” as Pam had intended.

Problem

How can the user select an item in a hierarchical structure?

Forces

- Applications and web sites are often arranged in hierarchies. To navigate, users need to choose a page from within the hierarchy.
- Hierarchy navigation should be fast.

Solution

To let the user locate an item within a hierarchy, provide a dynamic Drilldown.

The Drilldown allows for navigation through a hierarchy and ultimately for an item to be chosen. At each level, there are several types of elements in the Drilldown:

Current category

A read-only name for the present level.

Individual items

Items the user can choose in order to end the interaction.

Child categories

Deeper levels the user can drill down to.

Ancestor categories

Parent category and above that let the user “drill up.”

In some hierarchies, items and categories are mutually exclusive: items only exist at the edges—or leaves—of the hierarchy. Even when that’s the case, items and categories should be distinguished for the sake of clarity. The upward navigator goes by different names, but a general guideline is that it should tell the user which category that she is going back to.

Typical applications include navigating and filling out a field by drilling down a hierarchy of candidate items.

As the user drills down the hierarchy, will you show each progressive level? There are three main options (Figure 14-7):

- Keep the Drilldown in a fixed area. Each time the user clicks on a category, the Drill-down morphs to show only that category. This has the benefit of preventing page rearrangement.
- Show all levels at once. Each time the user clicks on a category, expand out the Drill-down region to add a new list for each category.
- As a variant on the second approach, all levels are shown, but the submenus are rendered in front of the rest of the document, so the menu is effectively modal: nothing else can be done from the time the menu is opened to the time the selection is made. This approach is similar to choosing an application from the Windows Start menu.

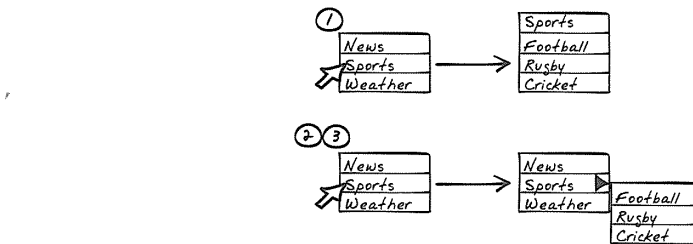


FIGURE 14-7. Drilldown styles

The first two options are modeless; the third is modal. Modeless interaction works well with Ajax, where it’s possible to dynamically update, without page refresh, the menu as well as any surrounding content related to the user’s selection. Specifically:

- Each navigation through the Drilldown can result in a server call to fill the next level with the latest content or even auto-generate new content.

- The display throughout the application can change to reflect the state of the Drilldown, giving the user a “tour” through the hierarchy. As a user drills down from the top level to a deep category, the application is always synchronized with the current level, so the display reflects the medium-depth categories along the way. Because the interaction is modeless, the user is then free to stay on those categories.
- The user can iterate between drilling down and performing other tasks. Imagine the user is a human resources clerk who needs to use a Drilldown to select an “employee of the month.” She’s already decided on the region and drills down to show all of the employees there. Now that she can see each candidate, she can go perform some research in another region of the page, or on another web site. When she returns, the Drilldown is still there, waiting to be completed.

Decisions

Will you call on the server each time the user navigates through the Drilldown?

Sometimes the entire hierarchy is loaded as a one-off event. Other times, categories can be pulled down as required using an *XMLHttpRequest Call* (Chapter 6). The choice is governed by two things:

- How big is the hierarchy? The more items, and the more information per item, the less desirable it is to transfer and cache the data.
- Is the hierarchy subject to change? In this case, you’ll need to retrieve fresh data from the server at the time a category is opened up. In a more extreme case, the server might even generate the hierarchy data on demand. For instance, an RSS feed aggregator might present a Drilldown with categories such as “sports feeds” or “politics feeds.” The contents of these will be generated at the time the user drills down to a particular feed.

Real-World Examples

Betfair

Betfair (<http://betfair.com>) uses a Drilldown to locate an event you wish to bet on (Figure 14-8). At the top level, the “All Markets” Drilldown contains various categories in alphabetical order, from “American Football” to “Special Bets” to “Volleyball.” Clicking on “Special Bets” yields several countries, along with a “General” category, and you can continue to drill down to the list of bets. Clicking on one of those bets sets the main page content.

“All Markets” is one of two Drilldowns. The other is “My Markets,” a personalized Drill-down available to registered users.

Backbase portal

The Backbase portal demo (<http://projects.backbase.com/RUI/portal.html>) contains several independent *Portlets*. Of interest here is the “local directory” Portlet, which is actually a

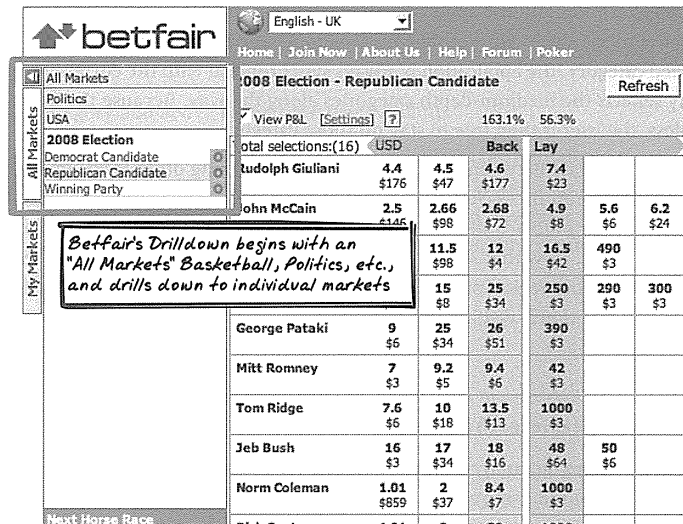


FIGURE 14-8. Betfair

Drilldown with top-level categories such as “education” and “health” drilling down to relevant links.

OpenRico accordion widget

The OpenRico framework (<http://openrico.org/rico/demos.page?demo=ricoAccordion.html>) includes an accordion widget. Clicking on a category bar reveals the content. Because it lacks a deep hierarchy of categories, it’s technically not a Drilldown, but it’s a good solution when you do have a flat set of categories.

Code Refactoring: AjaxPatterns Drilldown Portal

Overview

The basic Portal Demo (<http://ajaxify.com/run/portal>) illustrates *Multi-Stage Download* (Chapter 13), showing how different content blocks can be downloaded in parallel. One of those is a block of links to Ajax resources. To keep the block small, only a few links are present. But is there any way we could keep the link block physically small while offering a large number of links? Of course there is...a Drilldown will occupy roughly the same space, yet with a little interaction, the user will be able to navigate through a large collection of links.

The Drilldown Portal Demo (<http://ajaxify.com/run/portal/drilldown>) introduces a Drilldown. A Category can contain any combination of Categories and Links (Figure 14-9). For instance: The top-level category, “All Categories,” contains only Categories, the “Websites” category contains just links, and the “Overviews” category contains some overviews as well as a subcategory, “Podcast Overviews” (Figure 14-10). Categories and links are

rendered similarly, but not identically. Each category menu includes a link to the previous category level.

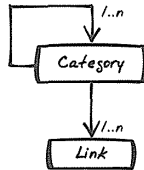


FIGURE 14-9. Categories and Links: the Composite Pattern

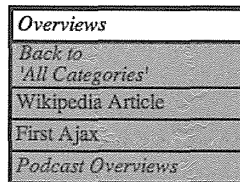


FIGURE 14-10. Drilldown with Categories (Italicized) and Links

The basic interaction sequence works like this:

1. User clicks on a category.
2. A callback function detects the click and issues an XMLHttpRequest query on the desired category.
3. Server responds with the new category name, its parent (null in the case of the top-level category), and each category and link in the Drilldown. All of this is in XML format.
4. Browser replaces existing Drilldown with a Drilldown based on the new category information.

On page load, the Drilldown is blank, and the browser requests information for the top-level category. The server response triggers the creation of a new Drilldown.

Browser-side implementation

The initial HTML just shows a blank div. The width is constrained because the Drilldown's parent container has a fixed width:

```
<div id="drilldown">Loading ...</div>
```

On page load, the top-level category is requested:

```
retrieveCategory("All Categories");
```

Then begins the standard process of requesting a category with XMLHttpRequest, then rendering the menu accordingly. This occurs not only on page load, but every time the user clicks on a category within the drop-down.

The server returns an XML file with category data.* The specification contains the name, the parent name, and a list of items. Each item is either a category or a link. Note that only the information for this level of the Drilldown is provided.

```
<category name="Overviews" parent="All Categories">
  <items>
    <link>
      <url>http://en.wikipedia.org/wiki/AJAX</url>
      <name>Wikipedia Article</name>
    </link>
    <link>
      <url>http://www.adaptivepath.com/publications/essays/archives/000385.php</url>
      <name>First Ajax</name>
    </link>
  </items>
</category>
```

The browser then parses this XML, using the standard DOM API, to produce HTML for the Drilldown. Links are shown as standard HTML anchor links; categories are Div elements. Event handlers ensure that when a category is clicked, including the “Back To (previous)” category, the browser will kick off another retrieve-and-render cycle:

```
function onDrilldownResponse(xml) {

  var category = xml.getElementsByTagName("category")[0];
  var html="";

  var categoryName = category.getAttribute("name");
  html+="

* For example, you can see the data for the “Overviews” category at http://ajaxify.com/run/portal/drilldown/drilldown.phtml?categoryName=Overviews.



348 CHAPTER FOURTEEN



Amazon/DivX  
Exhibit 1028  
Page 366


```

```

    }

    $("drilldown").innerHTML = html;
}

```

Server-Side Implementation

The server-side implementation relies on the *Composite* pattern (see Gamma et al., 1995). A *Category* consists of further *Category* objects and also of *Link* objects. We rely on *Category* and *Link* having two common operations:

asXMLTag()

Renders the item as an XML tag. For categories, there is a special optional parameter that determines whether or not the tag will include all items.

findItems(\$name)

Recursively finds an item—either a *Category* or a *Link*—having the specified name.

With these operations encapsulated in the *Category* and *Link* objects, the main script is quite small:

```

require_once("Link.php");
require_once("Category.php");
require_once("categoryData.php");

header("Content-type: text/xml");

$categoryName = $_GET['categoryName'];
$category = $topCategory->findItem($categoryName);
if ($category) {
    echo $category->asXMLTag(true);
} else {
    echo "No category called '$categoryName'";
}

```

Further refactoring: a Drilldown with dynamic content

In the refactoring above, the top-level category, and all of the data underneath it, is hard-coded in *categoryData.php*. In fact, the hierarchy data could easily be generated on demand to create a Drilldown with dynamic content. In a further refactoring (<http://ajaxify.com/run/portal/drilldown/syncLinks>), a *Cross-Domain Proxy* (Chapter 10) is introduced to grab the actual results from the *AjaxPatterns Wiki Links Page* (http://ajaxpatterns.org/Ajax_Links). It's not a true *Cross-Domain Proxy* because instead of grabbing the results in real-time, a process runs to pull them every sixty seconds and store them locally, where they can be picked up by the Drilldown script.

Alternatives

Live Search

Drilldown lets the user locate an item by browsing through a hierarchy. *Live Search* (see later) instead lets you locate an item by typing, and the data need not be hierarchical.

Tree

Like a Drilldown, a Tree widget lets the user navigate a hierarchy, just like the tree of files and folders in desktop file managers. Tree widgets, even if they expand and collapse, tend to take up more space than Drilldowns, but the more detailed view can be useful for longer, more complex, tasks.

Related Patterns

Microlink

Content blocks, produced when particular categories or items are selected can be associated with the Drilldown. Thus, the Drilldown contents are being used as *Microlinks* (Chapter 15).

Browser-Side Cache

If each navigation event leads to a query of the server, consider retaining results in a *Browser-Side Cache* (Chapter 13).

Portlet

A Drilldown is usually a form of *Portlet* (Chapter 15). It has its own state, and the user can usually conduct a conversation with the Drilldown in isolation.

Data Grid



Database, Query, Report, Summary, Table

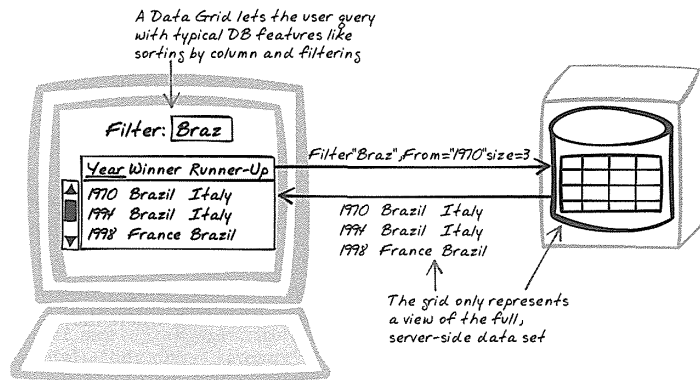


FIGURE 14-11. Data Grid

Goal Story

Reta is doing a little data mining on this season's sales figures. She's scrolling through a table showing a row for every transaction. Noticing that higher sales figures tend to come in the morning, she sorts the table by sales figure. There's too much data there, so she filters out any transactions below \$1,000, then groups the transactions by time of day.

Problem

How can you help users explore data?

Forces

- Ajax Apps have their own databases, as well as access to external systems.
- Users need a way to explore all this data, in order to verify it, understand how it works, predict future trends, and so on.
- You can't anticipate what users will need to do with data.

Solution

Report on some data in a rich table and support common querying functions. A Data Grid has the familiar database client interface: a table with a row for each result and a column for each property. Think of it as a traditional table on steroids. Typical database client functionality is as follows (and it's feasible to achieve each of these in an Ajax interface):

Sorting

Each column can be usually be sorted by clicking on the header. To let the user sort by more than one category, use a sorting algorithm that preserves the order of equal values. Then, users can click on one header followed by the other.

Filtering

The user can *filter in* to retain data matching certain criteria, or *filter out* to exclude that data.

Grouping

Data can be grouped by similarity. A large table is essentially broken into smaller tables, in which each item in a smaller table is similar to the other items in that table.

Aggregate calculations

Calculations can be performed across the whole table—for example, a sum or average for a column, or a sum of the products of two columns in each row.

Editing

Some Data Grids allow fields to be edited.

With the magic of Web Remoting (Chapter 6), the grid can become a *Virtual Workspace* (Chapter 15), giving the illusion of holding a massive set of data. In reality, the data lives on the server, and each *User Action* leads to a new server call. Most queries lead to a

structured response, such as an *XML Message* or a *JSON Message*, containing a list of results for JavaScript rendering, or alternatively an *HTML Message* with the specific view the user will see. The semantic style has the advantage of encouraging performance optimizations such as *Predictive Fetch*, which are essential if you want the grid to feel responsive.

Decisions

Will the Grid be Read-Only or Mutable?

Editing a table directly can be more productive for experts, though it's often more difficult for novices than editing a single record in a form, since a form usually has a more verbose interface. Grid mutability adds a couple of extra considerations:

- You need to validate the data. In a live context, this might mean showing a *Progress Indicator* during validation, then Highlighting invalid columns.
- Cells should ideally morph into input widgets when a user begins editing them. For instance, create a drop-down when the user begins to change a constrained field.

Real-World Examples

OpenRico Data Grid example

The OpenRico Data Grid example (<http://openrico.org/rico/livegrid.page>) shows a list of movies, each with several fields: ranking, title, genre, rating, votes, and year (Figure 14-12). You can scroll down the list and sort each column header. You can also bookmark a position in the table (an example of *Unique URLs*).

Listing movies 327 - 336 of 894

#	Title	Genre	Rating	Votes	Year
327	The Wind (1928/1)	Drama	8.0	538	1928
328	Requiem for a Dream	Drama	8.0	47559	2000
329	The Apartment	Drama	8.0	11165	1960
330	Se7en	Drama	8.0	88827	1995
331	To Kill a Mockingbird	Drama	8.0	30979	1962
332	Metropolis	Drama	8.0	11926	1927
333	It's a Wonderful Life	Family	8.0	42026	1946
334	The Wizard of Oz	Family	8.0	38263	1939
335	The Gold Rush	Family	8.0	6404	1925
336	Finding Nemo				

The OpenRico Data Grid downloads server content as and when user navigates to regions

FIGURE 14-12. OpenRico Grid Movie demo

NumSum

NumSum (<http://NumSum.com>) is a spreadsheet, and spreadsheets are a special, powerful case of Data Grids; they are to Data Grids what Data Grids are to conventional tables (Figure 14-13).

A spreadsheet is a kind of Data Grid

	A	B	C	D	E	F	G
1	Item Number	Link	Ended	Seller	Total Selling price	After Paypal Fees	Shipping Cost
2	8321713428	http://cgi.ebay.com	7/31/2005	Tonya	10.99	10.37	5
3	8321624081	http://cgi.ebay.com	7/31/2005	Jamie	33	31.41	7
4	8321623281	http://cgi.ebay.com	7/31/2005	Jamie	16.99	16.2	5
5	5411363110	http://cgi.ebay.com	7/31/2005	Jamie	14.65	13.93	5
6	8321624424	http://cgi.ebay.com	7/31/2005	Tonya	10.99	0	5
7	8319954577	http://cgi.ebay.com	7/24/05	Jamie	20.48	22.48	7
8	8321623642	http://cgi.ebay.com	7/31/2005	Jamie	15.5	14.75	5
9	8321623516	http://cgi.ebay.com	7/31/2005	Jamie	14.99	14.26	5

FIGURE 14-13. NumSum

Oddpost

Oddpost (<http://oddpost.com/learnmore>) is an Ajax mail client with a very similar look and feel to desktop clients like Outlook. The subject headers table is a form of Query-Report Table. As with the OpenRico example, you can scroll through the table and sort by column.

Delta Vacations

Delta Vacations (<http://www.deltavacations.com/destination.aspx?code=MONTGEBAY>) appears as a *Live Search* example, but it also serves as an example of filtering. Initially, all hotels in a destination are shown, and typing a search term retains only matching results.

Code Example: OpenRico Data Grid

The OpenRico Data Grid example (<http://openrico.org/rico/livegrid.page>) uses OpenRico's LiveGrid API. In the initial HTML, there are two tables: one for the column headers and one for the data. Note that the data table declares all the visible table rows, initialized with the values in the first few rows. This is an example of a *Virtual Workspace* (Chapter 15)—the rows will always stay fixed, but their data will change as scrolling occurs:

```
<table id="data_grid_header" class="fixedTable" cellspacing="0" cellpadding="0" style="width:560px">
  <tr>
    <th class="first tableCellHeader" style="width:30px;text-align:center">#</th>
    <th class="tableCellHeader" style="width:280px">Title</th>
    <th class="tableCellHeader" style="width:80px">Genre</th>
    <th class="tableCellHeader" style="width:50px">Rating</th>
    <th class="tableCellHeader" style="width:60px">Votes</th>
    <th class="tableCellHeader" style="width:60px">Year</th>
  </tr>
</table>

<table id="data_grid" class="fixedTable" cellspacing="0" cellpadding="0" style="width:560px; border-left:1px solid #ababab">
```

```

<tr>
  <td class="cell" style="width:30px;text-align:center">1</td>
  <td class="cell" style="width:280px"> Bend of the River</td>
  <td class="cell" style="width:80px">Western</td>
  <td class="cell" style="width:50px">7.3</td>
  <td class="cell" style="width:60px">664</td>
  <td class="cell" style="width:60px">1952</td>
</tr>
...
</table>

```

The grid is initialized on page load, with some configurable options passed in:

```

var opts = { prefetchBuffer: true, onscroll: updateHeader };
onloads.push( function() {
  tu = new Rico.LiveGrid(
    'data_grid', 10, 950, 'getMovieTableContent.do', opts );
});

```

The grid will then handle all user events. Notice the `getMovieTableContent.do` argument in its construction. That's the URL that will feed the grid with further data. It must be capable of accepting in certain parameters, such as initial position and number of rows to retain. For example, to load the page initially and scroll all the way to the end (rows 940 and on), the *XMLHttpRequest Call* (Chapter 6) goes to the following URL: `getMovieTableContent.do?id=data_grid&page_size=10&offset=940&_`. What's retrieved are 10 movies starting at 941 in an XHTML format, as shown in the following example. The grid component then updates itself with those rows.

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<ajax-response>
  <response type="object" id='data_grid_updater'>
    <rows update_ui='null' >

      <tr>
        <td>941</td>
        <td convert_spaces="true"> El Dorado</td>
        <td> <span style="font-weight:bold"> Western </span> </td>
        <td>7.4</td>
        <td>2421</td>
        <td>1966</td>
      </tr>
      ...
    </rows>
  </response>
</ajax-response>

```

Related Patterns

Virtual Workspace

As explained earlier in the “Solution,” Data Grids are usually *Virtual Workspaces* (Chapter 15).

Browser-Side Cache

To improve performance, retain data in a *Browser-Side Cache* (Chapter 13).

Predictive Fetch

To improve performance, consider *Predictive Fetches* (Chapter 13) that preload nearby results and aggregation functions.

Progress Indicator

Longer updates should be accompanied by a *Progress Indicator* (see earlier).

Metaphor

A Data Grid is the natural sequel to the traditional HTML table.

Acknowledgments

Christian Romney (<http://www.xml-blog.com/>) suggested the idea of a sort-and-filter pattern from which this pattern evolved, and also pointed out the Delta Vacations example.

Rich Text Editor



Editor, Formatting, Fonts, Rich, Text, Textarea, Toolbar, Write, WordProcessor, WYSIWYG

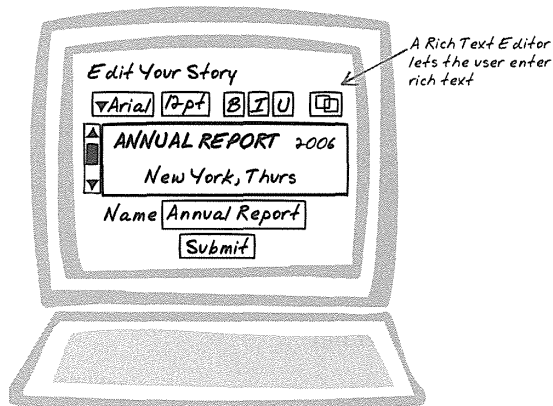


FIGURE 14-14. Rich Text Editor

Goal Story

Pam is working on a presentation for the steering committee; style will count here. From the toolbar, she sets the font to Arial, the size to 24 pt, and switches italics on. She then types out the heading, selects it, and moves to the toolbar again to set the color scheme.

Problem

How can users create and maintain rich content?

Forces

- Many Ajax Apps let users create and edit substantial chunks of content.
- Rich content for the Web needs to make the most of HTML and go well beyond a string of plain-text.
- Most users don't know HTML; even "easy" markup substitutes for HTML are complicated and inconsistent.

Solution

Incorporate a Rich Text Editor widget with easy formatting and WYSIWYG* display. Typically, the widget looks like a mini word processor: a toolbar on top with a rich editing area underneath. The editing area is usually a div rather than a textarea, meaning that any HTML content is possible.

Typical features include:

- Flexible font styles, sizes, boldfacing, etc.
- Flexible color schemes
- Embedded images
- Embedded tables
- Bullet-point and numeric lists
- Indenting and flexible text alignment

All of these features are usually accessible by the toolbar, as well as via keyboard shortcuts. It would also be possible to make them available from drop-down menus, though the main examples to date have avoided doing so. In addition, the toolbar sometimes offers other operations too:

- Undo, Redo
- Cut, Paste, Copy
- Save (*Explicit Submission* [Chapter 10])
- Spellcheck

Rich Text Editors are a great tool for nontechnical users, but as with GUI word processors, they can slow down power users. If power users are important to you, a few guidelines apply:

* "What You See Is What You Get" (WYSIWYG) interfaces are a staple of windows-based apps, where the editing interface is essentially the same as the output (i.e., a printout or a presentation).

- Offer an alternative “WYSIWYN” (What You See Is What You *Need*) interface, where the user can enter raw HTML and/or some other text-based markup such as Markdown (<http://daringfireball.net/projects/markdown/>).
- Offer keyboard shortcuts and advertise them well, e.g., as tooltip *Popups* (Chapter 15) on the corresponding toolbar icons.
- Offer personalized toolbars and keyboard bindings.

The major browsers do have some support for rich text editing. Firefox has Midas (http://kb.mozillazine.org/Firefox_-_Midas), an embedded text editor, and IE (<http://msdn.microsoft.com/workshop/author/dhtml/reference/properties/contenteditable.asp>) has a similar editor available on items flagged as `contentEditable`. However, neither mechanism is portable, and with both versions, you’re stuck with whatever version the user’s browser has. For that reason, the best solution right now is probably the *Cross-Browser Component* (Chapter 12) libraries mentioned in the following “Real-World Examples.”

Real-World Examples

FCKEditor library

Frederico Caldeira Knabben’s FCKEditor (<http://www.fckeditor.net/>) is a feature-rich, open source *Cross-Browser Component* you can pull into your own projects (Figure 14-15). It also has some server-side integration and web remoting support.

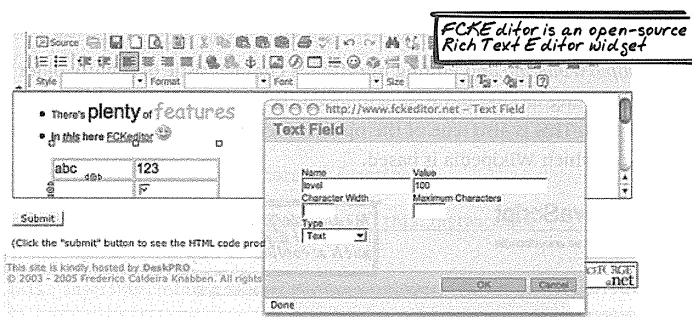


FIGURE 14-15. FCKEditor

Jotspot, Dojo Rich Text Editor

Jotspot is a wiki host that allows editing in three formats: WYSIWYG, markup, or XML (i.e., the underlying XHTML). The WYSIWYG supports all the Rich Text Editor capabilities mentioned in the previous Solution, and is based on the open source Dojo Rich Text Editor component (http://dojotoolkit.org/docs/rich_text.html). (See Figure 14-16.) In the simplest case, using the component is as simple as declaring a div with the right class.

```
<div class="dojo-Editor"> Initial
  content </div>
```

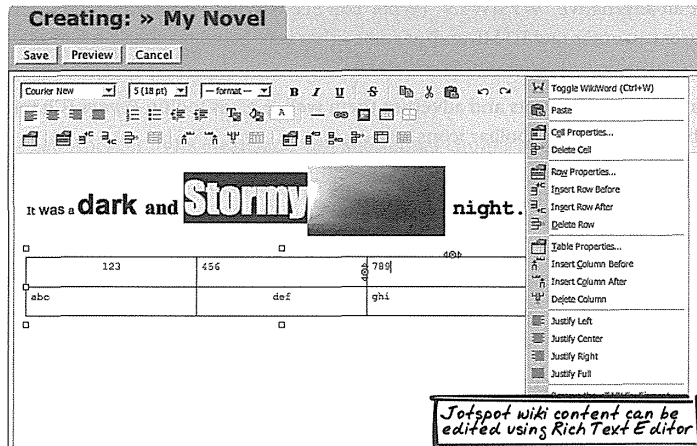


FIGURE 14-16. Jotspot Dojo Editor

Writely

Writely (<http://writely.com>) promotes itself as "The Web Word Processor." It edits content using a mechanism similar to that used by Jotspot. One nice feature is a spell-check.

Wikipedia

Wikipedia's (<http://Wikipedia.org>) editing interface is characteristic of slightly older editors that offer a rich editing toolbar but only a markup-based textarea mechanism for the content (Figure 14-17). This is also true of the open source MediaWiki framework (<http://mediawiki.org>) on which Wikipedia is based.

Editing JavaScript

From Wikipedia, the free encyclopedia.

This page is 31 kilobytes long. This may be longer than is preferable; see [article size](#).

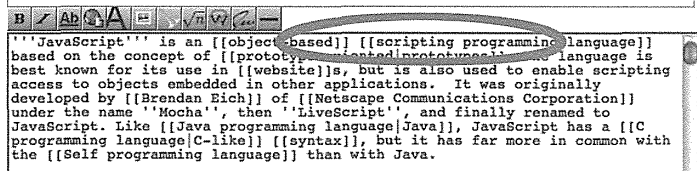


FIGURE 14-17. Wikipedia editor

Rich Text Editor

Kevin Roth's Rich Text Editor (<http://www.kevinroth.com/rte/demo.htm>) is a good demo of the native browser support mentioned in the preceding "Solution" because it provides an editor built on top of whichever browser it's running in.

Code Example: FCKEditor

This example introduces the FCKEditor API and looks at some of the internals. To use FCKEditor in your own project, first create a regular textarea:

```
<textarea id="comments"></textarea>
```

FCKEditor can then replace the text editor with a rich editor:

```
var commentsEditor = new FCKEditor("comments");
commentsEditor.ReplaceTextarea();
```

To track usage, register an initialization handler for the global FCKEditor module, which applies to all editor instances. Here you probably want to set up an event handler that will fire when a user changes an individual editor. See the API documentation (<http://wiki.fckeditor.net/>) for more details and options.

```
function FCKEditor_OnComplete(editor) {
    editor.Events.AttachEvent('OnSelectionChange', onEditorChange);
}
function onEditorChange(editor) {
    // Respond to change.
    // You can get the editor contents with editor.GetXHTML();
}
```

Now the internals. ReplaceTextarea will locate the textarea object, hide it, then insert the Rich Text Editor content just above it.

```
FCKEditor.prototype.ReplaceTextarea = function()
{
    ...
    var oTextarea = document.getElementById( this.InstanceName );
    oTextarea.style.display = 'none' ;
    ...
    this._InsertHtmlBefore( this._GetIFrameHtml(), oTextarea );
}
```

_GetIFrameHtml() outputs the HTML for the editor IFrame, and its source is fckeditor.html. The editor is structured as a three-row table. The top row is the toolbar (and some other controls for expanding and collapsing it); the second row is the WYWIWYG editing area, backed by an IFrame; the third row shows the HTML source, which is usually invisible.

```
<table height="100%" width="100%" cellpadding="0" cellspacing="0"
    border="0" style="TABLE-LAYOUT: fixed">
<tr>
    ...
    <td id="eToolbar" class="TB_ToolbarSet" unselectable="on"></td>
    ...
</tr>
<tr id="eWysiwyg">
    <td id="eWysiwygCell" height="100%" valign="top">
```



```

        <iframe id="eEditorArea" name="eEditorArea" height="100%" width="100%"
            frameborder="no" src="fckblank.html"></iframe>
    </td>
</tr>
<tr id="eSource" style="DISPLAY: none">
    <td class="Source" height="100%" valign="top">
        <textarea id="eSourceField" dir="ltr" style="WIDTH: 100%; HEIGHT: 100%">
    </td>
</tr>
</table>

```

The toolbar consists of toolbar buttons linked to commands. The commands follow the Command pattern (Gamma et al., 1995)—they are objects encapsulating a command and can be launched from the toolbar or with a keyboard shortcut. The command is identified in each toolbar button declaration as well as several display properties.

```

var FCKToolbarButton = function (commandName,
    label, tooltip, style, sourceView, contextSensitive)

```

For example, here's the toolbar button to paste plain-text:

```

oItem = new FCKToolbarButton('PasteText',
    FCKLang.PasteText, null, null, false, true);

```

The PasteText command in the preceding example is ultimately tied to a Command object that will launch a Paste Text dialog.

```

FCK.PasteAsPlainText=function() {
    FCKDialog.OpenDialog('FCKDialog_Paste',FCKLang.PasteAsText,
        'dialog/fck_paste.html',400,330,'PlainText'); }

```

Manipulations of the WYSIWYG editor content involve getting hold of the content element and simply altering its HTML. In the case of the Paste operation, it appends the new HTML, sHtml, at the right place.

```

var oEditor = window.parent.InnerDialogLoaded() ;
...
oEditor.FCK.InsertHtml(sHtml) ;

```

Related Patterns

Virtual Workspace

For editing large chunks of text, you might want to experiment with a *Virtual Workspace* (Chapter 15).

Progress Indicator

Provide a *Progress Indicator* (see earlier) while saving text. Indeed, many text editors take a while to start up as well, partly due to scripting overhead and partly due to toolbar images. Thus, a Progress Indicator might help during loading as well.

Status Area

Create a *Status Area* (Chapter 15) to help the user monitor details such as word count and cursor position, if these aren't already provided by the Rich Text Editor widget. You'll need to hook into the widget's event model to keep the status up-to-date.

Metaphor

A Rich Text Editor is like an embedded GUI word processor.

Suggestion



Auto-Complete, Fill, Intelligent, Populate, Predict, Suggest, Wizard

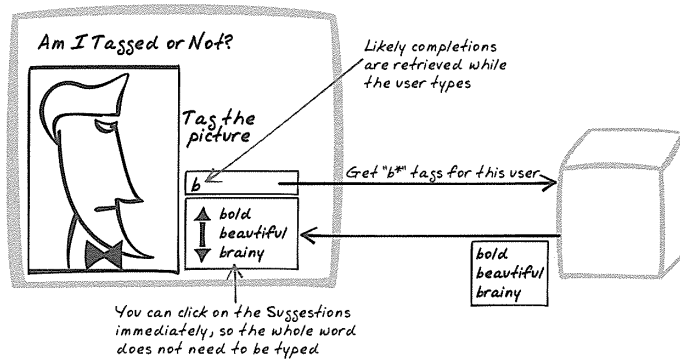


FIGURE 14-18. Suggestion

Goal Story

Doc is typing out an electronic prescription. He intends to prescribe "Vancomycin 65 mg," but as soon as he types in "V," the system has already detected a handful of likely choices, based on the patient's history and Doc's prescription style. Doc selects "Vancomycin 65 mg" from the list and proceeds to the next field.

Problem

How can you improve throughput?

Forces

- Free text remains the most powerful way for humans to communicate with computers. The trend seems to be toward more typing than ever before due to instant messaging, blogging, and email. Even search engines are undergoing a transformation to become general-purpose, as described in *Live Command-Line* later in this chapter.

- When presented with a free text area, people don't always know what they're meant to type in.
- Though many users are now quick on the keyboard, there are still many users for whom the Web is a click-mostly experience.
- Typing speed remains a bottleneck even for fast typists, most of whom think faster than they can type.
- People make mistakes when they type.

Solution

Suggest words or phrases that are likely to complete what the user is typing. The user can then select the Suggestion and avoid typing it in full. The results usually come from an *XMLHttpRequest Call* (Chapter 6)—the partial input is uploaded and the server responds with a collection of likely matches.

Suggestion has its roots in “combo boxes”—fields on traditional GUI forms that combine a text input with a drop-down list. The elements are kept synchronized. The user is free to type any text, and the current selection in the list will track what's been typed so far. The user is also free to choose an element from the list, which will then be posted back to the text field. In some cases, the list constrains what the user types; in other cases, it's there to provide Suggestions only.

It's the latter style that has become popular in recent years—free text entry, with some Suggestions for completion at any time. Auto-completion became popular with Internet Explorer 5, which auto-completed fields based on user history. Most users are comfortable with the approach, as all the modern browsers offer a similar feature, and the technique is also popular in mobile phone text messaging and East Asian text entry.

In an Ajaxian context, Google set the standard with its introduction of Google Suggest (<http://www.google.com/webhp?complete=1>), which suggests the most popular terms to complete the user's search query. Its release surprised many, as Google had managed to completely replicate conventional combo-box behavior, but this time, the terms were dynamically fetched from the server instead of being present when the form was created.

The mechanics of a Suggestion usually work like this:

- A standard input field is used. At the same time, an initially invisible div element is created to contain the Suggestions as they appear. The input field needs an event handler to monitor the text it contains so as to ensure that the list always highlights whichever Suggestion matches.
- Instead of requesting a Suggestion upon each keypress, *Submission Throttling* is usually adopted. Thus, every 100 milliseconds or so, the browser checks whether anything has changed since the last request. If so, the server is passed the partial query as a GET-based *XMLHttpRequest Call*.

- The server then uses some algorithm to produce an ordered list of Suggestions.
- Back in the browser, the callback function picks up the Suggestions and does some *Display Morphing* (Chapter 5) to show them to the user in a format that allows them to be selected. Each entry will have an event handler, so that if the entry is clicked, the input field will be altered.

A combo-box is not the only way to render results, but it has the dual virtues of efficiency and familiarity. “Code Example: Kayak,” later in this section, covers Kayak’s implementation.

You might think the main benefit of a Suggestion is to cut down on typing, but that’s not the case. If anything, it’s often slower to enter a single word using Suggestion, as it’s a distraction and probably requires some mousing. The main benefit is to offer a constrained set of choices—when there are more choices than would fit in a standard option list, but still a fixed set. For Google Suggest, it’s a list of choices that are considered probable based on search history. In tagging web sites, it’s a list of tag names that have been used in the past. In travel web sites, it’s a list of airports you can include on your ticket.

Decisions

What algorithm will be used to produce the Suggestions?

Ultimately, the Suggestion algorithm needs to find the most likely completions.

A few rules of thumb:

- In general, historical data is the best predictor. Log what users commonly search for and use that as a basis for Suggestions. So when the user types “A,” suggest the most frequent responses beginning with “A.”
- Personalize. Instead of completing “A” with the most common “A”-query *all* users have entered, return the most common “A”-queries for *this* user. The only problem here is lack of data, so consider a collaborative filtering algorithm to provide Suggestions based on similar users.
- Recent history is a more pertinent guide, whether you’re personalizing the results or not. In some cases, it makes sense to provide only recent queries. In other cases, consider weighting recent results more heavily, but taking into account older queries as well.
- In some cases, it might make more sense for the browser to provide *Guesstimates* (Chapter 13) rather than real results from the server in real-time. This information might be based on what’s in a *Browser-Side Cache* (Chapter 13) and, potentially, on a few algorithms to help the browser decide what’s relevant.

How will the Suggestions be ordered?

Typical ordering algorithms include:

- Estimated likelihood, e.g., historical frequency.
- How recently the query was last used.
- Alphabetical or numerical order.
- Application-specific ordering. A currency list often begins with the U.S. Dollar, for instance.

However you order your Suggestions, the first item is particularly important because it can be designed to enable the user to select the first element without explicitly choosing it from the list. For example, Google's combo-box implementation automatically inserts the first element in the text field, though with a text selection over the completion text so the user can easily override it.

How many Suggestions will be shown?

Deciding how many Suggestions to show is a balancing act. You want to show enough Suggestions to be useful, but not so many that the good ones get lost in the crowd. Space is limited as well, even if you use a pop-up list.

It's generally useful if your Suggestion algorithm not only ranks Suggestions, but also makes a relevance estimate. Then, you can ensure Suggestions are shown only if they are particularly useful. Sometimes it isn't really worth it to show any Suggestions.

What auxiliary information will be present with each Suggestion?

The core part of the Suggestion is always a word or phrase that completes what the user is typing. You can augment each Suggestion with supportive information. Google Suggest, for example, shows how many results each completion has. It would even be possible to present the Suggestions in a table, with several columns of background information per completion. Taken to an extreme, the Suggestion resembles a *Live Search* (see later in this chapter).

Real-World Examples

Kayak

Kayak (<http://kayak.com>) is a travel search engine that suggests airports as you type in a location (Figure 14-19). The Suggestions appear in a combo-box format similar to Google Suggest.

Google Suggest

Google Suggest (<http://www.google.com/webhp?complete=1>) was probably the first public usage of Suggestion and remains an excellent example. Results are shown in a combo-box style. The combo-box appearance is achieved with a custom control. A div is placed just

[Flights](#) | [Hotels](#) | [Cars](#) | [Deals](#) | [Buzz](#)

☒ Round-trip
 ☐ One-way
 ☐ Multi-city

From To

→ New York, NY USA - All airports (NYC)
 → New York, NY USA - LaGuardia (LGA) Airport
 → New York, NY USA - John F. Kennedy Intl (JFK) Airport
 New York, NY
 → New Orleans, LA USA - Louis Armstrong (MSY) Airport
 New Orleans, LA
 → New Bern, NC USA - Simmons Nott (EWN) Airport
 New Bern, NC
 → New Haven, CT USA - Tweed-New Haven (HVN) Airport
 New Haven, CT

hundreds of airlines from 100+ sites

Suggestions help complete an airport name

FIGURE 14-19. Kayak airport Suggestions

under the text input, with a Suggestion on each row, and the `zIndex` property is used to make the div appear “in front of” the usual page content.

Delicious, Amazon

Delicious (<http://del.icio.us>) and Amazon (<http://amazon.com>) both offer Suggestions to help complete tags (Figure 14-20). As you type a tag, several Suggestions appear immediately and are populated below the field. You can click on a Suggestion to complete the term.

Your tags: (What's this?)

software
 soft rock
 software development
 software programmer jobs
 software programmer
 social software
 soft
 soft pony
 software design
 sce software

Suggestions help complete a tag entry

FIGURE 14-20. Amazon tag Suggestions

Code Example: Kayak

Kayak (<http://kayak.com>) is a slick, Ajaxian interface with a handy Suggestion feature for source and destination airports. The list of Suggestions appears below the input field, is referred to in the code as a `smartBox`, and exists as long as the input field has focus.

Whenever a key is pressed, the `smartBox` is aborted—which stops any pending request—and a new request is initiated. A check is made to ensure that the text entry actually contains something, otherwise the `smartBox` is closed:

```
function _typer(input)
{
    if ((_input != null) && _input.value.length > 0) {abortSmartBox();_
runSearch(input);

}
    else {_setValue(-1);closeSmartBox();}
}
```

`_runSearch()` kicks off a call to the server to receive the Suggestion list. The query goes to <http://www.kayak.com/ml/smarty.jsp?where=LOCATION>. The result is some XML containing the airport names and codes and what appears to be an ID number, for example:

```
<i>28501</i><i>m>L</m><d>London, United Kingdom</d><a>LON</a>
```

The callback function accumulates an HTML string to be inserted into the smartbox. A loop runs over each result, appending a div element to the HTML:

```
var html = ""; var list = "";
for (var j = 0; j < ii.length; j++) {
    var id=client.getTagText(results[0], "i", j);
    var str=client.getTagText(results[0], "d", j);
    var match=client.getTagText(results[0], "m", j);
    ...
    list += _divB + _spanB + icon + str + _spanE + _divE;cnt++;
}
```

Also of interest is the optional icon, which appears beside the airport name, as shown in the preceding code. Depending on the resulting XML, the icon is chosen from one of two images:

```
var _iconAir = "<img style='vertical-align: middle' src='/images/airport_icon.gif'
border=
'0'>";
var _iconLoc = "<img style='vertical-align: middle; visibility: hidden' src=
'/images/place_icon.gif' border='0'>";
...
icon = (match == "L") ? _iconLoc : _iconAir;
```

An event handler monitors key presses. As the user moves up and down, the airport text is changed according to the selection, and the selection is incremented or decremented:

```
case UP:
    _cursel--;if (_cursel<0){_cursel=0;}selChoice(_cursel);
...
case DOWN:
    _cursel++;if (_cursel>_ids.length){_cursel=_ids.length-1;}selChoice(_cursel);
```

Finally, the value is set when the user clicks tab or enter. In the former case, focus will also proceed to the next field:

```

case ENTER:
    if (_ids.length>0){
        _setValue(_cursel);
        closeSmartBox();
    }
    ...
case TAB:
    if (_cursel>=0&&_cursel<_ids.length){_setValue(_cursel);}

```

Alternatives

Selector

In some cases, a plain old HTML selection input will suffice. Most modern browsers support navigation through the drop-down with a keyboard, so it's an easy, viable alternative if the range of inputs is constrained.

Related Patterns

Browser-Side Cache

A *Browser-Side Cache* (Chapter 13) is a practical way to speed up the search.

Guesstimate

Sometimes the browser may be smart enough to derive Suggestions without resorting to the server. This is an application of *Guesstimate* (Chapter 13).

Predictive Fetch

Suggestion is similar to *Predictive Fetch* (Chapter 13). Predictive Fetch queries the server in anticipation of a future user action, whereas Suggestion queries the server to help the user complete the current action. In theory, you could actually combine the two patterns—perform a Predictive Fetch to pull down some Suggestions that might be needed in the future. On an intranet, for instance, it might be practical to continuously make 26 parallel queries in anticipation of the user, who might type any letter of the alphabet.

Submission Throttling

To cut down on queries, don't issue an *XMLHttpRequest Call* upon every keypress. Instead, apply *Submission Throttling* (Chapter 10) to cap calls to a maximum frequency.

Highlight

When one of the Suggestions in the list matches the free text input, it's usually a good idea to *Highlight* (Chapter 16) it.

Lazy Registration

Lazy Registration (Chapter 17) involves the accumulation of profile information, and profiles can be used to generate personalized Suggestions. You can also retain user preferences regarding the appearance and timing of Suggestion lists.

Live Search

When full result details are provided, Suggestion resembles a *Live Search* (see the next pattern). However, the design goals are different and this should usually be reflected in the user interface. A Suggestion is intended to complete a word or phrase; a Live Search is intended to locate or report on an item being searched.

Want to Know More?

- “How Search Engines Rank Web Pages” by Danny Sullivan (<http://searchenginewatch.com/webmasters/article.php/2167961>)
- “Google Suggest Dissected” by Chris Justus (<http://serversideguy.blogspot.com/2004/12/google-suggest-dissected.html>)

Acknowledgments

Chris Justus’s thorough analysis of Google Suggest (<http://serversideguy.blogspot.com/2004/12/google-suggest-dissected.html>) was very helpful in explaining the magic behind Google’s Suggestion implementation.

Live Search



Feedback, Immediate, Live, Real-Time, Search

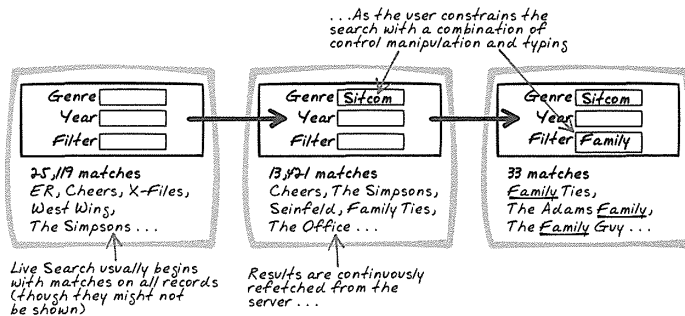


FIGURE 14-21. Live Search

Goal Story

Browsing a trade magazine, Reta has just spotted the “Curiously Costly” line of shoes and declared them a “must-have” for the upcoming season. She heads over to her favorite wholesaler’s web site and sees an empty search form with an assortment of products underneath it and the message, “50,000+ items.” First, she selects “shoes” from a drop-down and watches as the products below morph into a collection of shoes; the message is now “8,000+ shoes.” Then, she begins typing. As she types “C,” she now sees all 500 shoes beginning with “C.” This continues until the search field contains “Curio,” at which point only three items remain. One of those is “Curiously Costly,” which is what Reta clicks on to successfully conclude the search.

Problem

How can users quickly search for an item?

Forces

- Ajax Apps often require a search capability. Users search for business objects, other users, web sites, and more.
- Sometimes the user is searching for a needle in a very large haystack. In the case of search engines, a haystack of over a billion items.
- To help deal with the complexity, users should be free to experiment.

Solution

As the user constructs and refines his search query, continuously show all valid results. The search query is a combination of controls that lets the user narrow down the collection of items. It may be a straightforward text field, or a complex arrangement of *Sliders* (see earlier in this chapter), radiobuttons, and other controls. The results appear in a separate region and are continuously synchronized with the query that’s been specified. The search concludes when a result is chosen.

There are several benefits to using Live Search instead of the conventional style:

- Browsing and searching can be combined.
- Searching proceeds more quickly because no explicit submission is required. The results are ready almost immediately, so the user can experiment more easily.
- There’s no page reload, so the interaction feels smoother; results are updated, but the input query is completely unaffected.

The most common form of Live Search right now closely mirrors traditional searching. The user types into a free-text search field, and results are frequently updated as the user types. So, searching for “cataclysmic,” the user will see results for “c,” “ca,” and so on. (With *Submission Throttling* (Chapter 10) in place, the sequence might skip a bit, so a fast typist would see results “c,” “cata,” etc.)

Search and browse have often been considered mutually exclusive on the Web, which is unfortunate because both have their benefits. Search is good for finding something you already know about, while browse is good for exploring all of the items and stumbling across things you didn't know about. Live Search provides a way to combine the two. Some controls—such as selectors, radiobuttons, and *Sliders*—let the user choose an item from a small collection. If the search form can change dynamically (see *Live Form* later in this chapter), you can support browsing through a hierarchical structure.

Imagine how a web site search engine might achieve this. The initial control is just an empty text box and a selector with ten categories—Arts, Tech, News, and so on (Figure 14-22). The user chooses News on the selector and a new selector appears—World, Business, Science, etc. At any point, the user can change any selector. All the while, the results below are updating to show a sampling of results in the most specific category that's been specified. Furthermore, the text input is there all along, acting as an additional filter. So if you search at the top level, the interface degenerates into a standard live text search. But if you search while the selectors are set to a category, only the matching results from that category will be shown.

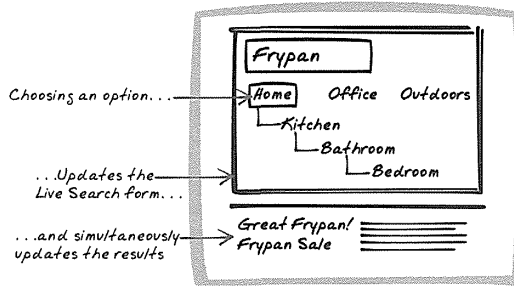


FIGURE 14-22. Live Search with Live Form

Live Search makes use of event handlers on the input controls, which listen for user activity and update the results. A typical sequence works like this:

1. The user changes a control; e.g., types a new character or clicks a radio button.
2. The entire input query—all of the input fields' settings—is transmitted to the server in an *XMLHttpRequest Call*.
3. The server decides which—if any—results should be shown and returns the list.
4. The browser replaces the list of results with this new list.

In the preceding scenario, the *XMLHttpRequest Call* need not occur in direct response to the user input. Often, *Submission Throttling* is used to delay the response.

A weaker form of Live Search doesn't show the actual results, but hints at them, e.g., indicating how many results and what type of results. The benefits to this are that there are

less processing, improved caching opportunities (because there's much less to store), and no space required for results while the search is occurring.

Live Search is not always desirable. Showing intermediate results, which happens when the search involves typing, can be distracting (<http://looksgoodworkswell.blogspot.com/2005/12/distracting-or-narrowing-looking.html>) and is not very useful anyway. If the user is searching for "cataclysmic," does she really care about the results for "cat"? An alternative might be to require an *Explicit Submission* (Chapter 10) but apply *Predictive Fetch* (Chapter 13) to keep pulling down results according to the user's current input. That way, the results will be ready as soon as the user clicks on the search button.

Decisions

How will you interpret incomplete text?

The user begins searching for "cataclysmic." Having typed as far as "cat," what will you show?

- Matches for "cat," "catalogue," "cataclysmic," and anything else *beginning* with "cat"?
- Matches for "tomcat," "delicate," scatter," and anything else *containing* cat.
- Just matches for "cat".
- A combination, with a few of each.

In the first two preceding options, "cat" is implicitly transformed to "cat*" and "*cat*", where "*" represents any character. For most searches, the "cat*" style is preferable because the user knows what she is looking for—a user searching for "cataclysmic" has no need to search for "*aclys*"—so inserting a wildcard at both ends merely clutters the interface and increases search cost.

Whether to use a wildcard at all is another matter. That is, should "cat" match just "cat," or also "cataclysmic"? There are benefits on both sides, which is why combining them is worth considering. It also comes down to how many results you generally expect. If there's only one or two results for "cat," then you might want to anticipate further typing. If there's a heap of results for "cat" alone, then there are probably enough to show that you don't need to anticipate any further characters.

How verbose will the results be?

As with regular search, you need to decide how much each candidate result will show. Because the results are sent frequently, you might need to limit results to just a basic summary. An interesting variation on this would be to use *Multi-Stage Download*—get a quick summary immediately, then a few seconds later, if the user is inactive, refine the results with a second query.

What will happen when there are too many results?

You can usually show only a fraction of the results at once—typically, up to 50 or 100 results and probably much less than that if bandwidth and server processing are critical constraints. However, there could be thousands of results. In the worst case, before the user has specified anything, there are no constraints and every item is a candidate. So you'll need to decide what happens when there are too many results to show.

One option is not to show any results at all, on the basis that the user should refine his search. This is not always the best idea, because you will have missed an opportunity to provide a little feedback and help the user explore. So you often want a way to provide a reduced list of matches. The search algorithm might have a way to prioritize results, perhaps based on popularity or the user's personal history.

You could also consider using a *Virtual Workspace* (Chapter 15) to give the appearance of having loaded all results and thus allow the user to navigate through them. The OpenRico Yahoo! Search Demo (<http://openrico.org/yahooSearch.page>) provides search results in this way. It's not a Live Search, but you could easily incorporate a Live Search into the approach.

A further possibility would be to place the results "on rotation"—that is, run a *Periodic Refresh* (Chapter 10) to show an ever-changing collection of results. I haven't seen this in an Ajax Live Search, but a slideshow-type navigation—rapidly rotating images—has been used in other domains to reduce information overload (for example, some TVs support channel surfing with a collage of rapidly changing channels).

Real-World Examples

Delta Vacations

Delta Vacations has a hotel booking facility. After identifying a destination on the destination page (<http://www.deltavacations.com/destinations.aspx>), you can perform a Live Search for a hotel. For example, if you type "dis" in the Los Angeles page (<http://www.deltavacations.com/destination.aspx?code=LOSANGELES>), you'll promptly see all of the Disney hotels appear (Figure 14-23). You can also change the sort order dynamically.

ListSomething.com

ListSomething.com offers Live Search for classified ads (Figure 14-24).

Amazon Diamond Search

Amazon's Diamond Search (<http://www.amazon.com/gp/gsl/search/finder/102-1689279-4344148?productGroupID=loose%5fdiamonds>) is a unique search interface, incorporating several *Sliders* (see earlier in this chapter) to set such variables as shape, price, and number of carats (see Figure 14-5). Each time you alter a Slider, a message appears beside it showing the total number of results for the current search. Unfortunately, this is a weak variant of Live Search in which the results themselves are not shown until an *Explicit Submission* (Chapter 10) takes place.

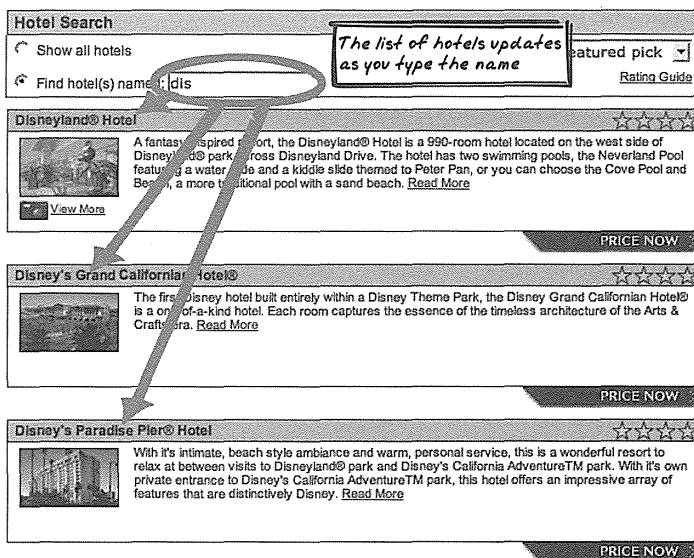


FIGURE 14-23. Delta

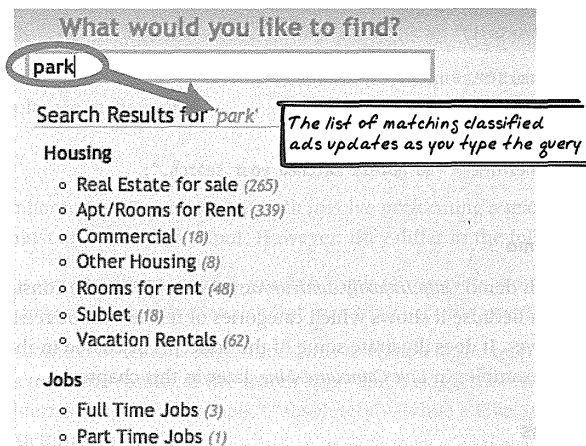


FIGURE 14-24. ListSomething.com

Skype

Skype's Skype-In service (<http://skype.com>) lets you choose a real-world phone number that will redirect to your Skype account. As you type some digits, you'll see Skype's available numbers.

Code Example: Delta Vacations

Delta Vacations

The Delta Vacations search registers an `updateHotelList` event handler on each control that manages the Live Search: the text input, a radiobutton pair determining whether to show all hotels or just those matching the text input, and a selector for the sort order. For example, here's the static HTML or the text input:

```
<input type="text" id="HotelNameBox" onFocus="setChecked('FindByName')"
onkeyup="updateHotelList('hdnDestCode', 'HotelNameBox', 'SortOptions');" />
```

`updateHotelList()` will then call the server. In this case, the server offers an *HTML Message* that outputs the exact HTML for the results. If you visit <http://www.deltavacations.com/ajax.aspx?d=LOSANGELES&n=dis&s=rating>, you can see that the result is just a segment of HTML:

```
<div id="H0" class="StandardHotel">
  <h2 class="FourStar">
    <a href="property.aspx?code=LAX0001&&dest=LOSANGELES">Disneyland Hotel</a>
  </h2>
  ...
```

The results are then morphed to show the HTML:

```
var container = document.getElementById('HotelListView');
if (null == container) return;
if (http.responseText)
{
  container.innerHTML = http.responseText;
}
else
{
  container.innerHTML = "No hotels matched your search.";
}
```

Assistive Search demo

The Assistive Search demo (<http://ajaxify.com/run/assistiveSearch>) can be considered a weak form of Live Search because it shows which categories of results will be retrieved but not the results themselves. It does illustrate some of the concepts discussed in this section and is analyzed in the examples in *Live Command-Line*, later in this chapter.

Related Patterns

Progress Indicator

While waiting for results to load after a user inputs data, you might choose to show a *Progress Indicator* (see earlier). Keep it fairly subtle, though, as it will appear and disappear frequently.

Data Grid

Live Search results can be shown in a *Data Grid* (see earlier).

Virtual Workspace

Early in the Live Search, there are often more results than you can show at once. Consider presenting them in a *Virtual Workspace* (Chapter 15) to let the user explore all the results.

One-Second Spotlight

One-Second Spotlight (Chapter 16) can be used to transition from older to newer results.

Submission Throttling

Instead of submitting upon each keypress, Live Searches often submit the input query at intervals—a form of *Submission Throttling* (Chapter 10). In many cases, it makes sense to submit only after an idle period so that the search term is exactly what the user is looking for rather than an incomplete version.

Unique URLs

Different search queries should generally have *Unique URLs* (Chapter 17) associated with them.

Predictive Fetch

You can look at Live Search as a kind of *Predictive Fetch* (Chapter 13) in which the prefetched results are made visible.

Suggestion

Suggestion is similar to Live Search in that both involve performing a search to synchronize the display against an incomplete input. However, they differ in the following ways:

- Suggestion is not just about search—you can use it to complete words as a user types. The backend algorithm involves search but searching isn't the user's goal.
- Even in the context of search, Suggestion aims to complete the query, whereas Live Search aims to fill the results. If the search says "ab," a Suggestion will give you potentially useful terms—e.g., "abdominals," "absolutely"—while a Live Search will give you all items matching "ab" or "ab...."

Live Command-Line



Assistive, CLI, Command-Line, Grammar, Supportive

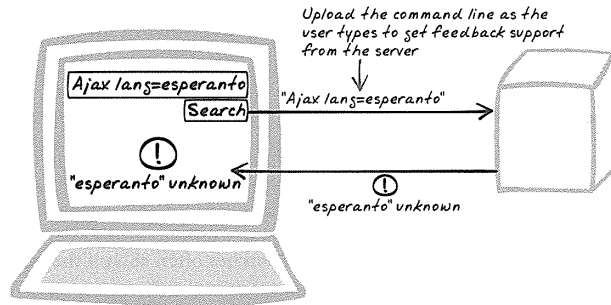


FIGURE 14-25. Live Command-Line

Goal Story

Frank is using an advanced query tool to run a report on current output. He begins entering an SQL-like query: "list all." At this point, a suggestion list appears with all the things that can be listed, and he chooses "products." Continuing on, the query now reads "list all products where" and a new list appears: "creationTime," "quality," and so on. He chooses "creationTime," and then enters the rest manually, preferring to ignore further suggestions. The query ends up as "list all products where creationTime is today and quality is faulty." There's a tick mark next to the query to indicate that it's valid.

Problem

How can a command-line interface be supported?

Forces

- Users can only say so much with point-and-click interfaces. The command line is more expressive and remains the tool of choice for expert users.
- URLs and search fields are becoming general-purpose command-lines (<http://www.codinghorror.com/blog/archives/000296.html>).

Solution

In command-line interfaces, monitor the command being composed and dynamically modify the interface to support the interaction. The interface changes according to data, business logic, and application logic. When the command line is to be executed on the server, as is often the case, the browser usually delegates to the server the task of evaluating the partial command.

The command line is the quintessential expert interface and it's worth bearing in mind that most users are "perpetual intermediates" (<http://www.codinghorror.com/blog/archives/000098.html>). That is, they learn enough to get by and generally learn new things only as and when required. So, while a powerful command line may be a superior choice in the long run, most users simply don't have the time to spend learning it in advance.

Traditional command lines present a problem because they offer the user a blank slate. Everything the user types must be based on prior, upfront, learning, and the only feedback occurs after the command line has been executed. That's not only inefficient for learning, but downright dangerous for sufficiently powerful commands. The model is: human enters command, computer executes command.

The solution is to be more interactive: with a smarter, more proactive command line, the construction of a command—its quick evolution from nothingness to an executable string—becomes something of a human/computer collaboration. For example, you can:

- Indicate whether the command is valid before submitting it. If it's not valid, prevent it from executing and inform the user what's wrong and how it can be rectified.
- Offer to fix invalid commands.
- Offer *Suggestions* to help complete part of the command.
- Help the user predict what will happen when the command is executed. For a query, this means hinting at the result—e.g., estimating number of matches (see *Guesstimate* [Chapter 13]).
- Provide graphical widgets to help the user construct the command. For example, build up a mathematical expression using a calculator-like interface to introduce numbers and functions. The query is still represented as plain-text, but the text doesn't need to be entered by hand.
- Provide graphical widgets to help the user refactor the command. For example, you could provide a "delete this" button next to each condition in a query.
- Provide visualization tools to render the query in other formats. For example, certain numerical queries might be visualized graphically.

The benefits apply to novices and experts alike. In supporting novices, the technique has sometimes been referred to as a "training wheels" approach. Like training wheels on a bike, you get a feel for the more powerful interface while still being productive.

Often, the command goes to the server, so it usually makes sense for the server to advise the use regarding partial commands. Basically, the procedure goes like this:

1. *Submission Throttling* (Chapter 10) periodically uploads the partial command string as an *XMLHttpRequest Call*.
2. The server then processes the command and returns supporting information.
3. The browser updates the user interface accordingly.

As an Ajax feature, this pattern is largely speculative: I'm not aware of any production-level Ajax Apps, but the command-line interface is growing at the same time as Ajax; it's inevitable that these things will collide, and as the discussion here suggests, there are plenty of synergies. One source of inspiration here is the evolution of IDEs over the past five years, particularly in the Java space. Environments like Eclipse (<http://eclipse.org>) and IntelliJ Idea (<http://www.jetbrains.com/idea/>) internally represent source files as data structures, leading to features such as ongoing error detection, offers to fix errors, suggestions, powerful refactorings, and alternative displays. All of this happens without the user ever having to compile the source code. As another source of inspiration, there was a recent experiment on training wheels for the command line (<http://www.cs.utep.edu/nigell/papers/HCI2005.pdf>). Students learned about the Unix command line from either a book or a GUI "training wheels" interface, similar to a Live Command-Line. The training wheels interface was found to be more productive and a more pleasant experience, too.

Decisions

Where will the partial command be analyzed—browser or server?

It's possible to perform some kinds of analysis in the browser—for example, the browser can check the query against a constant regular expression. So should you try to support analysis in the browser or delegate it to the server?

There are several advantages to analyzing the command server side:

- The analysis can take into account server-side data, external services, and server-side hardware resources.
- In most cases, the command will ultimately be processed by the server, so this allows related logic to be kept in the same place.
- Since the command is just a text string, it's in a convenient form for uploading and server-side processing. Also, it's easier to keep earlier results in a *Browser-Side Cache* because the command constitutes a key for the cache.

The downsides are:

- There is an extra load on the server.
- Extra bandwidth is required to download richer responses.
- There is extra lag time between the user's input typing and the browser's response.

How much information will the server provide?

Producing more information for partial commands will consume more server resources. The information will also take a bit longer to download. You'll need to trade off resources in favor of supportive information. For example:

- In supporting validation, you can often rule out many errors with a simple regular expression. But more complex errors relating to the existence of business objects, for

example, will get past that filter. Should you try to catch those too? Doing so would improve usability at the expense of resources.

- In hinting at the command's result, there's a spectrum of precision. For a query, you can say "there might be results," "there are results," or "there are 4123 results." In an extreme case, you can offer a *Live Search* (earlier in this chapter) and actually show the results.

Real-World Examples

I'm not aware of an all-encompassing Live Command-Line demo. However, the examples in *Live Search* and *Suggestion* (earlier in this chapter) are special cases of Live Command-Lines. There are also some JavaScript terminal emulators around, though they don't provide Live Command-Lines—see Web Shell (<http://a-i-studio.com/cmd/>) and JS/UNIX Unix Shell (http://www.masswerk.at/jsuix/jsuix_support/). Also, see the Try Ruby! tutorial (<http://tryruby.hobix.com/>) for an impressive web interface to the interactive Ruby programming environment.

YubNub

YubNub (<http://yubnub.org>) isn't Ajaxian, but it is noteworthy as the first explicit exploration of the "search as command line" meme. Yubnub lets users submit commands, which are then mapped into external queries. For example, if you type in "sum 10 20" (<http://yubnub.org/parser/parse?command=sum+10+20>), you get the sum of 10 and 20; if you type in "ajax form" (<http://yubnub.org/parser/parse?command=ajax+form>), you get search results from ajaxpatterns.org for "form."

Code Example: AjaxPatterns Assistive Search Demo

The Assistive Search Demo (<http://ajaxify.com/run/assistiveSearch>) was inspired by services such as Google Search, which are trending toward a general-purpose command line. It aims to illustrate the "training wheels" style of command-line support.

The Assistive Search Demo has a typical search engine form—a free-range text input with a Submit button—in addition to several images, one for each category that can be searched. Thus, the user is alerted to the capabilities of the search. As the user types, the categories highlight and unhighlight to help predict what kind of results will be returned (Figure 14-26). All of this information comes from the server side; the browser application doesn't know anything about the categories. It calls the server and startup to discover the categories and corresponding images. A timer monitors the command line, continually asking the server which categories it matches. The browser then highlights those categories.

The startup sequence loads the categories and kicks off the command-line monitoring loop. The server is required to have an image for each category. If it says that there's a "phone" category, then there will be a corresponding image at *Images/phone.gif*. The images are all downloaded and placed alongside each other:

```
window.onload = function() {
  initializeCategories();
  requestValidCategoriesLoop();
  ...
}
```

donut homer scandal

Search

Try: "donut homer scandal", "555-10*2"

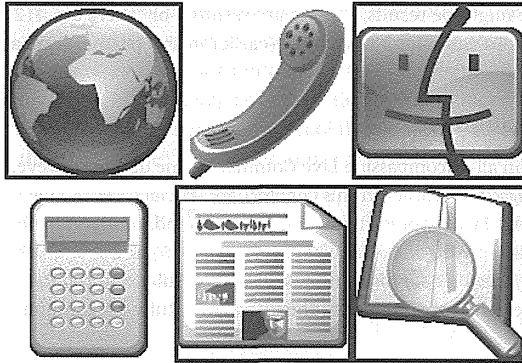


FIGURE 14-26. Assistive Search Demo

```
}  
  
function initializeCategories() {  
    ajaxCaller.getPlainText("categories.php?queryType=getAllCategoriesCSV",  
        onAllCategoriesResponse);  
}  
  
function onAllCategoriesResponse(text, callingContext) {  
    allCategoryNames = text.split(",");  
    var categoriesFragment = document.createDocumentFragment();  
    for (i=0; i<allCategoryNames.length; i++) {  
        var categoryName = allCategoryNames[i];  
        var categoryImage = document.createElement("img");  
        categoryImage.id = categoryName;  
        categoryImage.src = "Images/"+categoryName+".gif";  
        categoriesFragment.appendChild(categoryImage);  
    }  
    document.getElementById("categories").appendChild(categoriesFragment);  
}
```

The key to the Live Command-Line is the monitoring loop. Whenever a change occurs, it asks the server to return a list of all matching categories:

```
function requestValidCategoriesLoop() {  
    if (query() != latestServerQuery) {  
        var vars = {  
            queryType: "getValidCategories",  
            queryText: escape(query())  
        }  
        ajaxCaller.get("categories.php", vars, onValidCategoriesResponse,  
            false, null);  
        latestServerQuery = query();  
    }  
}
```

```

    }
    setTimeout('requestValidCategoriesLoop();', THROTTLE_PERIOD);
}

```

The server uses a bunch of heuristics to calculate the valid categories for this query. For instance, it decides if something belongs to the “people” category by looking up a collection of pronouns in the dictionary. A controller will ultimately return the categories as a comma-separated list generated with the following PHP logic:

```

function getValidCategories($queryText) {
    logInfo("Queried for '$queryText'\n");

    $cats = array();
    eregi("^.+", $queryText) && array_push($cats, 'web');
    eregi("[0-9\+][A-Z0-9 \-]*$", $queryText) && array_push($cats, 'phone');
    eregi("^[0-9\.\+\/\* \-]+$", $queryText) && array_push($cats, 'calculator');
    isInTheNews($queryText) && array_push($cats, 'news');
    isInWordList($queryText, "./pronouns") && array_push($cats, 'people');
    isInWordList($queryText, "./words") && array_push($cats, 'dictionary');

    return $cats;
}

```

The browser dynamically updates the category images using a CSS class indicating whether they are valid, which it determines by checking whether they were in the list. There’s also an event handler to perform searches on specific categories, as long as those categories are valid:

```

function onValidCategoriesResponse(text) {

    var validCategoryNames = text.split(",");

    // Create a data structure to make it faster to determine if a named
    // category is valid. For each valid category, we add an associative array
    // key into the array, with the key being the category name itself.
    validCategoryHash = new Array();
    for (i=0; i<validCategoryNames.length; i++) {
        validCategoryHash[validCategoryNames[i]] = "exists";
    }

    // For all categories, show the category if it's in the valid category map
    for (i=0; i<allCategoryNames.length; i++) {
        var categoryName = allCategoryNames[i];
        var categoryImage = $(categoryName);
        if (validCategoryHash[categoryName]) {
            categoryImage.onclick = onCategoryClicked;
            categoryImage.className="valid";
            categoryImage.title =
                "Category '" + categoryName + "'" + " probably has results";
        } else {
            categoryImage.onclick = null;
            categoryImage.className="invalid";
            categoryImage.title =
                "Category '" + categoryName + "'" + " has no results";
        }
    }
}

```

```
}  
}  
}
```

Alternatives

Point-and-click

The command line involves typing, while point-and-click is a simpler interface style that shows available options and lets the user click on one of them.

Drag-And-Drop

Another means of issuing commands is with *Drag-And-Drop* (Chapter 15)—dragging an item into a trash can to delete it.

Related Patterns

Submission Throttling

Instead of processing the command upon each keystroke, use *Submission Throttling* (Chapter 10) to analyze it at frequent intervals.

Status Area

Feedback such as input validation and result prediction is usually shown in a *Status Area* (Chapter 15).

Highlight

Some parts of the command can be highlighted to point out errors, incomplete text, and so on. Also, aspects of the *Status Area* can be highlighted to help with prediction.

Progress Indicator

If it takes more than one second to process the command line, consider showing a *Progress Indicator* (see earlier).

Browser-Side Cache

Introduce a *Browser-Side Cache* (Chapter 13) to retain the server's analysis of partial commands.

Fat Client

Consider creating a *Fat Client* (Chapter 13) that tries to handle as much of the analysis as possible, thus reducing server calls.

Live Search

Live Search (see earlier) is an extreme version of Live Command-Line applied to search, where the command is effectively executed while the user types.

Suggestion

Providing *Suggestions* (see earlier) is one characteristic of Live Command-Lines.

Want To Know More?

- Yubnub (<http://yubnub.org>) lets users search different sites based on a command line. Users are allowed to add new commands; is this “social command line sharing”?
- Russell Beattie: “Search as Practical Artificial Intelligence” (<http://www.russellbeattie.com/notebook/1008310.html>).
- Jeff Attwood: “Google-Fu: (T)he browser address bar is the new command line” (<http://www.codinghorror.com/blog/archives/000296.html>).

Live Form



Dynamic, Live, Real-Time, Validation

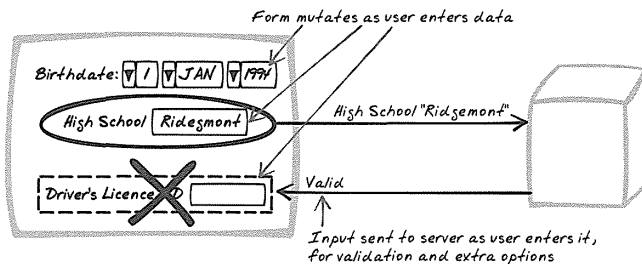


FIGURE 14-27. Live Form

Goal Story

Tracy is using a Live Form to apply for a new online brokerage account. Upon declaring her place of residence, some additional regulation-related questions specific to her region appear. Several asset classes can be traded. She clicks a box to indicate that she wants to trade in bonds, and further questions specifically for that asset class appear. She then clicks on another box, for options. A moment later, a warning appears that new users can trade in either options or bonds, but not both. After further refinement, she successfully submits the form and sees it fade into a welcome message.

Problem

How can the user submit data as quickly as possible?

Forces

- Most data submission tasks require some flexibility. A particular answer for one item may necessitate further questions. Often, this must occur server side, where business logic and data usually reside.
- Most data submission requires validation. This must occur server side to prevent browser-side manipulation of inputs.
- Users get frustrated waiting for data to be validated and continuously refining data upon each explicit submission.

Solution

Validate and modify a form throughout the entire interaction instead of waiting for an explicit submission. Each significant user event results in some browser-side processing, often leading to an *XMLHttpRequest Call*. The form may then be modified as a result.

Modifying the form is similar to responding to a *Microlink* (Chapter 15)—there’s usually a server call followed by some *Page Rearrangement* (Chapter 5). Typical modifications include the following:

- New controls appear.
- Controls become disabled.
- Information and error messages appear.

The result is a more interactive form. The user doesn’t have to wait for a page reload to find out if a form field is invalid, because it’s validated as soon as she enters the value. Due to the asynchronous nature of *XMLHttpRequest Call*, the validation occurs as a background process while the user continues entering data. If there’s an error, the user will find out about it while filling out a field further down on the form. As long as errors don’t happen too often, this is no great inconvenience.

Validation doesn’t always occur at field level though. Sometimes it’s the combination of data that’s a problem. This can be handled by checking validation rules only when the user has provided input for all related fields. A validation error can result in a general, form-level error message or a message next to one or more of the offending fields.

A Live Form usually ends with an *Explicit Submission* (Chapter 10), allowing the user to confirm that everything’s valid. Note that unlike a conventional form, the Explicit Submission does not perform a standard HTTP form submission but posts the contents as an *XMLHttpRequest Call*.

Real-World Examples

WPLicense

WPLicense (<http://yergler.net/projects/wplicense>) is a plugin for the WordPress blogging framework, allowing users to specify a copyright license for their blog content

(Figure 14-28). Refer to *Cross-Domain Proxy* (Chapter 10) for details on the plugin and to the section “Code Example: WPLicense,” later in this chapter, for a walkthrough of its *Live Search* implementation.

The screenshot shows a web form for selecting a license. At the top, there's a "Current License:" label with "(select)" and "(remove)" links. Below it, a box titled "Selected License: Attribution 2.5" contains a "License type:" dropdown menu set to "Creative Commons". A sub-form titled "The license terms sub-form changes according to the type of license chosen" is highlighted with a red box. This sub-form contains three questions: "Allow commercial uses of your work?" with a "Yes" dropdown, "Allows modifications of your work?" with a "Yes" dropdown, and "Jurisdiction of your license:" with a "Generic" dropdown.

FIGURE 14-28. WPLicense

Betfair

Betfair (<http://betfair.com>) lets the user make a bet using a Live Form—one which changes but doesn't actually interact with the server (Figure 14-29). You can try this even as a nonregistered user by opening up a bet from the *Drilldown* (see earlier in this chapter) menu. There are three columns on the page: the left column shows the current bet topic within the Drilldown, the center column shows odds for all candidates, and the right column shows your stake in the bet. Click on a candidate to make it appear in your stake form, where you can then set the amount for the bet you're placing. As you change your stake, a *Status Area* shows your total liability. Another column shows your profit with respect to each potential outcome. There are also some “meta” controls—e.g., the “Back All” button, which creates an editable row for every candidate.

MoveableType Comment hack

Simian Design (http://www.simiandesign.com/blog-fu/2005/07/ajax_comments_b.php#comments) is a blog where you can add comments, Ajax style. As you type into the text box, a live preview appears beneath it and takes markup into account. If you click on Preview or Post, the text fades and an animated *Progress Indicator* (see earlier in this chapter) bar shows that the server is processing. *Page Rearrangement* then embeds the result in the page.

Code Example: WPLicense

WPLicense (<http://yergler.net/projects/wplicense>) is a form with a live component. Initially, the “License Type” field simply shows the current license, if any. But when the user clicks on the update link, a new block opens up and a conversation between the browser and server ensues to establish the precise details of the license. (More precisely, the conversation is between the browser and the Creative Commons web site; see the corresponding example in *Cross-Domain Proxy* (Chapter 10). Each license type has associated questions, so the license options are fetched according to the user's license type selection.

NBA 05/06 - Outright Winner

Refresh

Place Bets My Bets Rules

View P&L (Settings) 7 112.1%

Total selections: (30) USD

	Back	Lay
San Antonio Spurs	2.94 \$55	2.96 \$118
Miami Heat	5.8 \$91	6.4 \$5
Detroit Pistons	4.5 \$5	4.6 \$118
Phoenix Suns	19.5 \$32	21 \$3
Indiana Pacers	9 \$71	15 \$91
Denver Nuggets	44 \$11	46 \$17
Houston Rockets	30 \$34	36 \$7
Dallas Mavericks	18 \$7	18.5 \$5
Cleveland Cavaliers	28 \$13	29 \$27
Chicago Bulls	110 \$11	120 \$10

Win Only Market

Clear All Back All Lay All

Back: To back a selection where odds are available, enter your stake and press submit. To ask for a new offer, enter your own odds. If you win your bet, your stake is returned to you together with your profit. If you lose, you lose your stake.

Back: You are backing Detroit Pistons

Your Odds: 4.3 Your Stake: 50 Your Payout: \$165.00

Lay: To lay a selection where odds are available, enter your stake and press submit. To ask for a new offer, enter your own odds. If you win your bet, your stake is returned to you together with your profit. If you lose, you lose your stake.

Lay: You are laying against Phoenix Suns

Backer's Odds: 19.5 Backer's Stake: 13 Backer's Payout: \$240.50

The form updates to show consequences of bet

In this case, you can decide what figure updates (Payout or Liability)

FIGURE 14-29. Betfair Live Form

When the user launches the plugin, a PHP function begins by drawing the Live Form. In this case, the form is ultimately submitted as a conventional POST upload, so the form has a conventional declaration:

```
<form name="license_options" method="post" action="" . $_SERVER[REQUEST_URI] . ">
...
<input type="submit" value="save" />
<input type="reset" value="cancel" id="cancel" />
...
</form>
```

The current license is shown and loaded with initial values along with the link that triggers the cross-domain mediation to update the license setting:

```
<tr><th>Current License:</th><td>
<a href="'.get_option('cc_content_license_uri').'">'.get_option('cc_content_
license').</a>
...

```

The form includes hidden input fields to capture information gleaned during the Ajax interaction. They will be populated by JavaScript event handlers:

```
<input name="license_name" type="hidden"
value="'.get_option('cc_content_license').'" />
```

Also, an initially invisible div will be shown once the user decides to change the license. Don't confuse this with the hidden input fields—those will never be shown to the user, whereas this just happens to have a hidden style when the page initially loads. Most critically, the div contains a selection field:

```
<select id="licenseClass"> <option id="-">(none)</option>
...

```

```

foreach($license_classes as $key => $l_id) {
    echo '<option value="' . $key . '" >' . $l_id . '</option>';
}; // for each...
...
</select>

```

Now for the live part. The `licenseClass` selector is associated with `retrieveQuestions`, a function that fetches the questions corresponding to the chosen license. The call is made courtesy of the Sack library (<http://twilightuniverse.com/2005/05/sack-of-ajax/>), which paints the server's HTML directly onto the `license_optionsdiv`. When you switch to a new license class, the form automatically shows the questions associated with that class:

```

el.onchange = function() {
    retrieveQuestions();
} // onchange

function retrieveQuestions() {
    ajax = new sack(blog_url);
    ajax.element='license_options';
    ajax.setVar('func', 'questions');
    ajax.setVar('class', cmbLC.value); 'Current license passed to server'
    ajax.runAJAX();
    ...
}

```

The server determines the appropriate questions to ask and outputs them as *HTML Messages*. (See the section “Code Example: WPLicense” in *Cross-Domain Proxy* [Chapter 10] for details.)

Now how about those hidden fields? Well, not all the information related to the options is hidden: the license URL is shown to the user, so that also needs to be updated after an option is chosen. Each time the user changes the server-generated options, the server is called to get a bunch of information about the current selection. That information then comes back to the browser, and the callback function updates both the hidden fields and the URL accordingly:

```

[Updating the hidden fields]

document.license_options.license_name.value = licenseInfo['name'];
document.license_options.license_uri.value = licenseInfo['uri'];
document.license_options.license_rdf.value = licenseInfo['rdf'];
document.license_options.license_html.value = licenseInfo['html'];

[Updating the visible license URL]

href_text = '<a href="' + licenseInfo['uri'] + '">' + licenseInfo['name'] + '</a>';
document.getElementById("newlicense_name").innerHTML = href_text;

```

Now, what, so far, changed on a permanent basis? Well, nothing. Everything so far has served the purpose of setting up the form itself. But that's fine in this case, because the form is submitted in the conventional manner. So as soon as the user clicks on the Submit button, all of those hidden fields will be uploaded and processed as standard HTTP form data.

Related Patterns

Microlink

Like Live Form, *Microlink* (Chapter 15) opens up new content directly on the page. Indeed, a Microlink can be used on a Live Form to open up supporting information.

Live Search

Consider including a *Live Search* (see earlier) when users need to perform a search within the Live Form.

Suggestion

Consider offering *Suggestions* (see earlier) to help users complete fields within the Live Form.

Drilldown

Consider including a *Drilldown* (see earlier) when users need to identify an item within a hierarchy.

Progress Indicator

Use a *Progress Indicator* (see earlier) to show that the server is working to process a field with a Progress Indicator.

One-Second Spotlight

Since Live Forms are usually submitted with an *XMLHttpRequest Call*, a *One-Second Spotlight* (Chapter 16) effect, such as Fade Out, is a good way to convey that the form has been submitted.

Page Architecture

THE “PAGE ARCHITECTURE” PATTERNS ARE A VARIETY OF INTERACTION STYLES AND STRATEGIES FOR structuring content. In many cases, it is the possibility of web remoting that makes these patterns worth using.

Drag-And-Drop has been a fixture on the desktop for a couple of decades but is only now gaining currency in the browser. One situation in which it’s used is with the *Sprite* pattern. A *Sprite* is a little icon that lives “in front of” the main document and can move around freely like a character in a video game. The next pattern, *Popup*, uses the same overlay technique, but for larger, dialog-style content.

Malleable Content is content that appears to be read-only but becomes editable upon a gesture such as a mouse rollover, a fairly new idiom because it relies on web remoting. Similarly, a *Microlink* is a link that uses remoting to conjure up content for insertion in the page.

A *Portlet* is a small component of the page that acts like a mini-application, capable of conducting a conversation with the server independent of the main page flow. A *Status Area* provides information about current and past activity and is relevant to Page Architecture because it helps reduce the space occupied by each individual element. *Update Control*

allows the user to control information flow on the page. The final pattern, *Virtual Workspace*, lets the user explore a large server-side structure, yet it makes the remoting transparent, creating the illusion that everything's happening in the browser.

Drag-And-Drop



Drag, Move, Pull, Rearrange

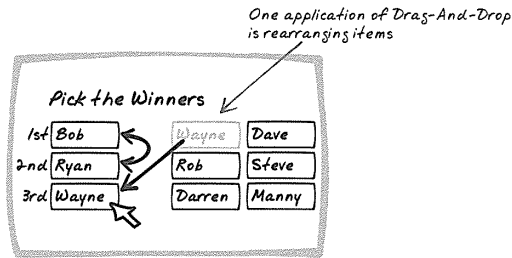


FIGURE 15-1. Drag-And-Drop

Goal Story

Stuart is purchasing books online. Each time he wants to buy one, he drags the book icon into a shopping cart.

Problem

How can users rearrange objects on the page?

Forces

- Object relationships are important and the user's task often involves changing those relationships.
- Ajax Apps often represent object relationships visually.
- Rearranging the visual structure from a separate form is messy—i.e., trying to map between the form and the visual structure is confusing and error-prone. It's easiest if users can directly manipulate the visual structure.

Solution

Provide a Drag-And-Drop mechanism to let users directly rearrange elements on the page. Drag-And-Drop has proven itself to be a powerful control mechanism in conventional desktop applications; it is certainly achievable using standard web technologies. The basics are straightforward: the user holds down the mouse button while the mouse hovers over a page element, then moves the mouse with the button still depressed. The element follows the mouse around until the user finally releases it.

Constraints are sometimes applied to the extent of movement. For example, an element may only be able to move in one direction, or within a bounding box. Also, the element might move permanently when the mouse is released, or flip back to its original position. Another variable is exactly which part of the element must be dragged. Sometimes it's better to define a "handle" region—a specified place where the element can be "picked up" for dragging. The main benefit to this is that it allows the user to click elsewhere in this region.

The following are among the applications:

Rearranging lists

This simple task can be ridiculously tedious on the Web. Some web sites require an entire page reload each time you move an object up or down. Others have implemented the buttons in JavaScript, but you still have to click Up or Down repeatedly just to place a single item. Drag-And-Drop is a natural fit—just pick up the list items and drop them where you want them.

Rearranging items in a geometric space

This is also a natural application: pick up the objects and drag them where you want them to go.

Operating on a geometric space

This might include dragging a magnifying glass or an eraser through an image.

Building up a collection

One example of collections building is dragging products into a shopping cart.

Expressing an action

Perhaps the most famous use of Drag-And-Drop was Apple's original deletion mechanism, where you delete something by dragging it into the trash can. The basic idea is to represent a command visually as an icon and let the user drag items into it for processing.

There are a few ways to implement Drag-And-Drop:

- Reuse an existing library. Libraries are becoming increasingly powerful and portable, and it's likely that those available will do the job for you.
- Leverage built-in Drag-And-Drop. Unfortunately, this isn't much of an option because only Windows IE supports Drag-And-Drop explicitly.
- Roll your own Drag-And-Drop. This is not recommended due to portability issues.

Here are a few Drag-And-Drop libraries, all of which have good cross-browser support and online demos:

Scriptaculous (<http://script.aculo.us>)

Provides, among other things, a general-purpose, portable Drag-And-Drop library

wzDragDrop (http://www.walterzorn.com/dragdrop/dragdrop_e.htm)

A Drag-And-Drop library that also includes resize capability

DOM-Dra (<http://www.youngpup.net/2001/domdrag/tutorial>)

A lightweight Drag-And-Drop library

Tim Taylor's Drag-And-Drop Sortable Lists (<http://tool-man.org/examples/sorting.html>)

A library designed specifically for list manipulation, supporting Drag-And-Drop-based reordering

The basic approach used for Drag-And-Drop is as follows:

1. Event handlers inspect the incoming event to determine which element is being dragged.
2. An `onmousedown` handler saves the starting coordinates, sets the `zIndex` so that the element appears in front during the drag, and changes some other style settings to indicate a drag has begun.
3. A `mousemove` handler inspects the mouse's coordinates and moves the element accordingly using positioning style properties (usually `left` and `top`). Here's where cross-browser support gets nasty—mouse coordinates in the event object are seriously platform-specific.
4. An `onmouseup` handler restores normal style settings and performs any other cleaning up.

Decisions

What constraints will apply to the drag operation?

You will need to decide directions in which the element can move and how far it can move. Generally, this should be fairly obvious from the visual representation being manipulated; often, there is a container in which similar objects live. This should be the bounding box for dragging operations.

Real-World Examples

Magnetic Poetry

Magnetic Poetry (<http://www.broken-notebook.com/magnetic>) is a fun application that simulates dragging magnetic tiles around a fridge.

Backbase Portal

Backbase Portal (<http://projects.backbase.com/RUI/portal.html>) shows various *Portlets* in a three-column structure. Users can easily rearrange the *Portlets* to suit their own taste using a Drag-And-Drop mechanism. Google's portal (<http://www.google.com/ig>) and Microsoft's Start.com (<http://www.start.com/3/>) follow a similar approach.

A9 Maps

A9 Maps (<http://maps.a9.com>) offers photographs of map locations. A draggable magnifying glass appears on the map, and you can move it to different regions of the map to see what the area looks like in real life.

Code Example: Magnetic Poetry

Magnetic Poetry uses the DOM-Drag library (<http://www.youngpup.net/2001/domdrag/tutorial>) to support dragging. On initialization, each tile is passed successively to `Drag.init`. The tiles may only be dragged within the board region, so the container's dimensions are passed in during the initialization sequence:

```
for(i=1; i<=numWords; i++){
    var currentTile = document.getElementById("word_" + i);
    var x1 = parseInt(document.getElementById("board").style.left);
    var x2 =  parseInt(document.getElementById("board").style.width)
        - parseInt(currentTile.style.width) - 6;
    var y1 = parseInt(document.getElementById("board").style.top);
    var y2 =  parseInt(document.getElementById("board").style.height)
        - parseInt(currentTile.style.height) - 6;
    // the last 4 args restrict the area that the tile can be dragged in
    Drag.init(currentTile, null, x1, x2, y1, y2);
}
```

The initialization is all you need to support dragging. With the preceding code, the tiles can now be happily moved around the board space. However, the application does a little more than that: it tracks the currently dragged tile, and it saves the position once dragging is finished using an *XMLHttpRequest Call*. To that end, `onDragStart` and `onDragEnd` handlers are registered on each of the tiles:

```
echo 'document.getElementById("word_" . $i . " ").onDragStart =
function(x, y) { dragStart("word_" . $i . " ", x, y); }';
echo 'document.getElementById("word_" . $i . " ").onDragEnd =
function(x, y) { dragEnd("word_" . $i . " ", x, y); }';
```

Alternatives

Separate Editing Interface

The conventional solution has been to provide a visual representation in one region and controls in another. This is often cumbersome and error-prone.

Related Patterns

Sprite

Drag-And-Drop is often the mechanism used to move *Sprites* (see the next pattern) around.

* The JavaScript is outputted from a PHP script.

Portlet

A portal can be personalized by letting the user drag *Portlets* (see later) around.

Slider

A *Slider* (Chapter 14) is a special case of Drag-And-Drop, where the value indicator is dragged along the slider axis.

Sprite



Bitmap, Buffer, Character, Layer, Overlay, Shape, Sprite, Z-Index

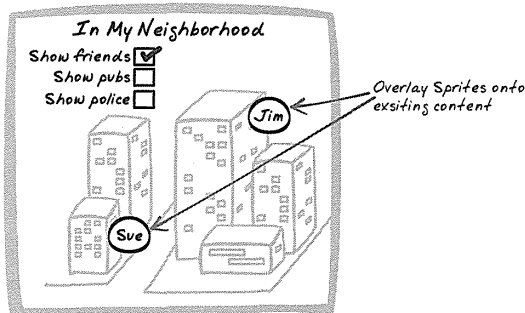


FIGURE 15-2. *Sprite*

Goal Story

Not for the first time, Pam is negotiating to eliminate a stack of requirements from the project goals. Having successfully convinced the client that “Security Features” was never really worthwhile, she drags a “Magic Eraser” Sprite from the Tool Palette to the “Security Features” requirement, jiggles the mouse a couple of times, and the eraser is animated for a second as the requirement fades away.

Problem

How can you ensure that visual content is flexible?

Forces

- Rich visual displays are often the most effective way to present information.
- Information frequently changes as the user interacts with and new data is received from the server; hence, the visual display needs to update frequently.
- It's expensive to continuously download new visual content.

Solution

Augment the display with Sprites—i.e., small, flexible, icon-like blocks of content. Typically, the Sprites are images, but they may also be div elements containing text or images or a combination of both. The name refers to the Sprites used in traditional graphics programming, particularly gamin—those little graphics representing players, enemies, and other objects capable of moving around a scene and animating themselves to reflect current activity. The entire scene does not need to be recomputed each time, only the Sprites. DOM elements are not (yet) capable of more advanced features such as arbitrary rotation, but many aspects of Sprites can actually be translated to a web context.

Google Maps is a case in point. If you search for “museum new york,” a set of thumbtack Sprites will appear on the map, one for each museum. Then, if you click on a thumbtack, a new speech balloon Sprite will show up with the precise address. Using Sprites, Google spares itself the burden of preparing images for every possible configuration, and from generating images on the fly. All it needs to do is ensure that there’s a basic display of the map in question, then overlay the thumbtacks and speech balloons.

Sprites are often implemented as div elements or simply as images. They usually appear “in front of” the rest of the document and are sometimes partially transparent. Following are the relevant CSS properties (also see *Page Rearrangement* [Chapter 5]):

zIndex

Determines how elements are stacked. Sprites usually have a high value in order to appear in front of other elements.

left, right, top, and bottom

Often used to position a Sprite. position can also be set to influence how the browser interprets these settings.

opacity

Often used to make the Sprite partially transparent. (IE requires the alpha filter workaround; see <http://www.domedia.org/oveklykken/css-transparency.php>.)

backgroundImage

Used to select a background image when the Sprite is a div. As long as there is no foreground content, the background image is shown. The image doesn’t need to be rectangular because a transparent GIF file can be used to show any arbitrary shape.

You can also animate the Sprite by rapidly changing its appearance. Unfortunately, JavaScript cannot easily control animated GIFs, so that’s not a viable option. However, animation is easy enough with some *Scheduling* (Chapter 7). One simple technique is to set up a recurring `changeImage()` action, in which the element’s `backgroundImage` is continuously run through a cycle of different images. Ideally, you should preload the images to avoid a slow cycle the first time round. Several tricks are available for preloading, the most common being set an invisible image’s source to point to the new image. An alternative animation technique is outlined in the section “Code Example: DHTML Lemmings,” later in this chapter.

Decisions

What file format will the Sprite image be?

Sprites are usually images, but what file format should you use—JPEG, GIF, or PNG? There are several considerations:

- Portability
- Image quality
- File size
- Transparency

JPEG may be the most common format on the Web, but it is ruled out for most Sprites. It doesn't support transparent pixels, so the Sprite must be rectangular, and it is not actually very effective at storing the kinds of images typically used for Sprites: small, cartoonish line drawings. So JPEG is only a consideration in the rare case that you have a fairly large, rectangular, photo-like Sprite.

This leaves GIF and PNG as the most common choices. An intellectual property cloud used to hang over GIF, but the patent expired in 2003 (<http://en.wikipedia.org/wiki/Gif>), so it's no longer a valid reason to avoid GIF. PNG does have some technical advantages over GIF, particularly variable transparency, but unfortunately that has not been correctly implemented in IE6 and earlier. So to support older IE editions, you'll need to manipulate opacity if you want to achieve transparency, regardless of the format. Also, even older versions of IE don't support PNG at all (<http://www.libpng.org/pub/png/pngapbr.html>).

In summary, GIF should be the default choice; only use JPEG or PNG if there is a specific reason for doing so.

Will the Sprite be animated?

Ongoing animation can be distracting, but a quick animation is an effective way to convey what's happening. Consider how the "one-second" visual effects might be used:

- *One-Second Spotlight* (Chapter 16): You might *Materialise* a Sprite when it first appears or *Fade In* the Sprite when it's subject to an action.
- *One-Second Mutation* (Chapter 16): You might *Form* a Sprite when it first appears, or rapidly *Metamorphise* it in a cycle in order to provide the illusion of continuous change.

Real-World Examples

DHTML Lemmings

DHTML Lemmings (<http://www.oldgames.dk/freeflashgames/arcadegames/playlemmings.php>) by Tino Zijdel ("Crisp") is a game that uses Sprites in their traditional gaming sense. Lemmings are shown performing different actions, and the characters change shape as they move around the game space. The entire thing is orchestrated with DOM and CSS scripting.

Google Maps

As discussed earlier in the Solution, Google Maps (<http://maps.google.com>) overlays the basic map with thumbtacks to highlight certain locations. As an interesting variant, look at how thumbtacks are used by Housing Maps (<http://housingmaps.com/>), the Maps-Craigslist mashup overviewed in *Cross-Domain Proxy* (Chapter 10).

Quek

Quek (<http://www.quek.nl>) is a fun use of Sprites as chat avatars (Figure 15-3). The application is a “chat-and-surf” program, meaning that Quek users can chat about a web site they’re visiting at the same time. Traditionally, such applications have involved *Richer Plugins* (Chapter 8), but Quek avoids any of that. It works by rendering the entire target web site, so you’re always actually on a Quek page, but the page looks the same as the target site. Each user that logs into Quek is assigned a unique Sprite consisting of an image, a text ID underneath the image, and an input box (visible only on the user’s own Sprite). Typing in the input box animates the Sprite image, and clicking Enter submits the message for all to see. Because the user is free to drag the Sprites around the page, the appearance is of several Sprites moving around and talking to each other, with the web page as a backdrop.

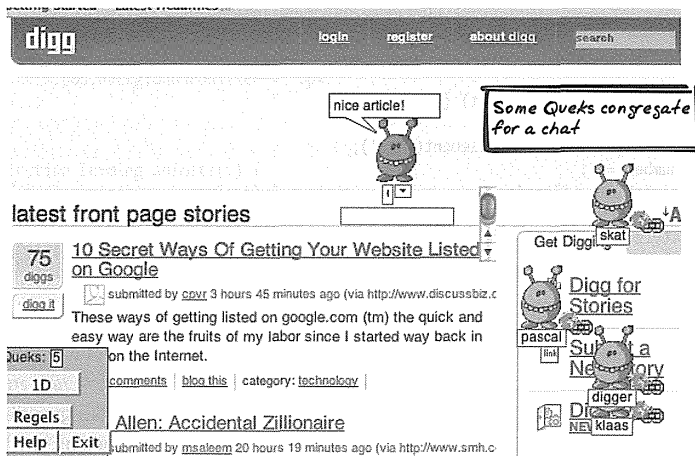


FIGURE 15-3. Quek

Code Example: DHTML Lemmings

DHTML Lemmings involves the most sophisticated use of Sprites I know of. The analysis here is only a rough summary of what’s achieved. Note that the code uses some special handling for IE, which the author, Tino, explained to me is due to a bug where IE (at least

some versions) bypasses the cache upon any background style change. To keep it simple, the following discussion skims over IE-specific handling.

In DHTML Lemmings, each lemming is represented by a `Lemming` object, which is initialized with a number to identify the lemming and an initial position corresponding to the place where the lemming first appears on this particular level. Various other state information is also tracked:

```
function Lemming(i,top,left) {
  ...
  this.top = top;
  this.left = left;
  ...
  this.number = i;
  ...
}
```

Lemmings themselves are regular JavaScript objects and are not visible in the browser. To visually depict the lemming, a `div` element is created and made an attribute of the lemming, simply known as `l`. Thus, we might say the `Lemming` model “wraps” a `Sprite` as a means of separating semantic content from visual content. Each of these `Sprites` resides in the playground, an area representing the game space where the lemmings roam around. Both the playground and the `Sprites` are rendered with absolute positioning. Once the `Sprite` element has been created, event handlers are also added to support the gameplay:

```
function Lemming(i,top,left) {
  ...
  var l = document.createElement('div');
  l.number = i;
  l.className = 'lemming';
  l.style.top = top+'px';
  l.style.left = left+'px';
  ...
  l.onmouseover = lemming_target_on;
  l.onmouseout = lemming_target_off;
  l.onmousedown = lemming_target_sel;
  ..
}
```

At any moment, each lemming is performing a single action, such as walking or climbing, and the action is tracked by a property of `Lemming`, `ani`. When the game begins, the lemming falls from a trapdoor, so `ani` is always initialized to `fall`:

```
function Lemming(i,top,left) {
  this.ani = 'fall';
  ...
}
```

Each type of action requires unique animation and movement. A periodic timer ticks every 60 milliseconds and updates the game’s state, including each lemming’s appearance

and positioning. Here's how the animation works. Each action is associated with a collection of 32×32 -pixel icons. The icons are strung together horizontally and retained in a single image. The lemming's `backgroundImage` style is always set to the entire image associated with its current action (Figure 15-4). Because the lemming image is set to 32 pixels, with overflow hidden, you will only ever see one of the icons. To select the appropriate icon, the `backgroundPosition` property is set. Thus, the `backgroundImage` remains fixed while the lemming is performing an action, but the `backgroundPosition` flows rapidly through each of the icons, beginning at zero and decrementing by 32 pixels each time until it reaches the full width of the image (as a negative value), at which point it becomes zero again.



FIGURE 15-4. DHTML Lemmings—background image used in floating sequence

Whenever the action changes, the script sets the image and initializes the position. Then, for each “tick” of the timer, the icon is advanced by altering the position of the reference image:

```

Lemming.prototype.changeAnimation = function(ani) {
  ...
  this.pos.backgroundImage = lem[ani][this.dir];
  this.pos.backgroundPosition = '0px';
  ...
}

function lemming_animate(i) {
  if (l.curleft == l.maxleft) l.curleft = 0;
  else l.curleft -= 32;
  if (ie) l.imgpos.left = l.curleft+'px';
  else l.pos.backgroundPosition = l.curleft+'px';
}

```

Why is the appearance changed using `backgroundPosition`? Wouldn't it be simpler to just run through a sequence of separate image files and alter the `backgroundImage` instead? Tino explains that the reason is performance: The image is held in memory, whereas image-swapping would require images to be retrieved from the cache.

As for motion, each “tick” delegates to a Strategy function (Gamma et al., 1995) for the action taking place. The Strategy function then directly manipulates the Sprite's position. So to model a lemming falling vertically, the div's top increases with each tick:

```

function lemming_fall_ani(l) {
  ...
  l.top += 4;
  ...
}

```


Likewise, a walking action involves making a change to the left property:

```
function lemming_walk_ani(l) {  
    l.left += l.dx;  
}
```

Alternatives

Tiling

Tiling is a technique in which a big image is built up from small tiles. Google Maps works this way; each map image is actually a grid of smaller image files. The benefit is that the browser will cache each tile in memory. So if you pan a little in one direction, only the new tiles need to be downloaded. Like Sprites, tiling is a technique you can use to change visual content without extracting everything from the server. And Google Maps shows that the two techniques combine effectively.

Related Patterns

Popup

Popup (see the next pattern) is a close cousin of *Sprite*. Both share the appearance of being “in front of” the rest of the document (this is typically implemented with the `zIndex` style), but they differ semantically. The *Sprite* pattern is intended to cover small objects, which are often draggable, animated, and icon-like. Often, Sprites are a standard fixture of the application, have a sense of unique identity, and remain present throughout the application. *Popup* is more about a transient, informational block of content that appears in order to show something or accept some input, then vanishes again. Popups tend to be larger and more text-based and are often dynamic in content.

Drag-And-Drop

Sprites can often be dragged around a space.

Metaphor

Think of a 20th century cartoonist creating a sequence in which a mouse walks onto a series of transparent “cel” sheets that will eventually be overlayed on a background scene.

Want to Know More?

- PNG File Format (<http://www.libpng.org/pub/png/pngintro.html>)

Acknowledgments

Thanks go to Tino “Crisp” Zijdel, author of DHTML Lemmings, for clarifying the analysis in this pattern.

Popup



Balloon, Dialog, Hover, Popup, Transparent, Tooltip

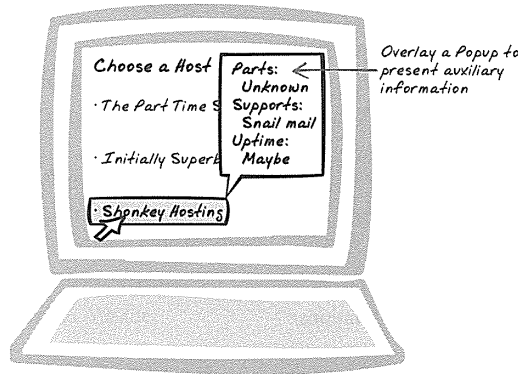


FIGURE 15-5. Popup

Goal Story

Frank is monitoring the factory equipment online and notices the T1000 machine is acting a little funny. He clicks on the icon, holds the mouse button down for a second, and up pops an information sheet about the T1000 model. The sheet is partially transparent and overlaid on top of a region of the plant diagram.

Problem

How can the user perform a quick task without distracting from the main flow?

Forces

- Users often need to make a quick change, which requires that they have some extra controls with which to interact.
- Users often need to perform a quick lookup, get further details about an object in the application, or just find some general information. This requires that extra information appear.
- Screen real estate is limited, and minor, quick changes don't warrant a major change to the user interface.

Solution

Support quick tasks and lookups with transient Popups, blocks of content that appear “in front of” the standard content. Usually, the content is partially transparent, is about the size of a standard dialog, and hovers above the page until the user

explicitly closes it. A more subtle variant is the tooltip, a solid block usually containing content between one word and a paragraph in length. The tooltip itself has a trendy new variant, a cartoonish balloon. For *Progress Indicators* (Chapter 14), a popular idiom is to use a small label in the corner with a message like “Saving....” Figure 15-6 illustrates several popular styles.

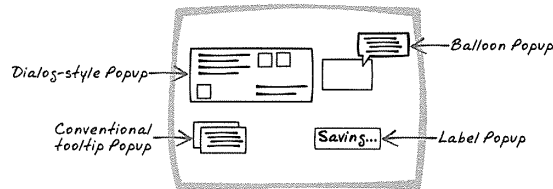


FIGURE 15-6. Popup styles

Popup is suitable for particular situations:

You expect the task to be quick

The Popup is limited in space and its transparent appearance makes it difficult to work with for a long time.

You expect the user to perform the task immediately

The rest of the document is partially blocked, making it difficult to work with while the Popup is around.

Often, a Popup has no permanent state; it becomes visible, and then is destroyed upon close. With many Popsups, there can be only one of its kind at any given time.

Popsups are usually divs. To ensure that only one Popup of a particular kind can appear, you can ensure that Popsups are always created at the time they are required and deleted on close. On opening, you just need to check whether the Popup already exists on the page. Alternatively, you can simply create the Popup on startup and toggle between showing and hiding it—show it when the user needs a Popup and hide it afterwards. The only catch here is that the state won’t be cleared, and often it’s more effective to treat each Popup as a new entity.

The following CSS styles are often useful for creating a Popup.

opacity (alpha filter in IE)

Used to produce a transparent feel.

zIndex

Used to place a Popup “in front of” the rest of the document. A solid (opacity=1) element will completely occlude everything that its *zIndex* is higher than.

visibility

Used to toggle whether the Popup is being shown.

Once the Popup is styled and is popping up at the right times, interaction proceeds as with any div element—the Popup just happens to look different than most content. Often, the interaction is similar to that of a *Portlet* (see later in this chapter)—i.e., each user action leads to an *XMLHttpRequest Call* (Chapter 6), with responses morphing the Popup.

Decisions

How will the Popup look?

The Popup's appearance is a compromise that allows you to present some different content without losing the original content. There's a tension between ease of Popup interaction and ease of comprehension and interaction with the rest of the page. Opacity and size will improve interaction with the Popup, but at the expense of the rest of the display. Different colors will also have an impact, and which color is the best depends on the content behind the Popup.

Here are some things to consider:

- Does the user need to see other page information while the Popup is present? If so, ensure that the Popup doesn't block it.
- How long will the Popup be present? If it will be a long time, consider making it less intrusive.

How will the user open and close the Popup?

There are two common approaches to opening the Popup:

- Clicking a button or an object related to the Popup content.
- Hovering over something for a while. Because this can happen quite often, it's more appropriate for a small, tooltip-style Popup.

There are several approaches to closing the Popup:

- Explicitly closing it with a button. Some Popups appear as desktop-like Popups, with an "X" on the top border of the window.
- Closing it when the interaction has come to a natural conclusion—e.g., a user submitting a form. In an Ajax App, the results of submitting it will soon be visible anyway, so there's still ample feedback.
- If the Popup was initiated with a hover event, closing it when the cursor hovers away from the underlying object.
- Closing it upon a timeout. This can be frustrating as it reduces user control. It also requires you to determine whether the user is in the middle of interacting with the Popup. If you use a timeout, consider providing a way to explicitly close it, too.

Real-World Examples

JSCalc

Cameron Adams' JSCalc (<http://www.themaninblue.com/experiment/JSCalc/>) provides a Popup calculator (Figure 15-7). You can use it while surfing on any site, because the calculator is actually a "bookmarklet"—a small JavaScript program which you add as a bookmark and then run by selecting it like a regular bookmark. The calculator appears as a grey, partially transparent display centered in front of the current page. You can type in equations, and when you eventually click on the rest of the page, the display fades away.

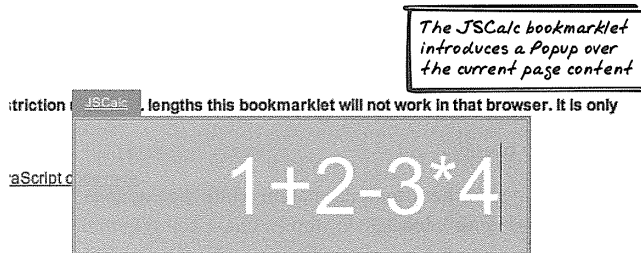


FIGURE 15-7. JSCalc

Flickr

Flickr (<http://flickr.com>) lets registered users add "notes" to anyone's photographs (Figure 15-8). A note initially appears as just a square outline over a region of the photo. When you hover over the note, its full text appears underneath. Creating a new note also uses a Popup effect to show a text field in which you can enter the note.

Netflix

The Netflix Top 100 (<http://www.netflix.com/Top100>) ranks films by all-time popularity (*Mystic River* is first right now, in case you're wondering). When you hover the mouse over a title, a speech balloon appears with a photo and summary. The effect is achieved with an *XMLHttpRequest* Call (Chapter 6). When you move the mouse away, the balloon disappears.

Hoversearch

HoverSearch (or "HoverSmack") (<http://www.forgetfoo.com/?blogid=3436>) is a Popup search. When you click on a hyperlink connected to an appropriate script, a transparent Popup appears and starts searching for a term. The results soon populate the Popup. The framework is capable of searching through different engines, including images, which are then shown in the Popup.

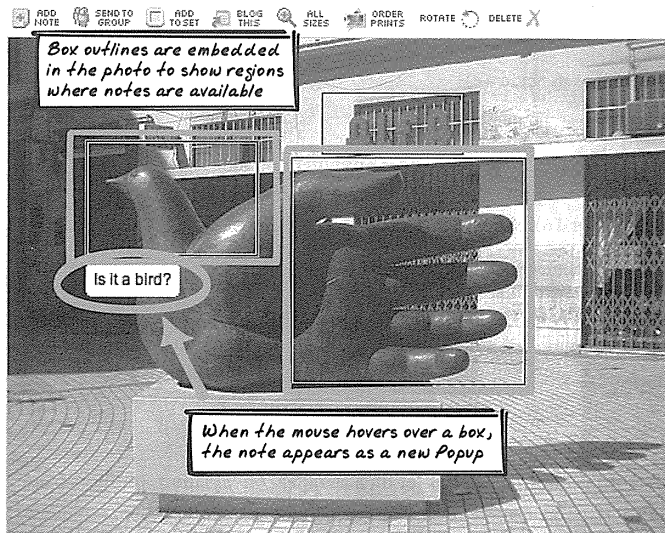


FIGURE 15-8. Flickr notes

Code Example: JSCalc

Being a bookmarklet, JSCalc is compressed into one line and has been reformatted in the following example. A div holds most of the calculator display and is appended to the page body. Since all pages have a body, the bookmarklet will work on any web page.

```
function JSC(){
  var d=document
  var b=d.getElementsByTagName('body')[0]
  var dv=d.createElement('div')
  ...
  b.appendChild(dv)
  ...
}
```

The calculator is positioned in the center of the document and has a slightly transparent opacity.

```
dvs.position='absolute'
dvs.top='50%'
dvs.left='50%'
dvs.width='300px'
dvs.height='60px'
...
dvs.opacity='0.95'
dvs.backgroundColor='#CCC'
```

Inside the div is a standard text input, also absolutely positioned on the page. It's given initial focus and registered with an onkeypress handler to track the calculation.

```

var inp=d.createElement('input')
...
inp.focus()
inp.onkeypress=function(e){
  ...
}

```

The Popup finishes when the user clicks anywhere outside of it. An `onblur` event is therefore registered. The Popup then fades out and is eventually destroyed in a *One-Second Spotlight*-style (Chapter 16) effect.

```

inp.onblur=function(e){ op(this.parentNode) }
function op(t){
  //(Fades out and removes the calculator elements)
}

```

Alternatives

Portlet

Portlet (see later) is another way to set up a parallel conversation. It differs from Popup in two ways: it's usually permanent or present for a long period of time, and it's usually not transparent. Because of their different characteristics, a Popup is more like a short detour, whereas a Portlet is like a completely parallel track.

Microlink

Microlink (see later) also opens up new content but mixes it directly on, as opposed to in front of, the existing content.

Related Patterns

Sprite

Sprite (see earlier) and Popup both appear “in front of” the rest of the document, but as the section “Related Patterns” in *Sprite* points out, their intent is different.

Drag-And-Drop

A *Drag-And-Drop* (see earlier) mechanism is a good thing for most Popups because it helps users reveal content that is underneath.

Slider

How transparent should the Popup be? The optimal value depends on the user's taste and the task he's working on. Thus, tie a Popup's opacity factor to a *Slider* (Chapter 14)—ranging from “invisible” to “solid”—for maximum flexibility.

Metaphor

Think of Terminator-style augmented reality, with a heads-up display helping you explore the world as you move through it.

Malleable Content



Affordance, Collaborate, Edit, Hint, Input, Microcontent, Mutable, Outline, P2P, Selection

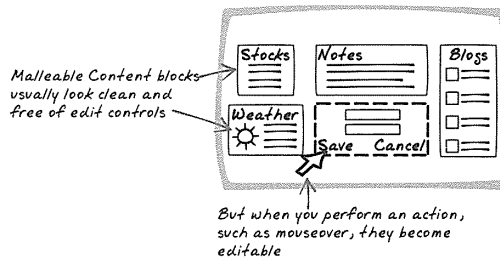


FIGURE 15-9. Malleable Content

Goal Story

Reta is logged in and is revamping the catalogue web site. It looks clean, a replica of the read-only page that public users will see. But as she runs the mouse down the screen, she sees that many regions can be edited.

Problem

How can you let users change content?

Forces

- The Web is not just about consuming information. Administrators, and increasingly everyday users, spend a lot of time adding and editing content.
- All users, no matter what their roles are, need to know what they're allowed to change.
- To encourage contributions and improve user experience, editing should be simple and free of barriers.
- Editable content pages are often ugly, with an overload of form controls, labels, and buttons. Worse, they create a divide between the contributors and the readers; while editing content, it's often difficult to imagine how a reader will see it.

Solution

Compose the page of Malleable Content blocks—small chunks of content that can be edited in page. The page is structured in chunks of small “microcontent,” each

editable in isolation. There might be a way to edit the whole page too, but the usual style of editing is to view the read-only display and carve up little pieces as you see fit.

The reload problem is one reason why conventional web sites must present large forms in which everything can be set at once. It's often the only practical way to edit content. However, users have difficulty relating the editing interface to the readable interface. Worse, it discourages editing; users are subjected to a sequence of at least two-page reloads in order to change a single character.

Ajax technologies lead to a smoother editing process. Display Manipulation (Chapter 5) and Dynamic Behavior (Chapter 7) help the user locate and open up editable Malleable Content. Display Manipulation also shows you how to conjure up a new editable area, or morph read-only Malleable Content into an editable form. Once the user has made the edit, you can seamlessly upload the response, as described in Web Remoting (Chapter 6), at which point the content can revert to read-only again.

In most cases, each block of Malleable Content is a div or textarea element. It can become editable in one of several ways:

- The display of the original element is adjusted and augmented with editing controls.
- A new div or textarea is created in its place.
- The original element becomes hidden and a hidden div or textarea is revealed.

Is that the sound of your inner usability critic banging his fists on the table and exclaiming “this defies everything that users have come to know about the Web”? Fair point. The forms are gone, so where's the affordance to let the user know what can be edited? Affordances can indeed be provided, as discussed in the following section.

Even if sufficient affordance can be provided, you might object that the whole thing is a complete departure from the usual forms-based approach. You've got me there. It certainly is a departure, but it's a departure from a ten-year-old approach that's remained frustrating since day one. Consistency is certainly an admirable goal—one that shouldn't be neglected because you have a few “one-percenter” improvements—but Ajax represents a new approach, and the benefits of in-page editing are just too big to ignore in the name of consistency. Better to adopt the approach and use sensible design to guide users through it. In any event, Malleable Content is likely to take off and will become familiar to users in the not-too-distant future.

Decisions

How will users locate Malleable Content and open it up for editing?

A read-only page will contain several editable Malleable Content blocks and various other regions that cannot be edited. Some of those regions are program-generated; others might be Malleable Content that this user is not permissioned to edit. So the user needs to know what can be edited. The static appearance should provide some cues, and events like mouse rollovers can refine that information.

The static appearance should make the location of the Malleable Content apparent. There are a few techniques to consider, some which are based on the Highlight pattern:

- When there are multiple Malleable Content blocks, increase spacing or alternate fonts between them to keep them distinct.
- Present the Malleable Content with a distinct background.
- Apply an explicit border around the Malleable Content.
- Expose the edit controls (e.g., Undo button), or the controls which initialize the edit (e.g., Edit button), but faded out.
- Add a label and auxiliary information beside each Malleable Content block. (See the following.)

These provide enough information for the user to detect that Malleable Content is present. However, too many of these cues will defeat the purpose of this pattern by making the whole thing hard to read. So you might use only a few subtle cues, or even none at all, and rely instead on dynamic behavior. Mouse movement is the most important activity to monitor; fields can be morphed to advertise their editability as the mouse rolls over them. Here, `onmouseover` sets up the cue and `onmouseout` takes it away. You can write a generic event handler and inspect the incoming event to see which piece of Malleable Content was altered. Then, apply (or unapply) the effect generically to that element.

All of the preceding static cues qualify as suitable dynamic cues, too. For example, change the background color as the mouse rolls over a particular block of Malleable Content, or make the label and edit controls visible only during that time. In addition, the cursor can also be varied using the `cursor` property. These dynamic cues, combined with just one or two static cues, are an effective way for users to pinpoint blocks of editable Malleable Content.

Will the Malleable Content be labelled? What auxiliary information will be included?

Many times, the Malleable Content has some form of identity, so you may wish to explicitly associate a name with it. The name might be shown above it as a heading or summary, on the side as an annotation, or as a tooltip.

The following auxiliary information, which can be automatically gathered by the server, might also be shown:

- The user who created the content
- The user who last edited the content
- The number of edits
- The creation date
- The last edit date

What content will be marked as Malleable Content? How big will the Malleable Content be?

Any editable information is a good candidate for inclusion as editable Malleable Content. One reason against making an item editable is that item's interaction with other data on the page. If there are certain combinations of data that are invalid, you may prefer to force the user to edit everything at once.

Malleable Content should be big enough to form a coherent unit of content, but small enough to be comfortably edited without scrolling and with enough room for some surrounding context. Of course, it's the users who will determine the exact size, and enforcing limits often achieves nothing but frustration. You can, however, influence the size of Malleable Content in several ways:

- Provide guidelines.
- Illustrate the guidelines and establish norms by restructuring existing content.
- Where the Malleable Content represents a data structure rather than just free text, ensure that the data structure is a suitable size. If it's too big, split it. If it's too small, aggregate it.

How will the user stop editing content?

Now you've decided how the user will open up content for editing, but how will she complete the edit? Here are the typical options:

- As soon as focus is blurred—e.g., when the user clicks on an external object—the content is saved and becomes read-only. This is good for power users who might wish to tab through content fields.
- Explicit controls, such as *Save* and *Cancel*, are added during editing. This may involve rearranging items on the page.

Is it okay to have several Malleable Content blocks open for editing at one time?

Because Malleable Content is an Ajax App, the user might not reload for a while, so he'll need to close each block manually in order to see the page become read-only again. You might enforce a rule that one field can be open at a time. This could be achieved by closing any existing block when a new one is opened, or by refusing to open a second block.

Real-World Examples

Flickr

When you manage one of your photos with Flickr (<http://flickr.com>), you'll see a title above the picture and a caption below it (Figure 15-10). Both are rendered in black-on-white

and appear as ordinary read-only text, so there's actually no static cue. But it's easy enough to discover it, especially given that most users are aware that these items can be edited. Hover over these fields and the background will switch to a light tan color. Then, click the button, and the field becomes editable, as shown in Figure 15-10.

Several things happen here:

- A border appears, outlining the field and suggesting that it's editable.
- The entire text is initially selected, an additional cue that you're now editing the field and a useful time saver as typing a new entry automatically deletes the existing text.
- Save and Cancel buttons are inserted onto the page, slightly displacing the content below. Clicking on Save or Cancel reverts the content back to read-only; otherwise, the field remains open and multiple fields can be open at once.

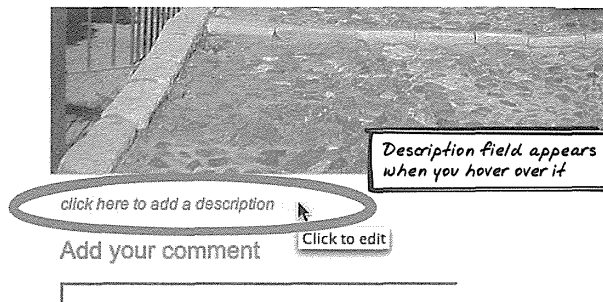


FIGURE 15-10. Flickr caption

Monket Calendar

The Monket Calendar (<http://www.monket.net/cal/>) lets you add events to a shared Ajax calendar (Figure 15-11). Each appointment is rendered as an orange-background rectangle with curves on either side. As the mouse hovers over a field, the field remains as is, but the cursor changes to a pointer style, suggesting that the field can be clicked. Once you click on the field, the rounded edges remain, but the central, horizontal component changes to white, the existing text is selected in full and becomes editable, and an "X" appears that lets you easily remove the appointment.

Tiddlywiki

With Tiddlywiki (<http://tiddlywiki.com>), controls appear as you roll the mouse over the Malleable Content (Figure 15-12). They are actually always present but are usually invisible, which means that surrounding fields don't move when you mouse around—the fields just toggle in their visibility status. One of these controls lets you edit: it switches the field into

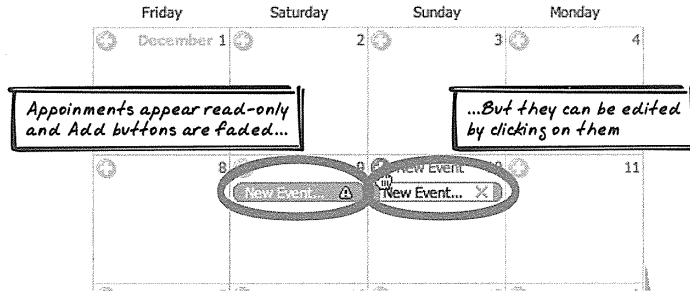


FIGURE 15-11. Monkei Calendar

5 December 2005

LarsEnglund has published a Swedish translation of TiddlyWiki experimental ServerSide using RDF.

no tags

Annotations: "Edit controls are usually invisible..."

30 November 2005

This is, again, a minor update to resolve persistent issues. The next major release will be 2.0.0, currently in beta at <http://www.tiddlywiki.com/beta>.

For the mathematicians out there, BobMcElrath has published L^AT_EX TiddlyWiki which uses jsMath to provide easy-to-write, high quality mathematical notation. Check it out at <http://bob.mcelrath.org/tiddlyjsmath-1.2.37.html>

There's also a new Basque translation of TiddlyWiki from SéamusPálás of the University of the Basque Country.

tags: news

close edit permalink references

Annotations: "...And only become visible when the mouse hovers over the corresponding Microcontent"

10 November 2005

Revision 1.2.38 is a minor revision of TiddlyWiki that just addresses an issue with Safari. The next major release, Revision 1.2.39, has just entered beta at <http://www.tiddlywiki.com/beta/>

FIGURE 15-12. Tiddlywiki

a textarea with an input field for the title. As you hover over the textarea, new controls appear, allowing you to save, cancel, or delete the item. It's possible to have more than one Malleable Content open at once.

Thomas Baekdal's Business Card Creator

Thomas Baekdal's Business Card Creator (<http://www.baekdal.com/articles/Usability/usable-XMLHttpRequest/>) contains a business card with mostly black content on a white background. As you hover over a field, the border changes, and when you edit, the background color also changes. This interaction style was the basis of the following wiki demo.

Code Example: AjaxPatterns Wiki

The Wiki Demo (<http://ajaxify.com/run/wiki>) breaks page content into a few Malleable Content blocks. A single div is declared in the initial HTML to hold these blocks:

```
<div id="messages"></div>
```

At periodic intervals, the incoming XML is parsed to extract each message into its own block of Malleable Content. Each message is actually a textarea but is rendered to appear read-only until it's edited. The CSS class, which initially corresponds to a grey background with no border, is set to show the text in read-only mode. Handlers are registered to provide dynamic cues that the Malleable Content can be edited and to handle the editing process:

```
function onMessagesLoaded(xml, callingContext) {  
  
    for (var i=0; i<wikiMessages.length; i++) {  
        ...  
        var messageArea = document.createElement("textarea");  
        messageArea.className = "messageArea";  
        messageArea.id = serverMessage.id;  
        messageArea.serverMessage = serverMessage;  
        messageArea.rows = 3;  
        messageArea.cols = 80;  
        messageArea.value = serverMessage.content;  
        messageArea.onmouseout = onMessageMouseOut;  
        messageArea.onmouseover = onMessageMouseOver;  
        messageArea.onfocus = onMessageFocus;  
        messageArea.onblur = onMessageBlur;  
        ...  
    }  
    ...  
}
```

We can examine each event handler to see how the editing cycle occurs. First, the user rolls over a message block. We'll outline the message with a dotted border to hint that it's in the twilight zone between read-only (no border) and editable (full border). We wouldn't want this to happen if the user was already editing the content, so we make an explicit check for that condition. Incidentally, you'll notice that the event handlers also manipulate colors here. That shouldn't be necessary; it is a workaround for the fact that some browsers unfortunately resize the border when it changes style. Without the workaround, the other content blocks would undergo slight movement each time one block changes.

```

function onMouseOver(event) {
    var message = getMessage(event);
    if (!message.hasFocus) {
        message.style.borderStyle="dotted";
        message.style.borderColor="black";
    }
}

```

And when the user rolls off this message block, the appearance is reset:

```

function onMouseOut(event) {
    var message = getMessage(event);
    if (!message.hasFocus) {
        message.style.borderStyle="solid";
        message.style.borderColor="white";
    }
}

```

Okay, our user has just rolled over a message block and now clicks on it to begin the edit. In the previous handler, we performed a check based on a `hasFocus` field to see whether the message was being edited. That's not built into the DOM, so we have to set it manually when the edit begins. During editing, the background is white and the border solid black. Finally, we ensure that no synchronization occurs while the user is editing (which is not always advisable):

```

function onFocus(event) {
    var message = getMessage(event);
    message.hasFocus=true;
    message.style.borderStyle="solid";
    message.style.borderColor="black";
    message.style.backgroundColor="white";
    stopPeriodicSync();
}

```

The user signals end of edit by clicking elsewhere on the page, causing a blur event to occur. The message returns to its original read-only form. Or, if a change has occurred, the message is highlighted and will eventually be reset once the change has reached the server. Also, a resize function adjusts the height according to the new content. Finally, synchronization can begin again:

```

function onBlur(event) {
    var message = getMessage(event);
    message.hasFocus=false;
    message.style.borderStyle="solid";
    message.style.borderColor="white";
    message.style.backgroundColor="#cccccc";
    resize(message);

    var initialMessageContent = message.serverMessage.content;

    if (message.value != initialMessageContent) {
        pendingMessages[message.id] = true;
    }
}

```

```

        message.style.backgroundColor = "#ffcc33";
    }

    startPeriodicSync();
}

```

Alternatives

Compound Edit

It's possible to edit the whole page content at once—the typical approach in conventional web apps. As explained earlier in this section, there are problems with the approach, but users may prefer it for long editing sessions. Ideally, both styles should be provided.

Related Patterns

Live Form

Upon becoming editable, a Malleable Content block might become a *Live Form* (Chapter 14).

Rich Text Editor

One way to edit Malleable Content is with a *Rich Text Editor* (Chapter 14).

Microlink

Malleable Content can be summoned onto the page using a *Microlink* (see the next pattern)—the approach used by TiddlyWiki (<http://tiddlywiki.com>).

One-Second Spotlight, One-Second Motion, and One-Second Mutation

The “One-Second” visual effects (Chapter 16) are a good way to support editing. It's becoming common to Fade In recently saved content as a means of hinting that it has been acknowledged. Mutations such as Blow Up and Disappear imply that Malleable Content is being deleted.

Highlight

Malleable Content is often highlighted to show that it's being edited and as a dynamic cue that it's available for editing.

Metaphor

Applications using Malleable Content are like a communal whiteboard.

Microlink



Appear, Conjure, Emerge, Hyperlink, Insert, Introduce, In-Page

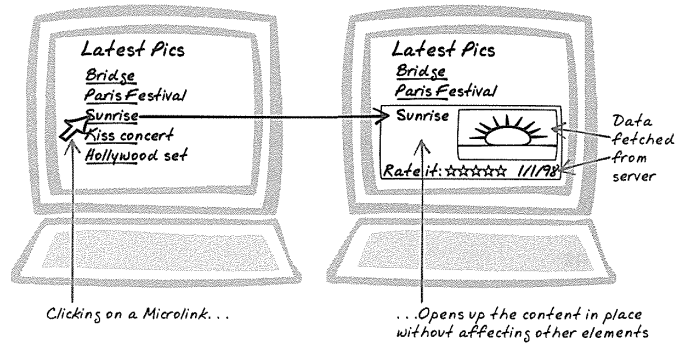


FIGURE 15-13. Microlink

Goal Story

Frustrated with a developer's productivity, Pam logs into the version control system and sees a "Project Committers" table. Each row includes a developer's name and photo, and Pam immediately clicks on one of the names. The row expands out to reveal a mini-summary of the developer's project activity, and a bunch of links to further content.

Problem

How can the user quickly navigate through content?

Forces

- Ajax Apps often respond to user requests by providing new content.
- To encourage interaction with the system, content should be shown quickly and without distraction.
- Page reloads slow down interaction, break concentration, and obscure the nature of whatever changes have occurred.

Solution

Provide Microlinks that open up new content on the existing page rather than loading a new page. Microlink is an Ajaxian upgrade of the traditional hyperlink. Most often, it entails fetching content from the server with an *XMLHttpRequest Call* and inserting it onto the page. The content is usually a tightly scoped block of Microcontent, though it need not be—a Microlink could be used to open a whole page of content, for example.

The term Microlink should not be taken here as a literal translation of textual hyperlinks. It's used to represent any form of content being inserted into the page. The trigger might be something as subtle as a form field focus event.

Microlinks can be used in many contexts:

- They can open a *Popup* (earlier in this chapter) to augment a particular word or phrase. Some conventional web sites provide glossary lookups in a pop-up window. A Microlink could instead be used to place a small explanation directly next to the term.
- They can be used to drill down by expanding out successively deeper levels of content. You could show all teams in a football league, each expanding to a list of players, in turn expanding to a list of games, and so on.
- They can let the user switch content around on the page, which is useful if the Microlinks are included in a *Drilldown* (Chapter 14).

The name “Microlink” refers to the subtle effect it has on interaction flow. Microlink carries several benefits over the full-blown page refresh caused by a standard hyperlink:

Content appears faster

There is much less to transfer because only the new content is loaded rather than the whole page. Also, the browser needs only to make a few changes to the DOM as opposed to rerendering the whole page.

Application flow is smoother

To the user, a page reload is distracting. Microlink avoids this distraction.

Changes are more salient

The user is able to watch as an element appears on the page. Visual effects can be used to further improve change detection.

Internal state is maintained

The browser-side application's state—as held in the DOM—is retained, whereas a page reload destroys it. This is a technical benefit.

Microlink usually involves an *XMLHttpRequest Call* (Chapter 6) followed by a *Page Rearrangement* (Chapter 5). Often, a new div is created and appended to a container element. As a variant, sometimes the Microlink refers to something already on the page, so the Microlink causes the script to make it visible or to scroll toward it (using `document.scrollTo`).

Decisions

What will happen to other content?

A Microlink inserts new content into the existing page, and you obviously can't keep piling on content forever. Furthermore, you need to remove the clutter so that new content is more salient. At some point, you'll have to clean up existing content, and you'll need to decide how that should happen. Typical strategies include those described in the list that follows.

- The new content simply replaces the old content. The old content is overwritten, or the new content is added and the old content replaced.
- A fixed-capacity content queue is set up, such that newer content displaces older content, which eventually disappears.
- The user must explicitly close all content.

How will Microlink be presented visually?

Microlink presentation is tricky because you want to leverage existing knowledge about hyperlinks, but do it without causing confusion. Many interfaces will combine Microlinks and hyperlinks (e.g., links to company homepages or external web sites), and you don't want a situation where the user can't distinguish between them. Also keep in mind that Microlinks don't have to be words or phrases—they can, for example, be buttons or `imagemaps`.

You need to ensure that users know Microlinks are clickable. You also need to have some ability to predict what they'll see when that happens. To provide affordances that an element is clickable, consider doing the following:

- Change the cursor icon as the user hovers over the element; control this with the `cursor style` property.
- Leveraging the conventions of hyperlinks, namely blue text (`color style`) and underlining (`text-decoration style`), but be beware that this may cause confusion as well.
- Make the clickable region apparent visually using cues like borders and different background colors.

Real-World Examples

TiddlyWiki

TiddlyWiki (<http://tiddlywiki.com/>) is a wiki based on *Malleable Content* blocks. At any point in time, you're looking at a list of Malleable Content blocks. Each block contains text including hyperlinks to other Malleable Content as well as to external sites. The external links are distinguished using a bold font. When you click on a Microlink and the corresponding context doesn't yet exist, the context is inserted just below the block containing the Microlink. If it does exist, the window scrolls to show the content. For more on this, see the section "Real-World Examples" in *Malleable Content* earlier in this chapter, and the section *One-Second Mutation* (Chapter 16).

Tabtastic library

A great application for Microlinks is the tabbed content metaphor, in which different blocks of content are shown as tabs. The technique was popularized by Amazon in the late 1990s, and Ajax allows for the content in another tab to be downloaded without a page refresh. Tabtastic (<http://phrogz.net/JS/Tabtastic/index.html>) is a JavaScript toolkit you can

use to incorporate tabs into your application—Microlink style. The web site demonstrates how it's done (Figure 15-14).

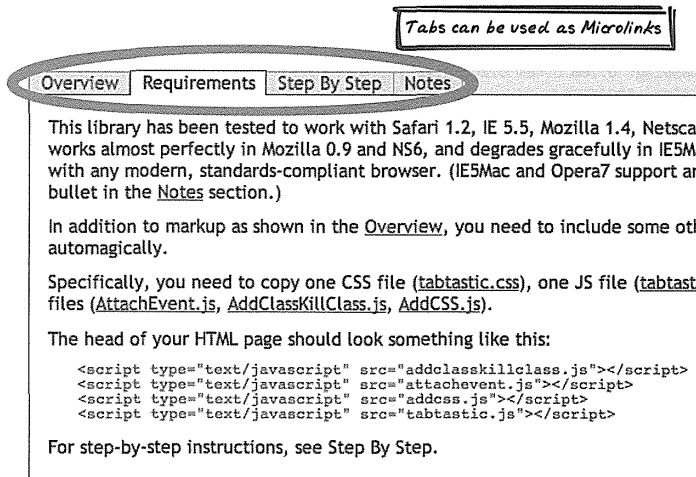


FIGURE 15-14. Tabtastic

Rpad

Rpad (<http://www.rpad.org/Rpad/>) provides an assortment of numerical analysis applications. Here, the linked content is not stored but dynamically generated. Click on Calculate within the General Demo (<http://www.rpad.org/Rpad/Example1.Rpad>) and you'll see several tables and graphs appear, freshly generated from the server.

Gmail

Gmail (<http://gmail.com/>) uses Microlinks to repeatedly switch main content around. There are *Microlinks* for all of your mailboxes, your contacts, for composing a new email, and so on.

The appearance is structured like most web sites—fixed-size blocks on the top and left containing links and search and the main content on the bottom right. However, most conventional web sites achieve this fixed structure by continuously reloading the entire structure, even if only the main content has changed. Gmail, however, is an Ajax App and only has to morph the main content area as each Microlink is clicked.

Code Example: TiddlyWiki

In TiddlyWiki, Microlinks are created as anchor elements with a custom CSS class, `button`. There's a general-purpose function used to create buttons, and the link creator delegates to it. It passes in display information as well as a function, `onClickTiddlerLink`, to handle what happens when the user clicks.

```
var btn = createTiddlyButton(place,text,subTitle,onClickTiddlerLink,theClass);
```

Buttons are anchor elements rendered with a special CSS class, which provides a different appearance while being hovered over or activated:

```
.tiddler .button {
  padding: 0.2em 0.4em 0.2em 0.4em;
  color: #993300;
}

.tiddler .button:hover {
  text-decoration: none;
  color: #ccff66;
  background-color: #993300;
}

.tiddler .button:active {
  color: #ffffff;
  background-color: #cc9900;
}
```

When the link is clicked, the handler function inspects the event to determine the desired content:

```
function onClickTiddlerLink(e)
{
  if (!e) var e = window.event;
  var theTarget = resolveTarget(e);
  var theLink = theTarget;
  ...
}
```

The new *Malleable Content* can then be inserted. An algorithm determines where it will be placed and the `createTiddler()` function creates a new div to host the content. If animation is turned on, the new content block will appear to “leap out” from the hyperlink into its own region of the page, as detailed in *One-Second Motion* (Chapter 16). Otherwise, the window performs a straightforward scroll to show the new content:

```
function displayTiddler(src,title,state,highlightText,
  highlightCaseSensitive,animate,slowly) {
  var place = document.getElementById("tiddlerDisplay");
  var after = findContainingTiddler(src);
  // Which tiddler this one will be positioned after
  ...
  var theTiddler = createTiddler(place,before,title,state,highlightText,
    highlightCaseSensitive);
  if(src)
  {
    if(config.options.chkAnimate && (animate == undefined || animate == true))
      anim.startAnimating(new Zoomer(title,src,theTiddler,slowly),
        new Scroller(theTiddler,slowly));
    else
      window.scrollTo(0,ensureVisible(theTiddler));
  }
}
```

Related Patterns

Malleable Content

Malleable Content (see earlier) is a companion pattern, because Microlinks are often used to conjure up Malleable Content blocks. However, a Microlink can also produce a more complex structure, too, such as the Gmail links that switch all of the main page content.

“One-Second” visual effects

The visual effects (*One-Second Spotlight*, *One-Second Mutation*, and *One-Second Motion*; see Chapter 16) help the user comprehend what’s going on when a Microlink is clicked, as TiddlyWiki demonstrates.

Drilldown

The categories in a *Drilldown* (Chapter 14) are a kind of Microlink, as they cause the Drill-down content to change without a page reload.

Popup

A Microlink may be used to launch a *Popup* (see earlier).

Live Form

A *Live Form* (Chapter 14) can include Microlinks to more advanced controls and to content that supports the user’s decision making.

Portlet



Block, Conversation, Dialogue, Gadget, Independent, Parallel, Portal, Portlet, Widget

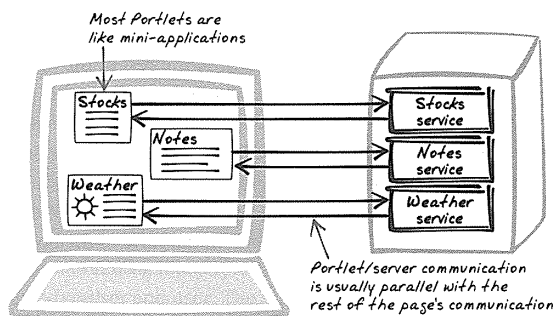


FIGURE 15-15. Portlet

Goal Story

The sidebar of Tracy's stock trading system contains a few independent blocks down the side: a currency Portlet, a calculator, and a miniature chart generator. Right now, she is checking on the USD/EUR exchange rate in one of those blocks. She types "USDEUR" into the text field, and the block updates a moment later to show the rate. Everything else on the page remains the same.

Problem

How can the user conduct rich conversations in parallel?

Forces

- A complex application contains a range of loosely related functionality. The user often needs to work on these things in parallel, gathering knowledge about one thing and using it to influence another.
- It isn't just different functionality, but different data that users need to work with in parallel.
- These conversations need to be "rich." They require conversational state to keep track of what's occurred so far and what can happen next, and they require interaction with the server.
- It's important to retain the state of each conversation between the user and the application. You don't want to invalidate one activity just because the user chose to initiate another.

Solution

Introduce Portlets—isolated blocks of content with independent conversational state. A Portlet is like a mini-application inside a regular application. It presents a little bit of information and usually some controls. The user can communicate via the controls, which causes the Portlet to update.

Usually, the Portlet state is completely independent of the rest of the application. For example, a travel web site can include a weather Portlet. Users are able to search for locations and drill down from a result to an extended forecast. The conversation can be as rich and stateful as the display allows. However, the conversation has no effect on the rest of the application. If the main site is in the middle of a booking transaction, it will stay that way while users spend a few minutes checking the weather. Browser/server communication will take place, but it won't affect the conversational state of the booking transaction, nor will it affect the state of any other Portlets.

In some cases though, actions on the Portlet can affect the general application state, and vice versa. A search Portlet can allow a user to locate content in the application. While locating the content, the user conducts a rich conversation alone with the Portlet, but once located, the user can click on the search result to open it up in the main content area.

If you aggregate Portlets together on the same page, you will end up with a Portals. A news Portal for example, contains a Portlet for each news category. But Portlets need not reside in Portals. A single Portlet can also be embedded in a regular web app (Figure 15-16), as the preceding example illustrates.

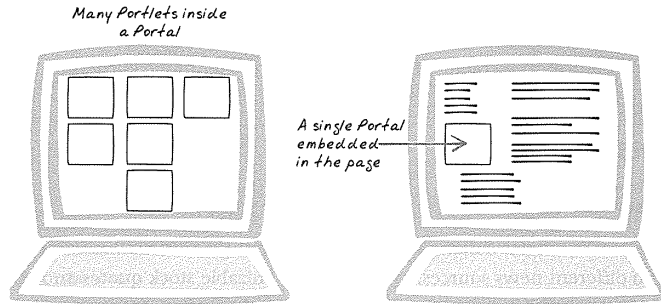


FIGURE 15-16. Portals and embedded Portlets

Portlets have been around for a long time, and Portals were once touted as the next big thing after search engines. However, they've never really taken off, and the interface is part of the problem. Chapter 1 explains in detail the problems with traditional portals and how much they gain from Ajax. The various Ajax Patterns help in different ways:

- *XMLHttpRequest Calls* alleviate the need to reload the entire page on each interaction. The response is minimal, and the state of everything else is effortlessly retained—a truly parallel conversation.
- *Display Morphing* lets you change the portal's appearance in response to user events and server responses.
- *Periodic Refresh* allows you to auto-update Portlets based on external changes and activity from other users.
- *Drag-And-Drop* lets you pick up Portlets and drag them around, a great improvement over the contrived page rearrangement interfaces used in the past.

An interesting variant is the “Cross-Domain Portlet,” a mixture of *Cross-Domain Proxy* and *Portlet*. The idea is to let the user have a complete conversation with an external web site. Consider how this might impact Tim O'Reilly's proposal of “Advertising as Conversation” (http://radar.oreilly.com/archives/2005/05/advertising_as.html), where he suggested that web ads can work more as interactive conversations between user and advertiser rather than simply being mindless branding imagery. A Portlet can be used to mediate a complete conversation between advertiser and user. The preceding weather Portlet example could actually come from an external service provider. The travel site could even contain a booking Portlet. While viewing online discussions about a destination, a user can conduct a parallel conversation with an airline company to check flight availability.

Decisions

Will users be able to move Portlets around?

Drag-And-Drop provides a nice way to rearrange Portlets on the page, but do you really want to? Generally, Portal-style applications support this activity, but for regular applications with one or two Portlets, you need to decide if it's worthwhile. Often, you can support some constrained movement—for example, let the user move Portlets up and down a side column.

Real-World Examples

Claude Hussenet's Portal

Claude Hussenet's portal is a demo informational portal (<http://claudehussenet.com>). Information from different news sources, as well as customizable stock quotes and quick links (bookmarks), are available in independent Portals. Each can be refreshed independently, and all editing occurs in the Portlet and without reloading the entire page. A checkbox lets you show and hide each Portlet individually.

Google Homepage

Google Homepage (<http://www.google.com/ig>) is a portal with three columns of Portlets. Registered users can click on "Add content" to keep adding Portlets. Most of the Portlets, like news, RSS feeds, and Word of the Day, contain recent updates. There are also some more interactive Portlets: Search shows personal search history, Bookmarks lets you accumulate bookmarks in a Portlet, Gmail summarizes recent mail messages. All Portals' existing Portlets can be dragged around the page and edited. Editing morphs the Portal into a form. At present, unfortunately, the form is submitted as a regular form, causing a page refresh.

Backbase

The Backbase Portal Demo (<http://projects.backbase.com/RUI/portal.html>) contains several independent Portlets, each with its own state (Figure 15-17). This is only a demo, but it's a good demonstration of the independent nature of Portlets.

Dobrado

Dobrado (<http://dobrado.sourceforge.net>), an open source project, is different from the preceding information portals in a couple of ways. First, the *Portlets* are free-flowing—they can be dragged anywhere on the page as opposed being restricted to the more common style of docking a Portlet into one of three columns. Second, Dobrado is not actually an information *Portal* per se, but rather a web site authoring application; for novices, it's easy to create a page by conjuring new Portlets, arranging them on the page, and typing in some content.

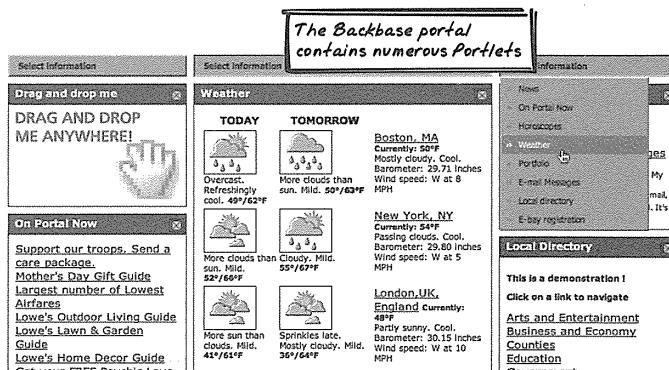


FIGURE 15-17. Backbase

Code Example: Claude Hussenet's Portal

Claude Hussenet's portal (<http://claudehussenet.com/>) delegates server calls to the Direct Web Remoting (DWR) framework (<http://www.getahead.ltd.uk/dwr/>). Let's focus on the "Quick Links" Portlet, which allows you to maintain a collection of bookmarks. (Some of the following code here has been reformatted.)

The initial HTML represents Quick Links as a table. There's a span for the links and another span for input:

```
<TABLE border="0" cellPadding="0" border="0" cellSpacing=0 width="100%" >
  <tr><td height="2" colspan="2" bgcolor=#FF6633></td></tr>
  <tr>
    <td valign="top" nowrap><B>Quick Links</B></td>
    <td width=1% align=right> <input type="checkbox"
    id='buttonMinimizedPortletQuickLinks'

      value="Min" onclick="refreshPortletQuickLinks
      (document.getElementById('portletQuickLinks'));"></td>
  </tr>
  <tr>
    <td>
      <span id="portletQuickLinks" ><B>Loading Content</B></span>
    </td>
  </tr>
  <tr>
    <td>
      <span id="inputLinksPortletQuickLinks" ></span>
    </td>
  </tr>
</TABLE>
```

A call is made on startup to get content from an *HTML Message* on the server. This delegates to a DWR function, and the server call finally morphs the links:

```

QUICKLinks.getContent(writePortletPortletQuickLinks,
    {timeout:3500,
      errorHandler:'sayHello2('+"'portletQuickLinks"+"')'}));
...
function QUICKLinks() { }
QUICKLinks.getContent = function(callback) {
    DWREngine._execute('/portal/dwr', 'QUICKLinks', 'getContent', callback);
}
...
function writePortletPortletQuickLinks(data) {
    if (data!=null)document.getElementById("portletQuickLinks").innerHTML = data;
}

```

The input for new links has a Submit button, and when the user submits, a JavaScript function is invoked. It causes the new link details to be posted to the server. Because the server is keeping track of the session, it is able to send the HTML back for *all* links. The callback function is the same as before:

```

function addQuickLinksPortletQuickLinks(elem) {
    var name = document.getElementById("linkname").value;
    var link =document.getElementById("url").value;
    ...
    QUICKLinks.addLink(writePortletPortletQuickLinks,name,link);
}

```

Alternatives

Tabbed Browsing

Tabbed browsing is a workaround users resort to in order to conduct multiple conversations with the same web site.

Related Patterns

Periodic Refresh

Portlets often use *Periodic Refresh* (Chapter 10) to stay up to date.

Distributed Events

Using *Distributed Events* (Chapter 10) is a good way to keep the Portlet updated. This is particularly important when one Portlet is dependent on another: instead of tying them directly, use events to add a layer of indirection.

Drilldown

A dynamic *Drilldown* (Chapter 14) is an effective way to conserve space within a Portlet.

Live Form

Live Forms (Chapter 14) appear in many Portlets because they are a good way to conduct a rich conversation with the server in a manner that avoids any page refresh.

Cross-Domain Proxy

Portlet can be used to show a conversation to the user and to an external domain.

Metaphor

A portal full of Portlets is like having a 21st century dashboard as depicted in a 20th century sci-fi movie—a conversation with the president on one screen, a little navigation through the stars on another screen, and a looping fast-food ad in the middle.

Acknowledgments

Tony Hill of Thomson Consulting gave me the idea for the pattern diagram.

Status Area



Console, Log, Message, Status

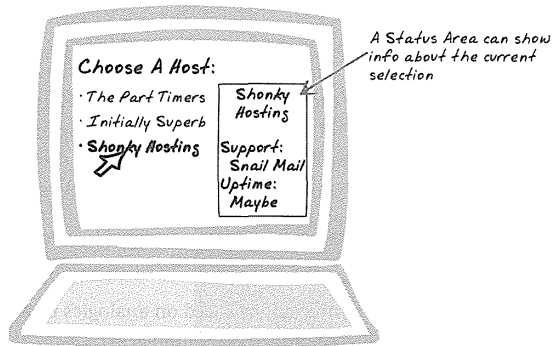


FIGURE 15-18. Status Area

Goal Story

Pam wants to revise the list of project issues. As she begins checking off those that are now resolved, she notices the Status Area below update, which is showing the number of issues resolved and the number still outstanding.

Problem

How can you show auxiliary information?

Forces

- The same data can be represented in different ways.
- Users often benefit from redundant summary information.

- Screen real estate is limited; you can't augment each field with its own summary information.

Solution

Include a read-only Status Area to report on current and past activity. The Status Area is usually auto-generated text based on some aspect of system state. The main purpose is to save space by occupying a region with information from different sources. This is done by dynamically altering the information according to current context.

Applications of the Status Area include the following:

- Summarizing information about elements the mouse is hovering over.
- Summarizing information about the element being edited.
- Summarizing information about the overall application state.
- Capturing past events in a log.
- Offering a preview.

Often, there's no server-side processing involved—the browser has enough information to maintain the Status content itself. For example, the browser can easily show a count of selected elements or a log of past data that's been retained. A Status Area can be particularly valuable for monitoring the state of dynamic objects. For example, an e-commerce system can use *Periodic Refresh* (Chapter 10) to continuously update the state of an order within a user's profile ("Submitted," "Credit Card Verified," "Stock Available," and so on).

The Status Area is usually a div element, with changes triggered by events such as mouse rollovers and form editing.

Note that this pattern is mostly speculative and is based on analogies from conventional desktop systems (where Status Areas are indeed commonplace).

Decisions

How will you size the Status Area? What if it overflows?

The Status Area is usually a relatively small element—sometimes just one row of text. You need to perform some analysis to determine the worst-case situation, i.e., what's the most content the Status Area can hold, and how will you deal with overflow. Strategies for dealing with this include:

- Compressing the text somehow—e.g., by trimming the message or extracting a summary.
- Introducing scrollbars. This is reasonable for a console-like Status Area that retains a history.
- Dynamically resizing the Status Area (which is not very common).

How will you structure the Status Area's content?

The Status Area need not be plain-text. It's often useful to keep a common structure in which each position always reflects the same variable. Consider which variables are being maintained and how they relate to each other.

Real-World Examples

BetFair

Betfair (<http://betfair.com>) includes a *Live Form* (Chapter 14) for creating new bets (see "Real-World Examples" in Live Form). A Status Area tracks your total liability, which is dynamically updated as you change the stake.

Lace Chat

Brett Stimmerman's Lace Chat (<http://www.socket7.net/lace/>) is an Ajax chat app. A Status Area contains a live preview of your message. As you type, a preview of the output, including any markup, is shown.

Code Refactoring: AjaxPatterns Status Wiki

The Basic Wiki Demo (<http://ajaxify.com/run/wiki/>) is refactored in the Status Wiki Demo to include a *Status Area* (<http://ajaxify.com/run/wiki/status>). While the focus is on a message, there's a Status Area below maintaining the word count, row count, and character count (Figure 15-19).

Message ID	10
Row Count	4
Word Count	11
Character Count	55

10.Mike.100.0

This is the first wiki entry!!!
Oh, one more thing ...|

FIGURE 15-19. Showing the status of a wiki entry

The static HTML has been refactored to include a Status Area with a table for all the values. This means the script need only set the actual count values rather than perform HTML manipulation:

```
<div id="status">  
  <table id="statusTable">  
    <tr>
```

```

        <td class="label">Message&nbsp;ID</td>
        <td class="value" id="messageId"></td>
    </tr>
    <tr>
        <td class="label">Row&nbsp;Count</td>
        <td class="value" id="rowCount"></td>
    </tr>
    <tr>
        <td class="label">Word&nbsp;Count</td>
        <td class="value" id="wordCount"></td>
    </tr>
    <tr>
        <td class="label">Character&nbsp;Count</td>
        <td class="value" id="characterCount"></td>
    </tr>
</table>
</div>

```

The CSS stylesheet makes the status initially hidden:

```

#status {
    visibility: hidden;
    text-align: center;
}

```

When the message gains focus, it makes the status visible and calls `showStatus()` to set its values according to the message's initial state. The Status Area remains until the message is blurred. To update on each change, `showStatus` is called on keyup:

```

function onMessageFocus(event) {
    ...
    $("#status").style.visibility = "visible";
    showStatus(message);
}

function onMessageBlur(event) {
    ...
    $("#status").style.visibility = "hidden";
    ...
}

function onMessageKeyUp(event) {
    var message = getMessage(event);
    showStatus(message);
}

```

The `showStatus` message analyzes the message and posts its results to the status-table cells:

```

function showStatus(message) {
    $("#messageId").innerHTML = message.id;
    $("#characterCount").innerHTML = message.value.length;
    $("#rowCount").innerHTML = message.value.split('\n').length + 1;
    var messageCopy = message.value.replace("\n", " ");
    messageCopy = message.value.replace(/^ */, "");
}

```

```

        messageCopy = messageCopy.replace(/ *$/, "");
        $("wordCount").innerHTML = messageCopy.match(/^ *$/) ? 0:messageCopy.split(/\s+/g).
length;
    }

```

Alternatives

Popup

A *Popup* (see earlier) is another way to show auxiliary information without dedicating a permanent space on the page for it. A Status Area is less intrusive but at the expense of some screen real estate.

Browser Status Bar

Most browsers contain a status bar at the bottom, which you can access with the window.status object. That's an okay place to include status information, but it has drawbacks. First, it's a very rudimentary interface with no support for structured content, styling, and animation. Second, users expect URLs to appear there when they hover over hyperlinks, and you might be breaking that model. Finally, there are cross-browser issues. Firefox, for instance, disables scripting the status line by default due to security concerns (malicious web sites can fake the URL you're heading to).

Related Patterns

Periodic Refresh

Where the Status Area shows server-related status, use a *Periodic Refresh* (Chapter 10) to keep it up to date.

Metaphor

A Status Area is like a time-share property; both reduce real estate costs by letting multiple participants occupy the same area at different times.

Update Control



Backwards, FastForward, Forwards, Freeze, Pause, Push, Refresh, Rewind, Reverse, Speed, Sticky, Update

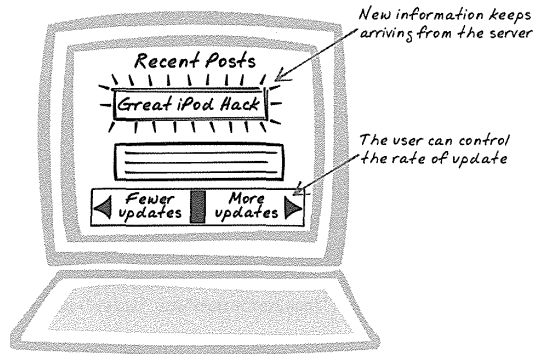


FIGURE 15-20. Update Control

Goal Story

Sasha is entranced by a world map that continuously updates with events from around the world. She plays around with the velocity controls, first speeding it up so there's a new event every second (no matter how trivial), then backtracking through previous events, and finally pausing on something that catches her eye.

Problem

How can the user deal with continuous information entering the browser and updating the page?

Forces

- Many Ajax Apps continuously grab data from the server and place it on the page.
- The web page has a limited area, so you'll need to remove or relegate older content.
- The optimum speed of updating depends on the user and his task. It should be fairly quick if the user is actively monitoring, for example, and probably quite slow if the content is a bit of eye candy at the edge of a page.

Solution

Let the user control the rate and criteria of updates. When using patterns such as *HTTP Streaming* (Chapter 6) and *Periodic Refresh* (Chapter 10), it's possible to keep grabbing fresh content from the server, so you can show news updates, system events, and more. The trend is increasing, with photo slideshows, news updates, and so on, and it can easily lead to information overload. This pattern is about giving the user control over the incoming stream of information. It takes several forms.

First, the user can control the rate of change, by pausing, rewinding, and fast-forwarding. Pausing is important for several reasons: it gives the user an ability to reflect on the content, it lets him keep the content open while perform work related to it, and it lets him

interact with the content in the case that it contains links or other forms of control. This is a serious issue in interfaces like Digg Spy (<http://digg.com/spy>) that stream new links every second or so. Go to click a link and—BAM!—it's already been replaced by a new link by the time your mouse pointer gets there.

Rewinding is useful too, because it lets users see things they didn't catch the first time, or lets them revisit something in the light of new information.

Fast-forwarding is another form of Update Control. More generally, this relates to setting the speed of updates to be faster or slower than the application's default. With this tool, users can tailor update behavior to their own needs and the task they're performing. Different users will have different mental processing capabilities and preferences—some are hungry for a torrent of incoming data, whereas others prefer a more casual pace. Furthermore, consider how the information is being used. In one case, the user, perhaps a trader monitoring company news, might be actively watching the information. In another case, the user might be trying to find an interesting story to read while viewing a list of new RSS items. Instead of second-guessing how the user's using all of this content, the advice in this pattern is to set a suitable default rate and give her the power to change it.

In theory, you only need a single "speed control" to allow for Update Control. A negative speed corresponds to rewind, zero to pause, one to default speed, and a high number for fast-forward. This could be set with a *Slider* (Chapter 14) or an input field or both. However, you can probably do a better job than that. The media player metaphor is particularly compelling given how closely it relates to this problem as well as how universal the concepts are. You may also opt to control the speed by adapting to the user's behavior. If the mouse is hovering near some content, that's probably a good clue to pause it, especially if the content is interactive.

You can also let the user choose what kind of content will be shown, which is an indirect way of influencing the rate. For example, a user might select between "Critical", "Informational," and "All." Or the criteria may be more domain-specific, as with Digg Spy, which provides checkboxes for the kinds of things you want to monitor—new stories, new comments, and so on.

Real-World Examples

Digg Spy

Digg Spy (<http://digg.com/spy>) shows new stories and events such as user moderation as they happen (see Figure 5-5). The information can change so quickly that it's difficult to click on a link, but fortunately a pause button is present. In addition, you can indirectly control how fast the content appears by tweaking the filtering options, so, for example, you can ask Digg to show only you new story submissions.

Slide

Slide (<http://Slide.com>) shows a stream of visual content from Flickr, EBay, and elsewhere (Figure 15-21). A minus (-) button moves content to the left, a pause button pauses it, and a plus (+) button speeds it up. It will also pause when you hover the mouse over it. The rolling slideshow is implemented with Flash, but the idea is equally applicable to pure Ajax.

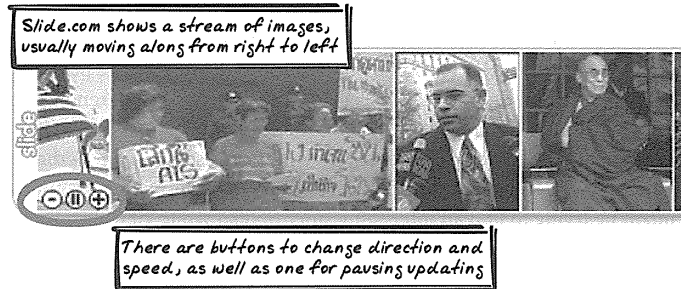


FIGURE 15-21. Slide

WBIR

WBIR (<http://wbir.com/>), a regional news provider, sometimes shows “slideshow images” (Figure 15-22). Like animated GIFs, these are a sequence of several images—shown one at a time—in the same container. But unlike animated GIFs, you can click to pause a single image.

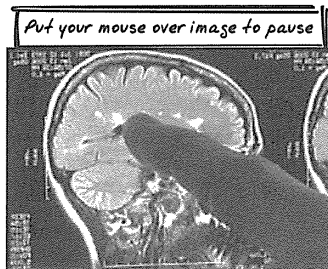


FIGURE 15-22. WBIR.com image

Code Example: Digg Spy

Digg Spy (<http://digg.com/spy>) includes a pause and a play button in the initial HTML. At any point in time, exactly one of these buttons is active and will respond to a click, which is registered to notify the `togglePause()` function. `togglePause()` doesn’t actually check which button the event originated from, because it instead tracks the pause mode with an

explicit pause variable. Thus, the first task of `togglePause()` is to toggle the state of the pause variable.

```
function togglepause() {
    pause = !pause;
    ...
}
```

If we've just entered pause mode, `togglepause()` cancels the timers used to make the updates. Otherwise, it does the exact opposite, i.e., schedules the updates, as well as making an initial update.

```
function togglepause() {
    ...
    if (pause == 1) {
        clearInterval(timer);
        clearTimeout(timer2);
        ...
    } else {
        update();
        timer = setInterval('addaline(true)', scrollDelay);
        timer2 = setTimeout('update()', updateDelay);
    }
    ...
}
```

The last thing `togglepause()` does is call `write_pause`, which updates the container (`play-pause-toggle`) containing both buttons. The HTML is mostly the same in both cases, but the hyperlink to `togglepause()` changes location according to which button should be active.

```
function write_pause() {
    if (pause == 0) {
        document.getElementById('pause-play-toggle').innerHTML = '<span class=
"spy-play"><strong>Play</strong></span><a href="#" onclick=
"togglepause()" class="spy-pause">Pause</a></span>';
    } else {
        document.getElementById('pause-play-toggle').innerHTML = '<a href=
"#" class="spy-play" onclick="togglepause()"><strong>Play</strong></a><span
class="spy-pause"><strong title="Pause the display of new items.">Pause</strong></span>
';
    }
}
```

Related Patterns

Periodic Refresh

Periodic Refresh (Chapter 10) is one means of keeping page content fresh—hence, a situation in which you might use Update Control.

HTTP Streaming

Like *Periodic Refresh* (Chapter 10), *HTTP Streaming* (Chapter 6) is a pattern that keeps page content fresh, the same situation in which Update Control applies.

Metaphor

Think of the navigation controls on a DVD player; conceptually, Update Control is very similar.

Acknowledgments

Thanks to Christopher Kruslicky for suggesting a pattern based around pausing, which ultimately led to this pattern.

Virtual Workspace



Camera, Desktop, Illusion, Infinite, InfiniteScrollbar, Lens, Move, Pan, Portal, Scroll, Solipsism, Viewport, Virtual, Visible, Window, Zoom

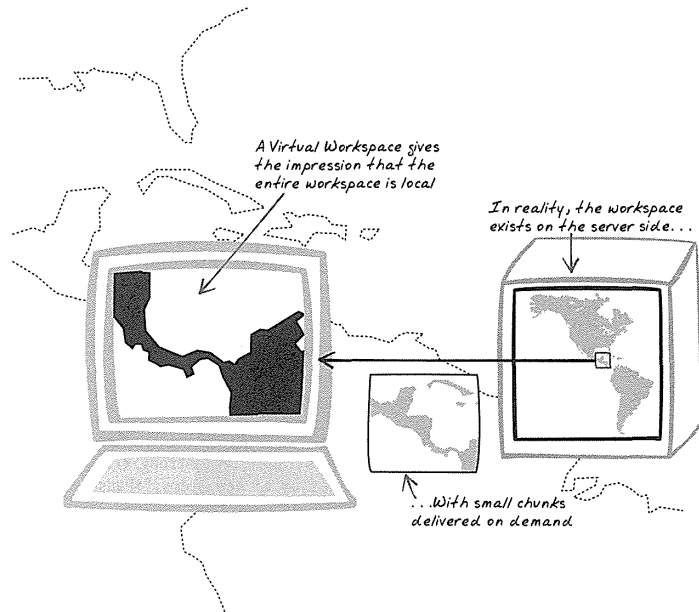


FIGURE 15-23. Virtual Workspace

Goal Story

Bill is using a meta-search engine to receive insurance quotes. The browser can't retrieve all 500 results at once, but Bill sees a table containing what looks like all the results. He can scroll up and down as if it were a regular table, and the newly appearing rows are populated on demand.

Problem

How can the user navigate through a large workspace?

Forces

- Ajax Apps often require data from the server.
- On the server side, data storage is virtually unlimited—search engines, for example, store billions of web pages. Even on corporate intranets, query results are often huge; try searching for every retail transaction on a given day.
- The application should respond to data requests as quickly as possible.
- Bandwidth constraints mean it's not feasible for an interactive application to download the entire search result. Yet users should be free to explore the whole thing.

Solution

Provide a browser-side view into a server-side workspace, allowing users to navigate the entire workspace as if it were held locally. The illusion is that the entire workspace is already in the browser, but the reality is that the server actually provides content on demand. At any time, the user is looking at an “opening” or “portal” into the entire workspace. He can pan across, jump to a different region, and zoom in and out. Each of these actions requires a view change, so the browser transparently fetches the data for the new portion of the workspace and renders it accordingly.

Here are some examples of navigable workspaces:

Lists

These include search results and user preferences.

Tables

These include *Data Grids* and spreadsheets.

Physical images

These include diagrams, photos and maps.

Documents

These include web content.

3D Models

These include computer-rendered landscapes and visualizations of virtual models such as chemical molecules.

Time-dependent artifacts

These include any artifact that changes with time—world population, for instance—and can be “navigated” by moving across time.

Users move through the space in different ways. Often they use a combination of the tools described next.

Scrollbars

Users can make the content scroll incrementally or jump to a completely different region.

Dragging tool

Gives the user the impression of dragging the document around while keeping the view fixed.

Keyboard shortcuts

Users can scroll incrementally using directional keys. Page Up and Page Down are often used to completely replenish the current view with an adjacent region. Home and End are often used to jump to the start and end.

Direct command

Users can use a separate control to specify an exact region—for example, typing in the dimensions they want to view. Or, for an image, placing a bounding box on a thumbnail view of the entire document.

There are also different zooming techniques:

Zoom slider

This is an adjustable *Slider* that represents the current zoom level.

Keyboard shortcuts

A common choice for zooming in and out is - and + (or =, since + usually requires pressing Shift+=).

Selecting a region

This is used to zoom further into a region within the current view.

Depending on the mechanisms you're supporting, you'll need to add different types of event handlers. So, for keyboard shortcuts, watch for events like `keydown`; for dragging and selecting a region, watch for `mousedown`, `mousemove`, and `mouseup`. In some cases, you'll need a control separate from the view itself, as is the case with a zoom slider or a thumbnail sketch of the entire workspace.

Whatever the event mechanism, the upshot is that the browser script will sometimes receive notifications that there is a new desired region or zoom level, if that's applicable. At that point, a Web Remoting (Chapter 6) call must occur, passing the server the details of the new region. Upon reply, the old data is either replaced or shifted along and the new data is rendered.

This pattern is often very bandwidth-intensive, considering that the entire workspace can be massive and the interaction complex. For that reason, there are several important performance optimizations—for more details, see "Related Patterns" later in this chapter.

Decisions

How will you handle panning?

Panning, as opposed to jumping straight from one position to another, provides some unique design challenges. Unlike a complete jump, the user will expect a smooth transition from one position to another. Fortunately though, you already have most of the workspace loaded. Thus, a *Browser-Side Cache* (Chapter 13) is very important if you wish to achieve smooth scrolling. With this, you need only to load the new portion of the workspace instead of the whole thing. This leads to a few more specific questions:

- How much of the workspace will you cache? A standard caching trade-off between memory and speed.
- How will you track changes? Again, this should be the responsibility of the cache. At one level, the script will ask for the whole workspace, but the cache will decide which portion of that actually requires server content.

How will the view appear initially?

There has to be a default view position and zoom level within the overall workspace. A typical choice for the view is at the logical start or center. Zoom level should usually be quite high to let the user quickly drill down from the starting point.

What do you display while a region is being repopulated?

Most of the time, users are navigating to areas that are partly or completely unpopulated, requiring a result from the server to render them. What do you show in the interim? Here are a few options:

- Whatever was there previously, possibly with a change of appearance.
- Nothing—clear it while waiting.
- A *Progress Indicator* (Chapter 14).
- A *Popup* (see earlier in this chapter)—perhaps with just a single word—indicating what content will be placed there.
- A *Guesstimate* (Chapter 13) of the content, perhaps made by extrapolating neighboring data held in the cache.

How will you handle changes to the existing view?

Sometimes the workspace changes while the user is watching it. For example, a user might introduce a filter to a result set. A change to the workspace means the view must change too. The easiest approach is to abandon the previous view location and revert to the default. However, there is often a more logical solution. Following are some examples:

- You could keep the proportions the same as before—e.g., if the user was looking at rows one-third of the way down a table, then show the new rows that appear one-third of the way down the reduced table.
- You could fix on certain content—e.g., keep the top row in the same spot and show the remaining rows that are now directly below it. If the top row no longer exists, continue working downward until one of the rows does exist.

Real-World Examples

map.search.ch

<http://map.search.ch> is a Swiss Ajax map. Like Google Maps (<http://maps.google.com>) (which it predates) and similar products, the map constitutes a huge Virtual Workspace, and the user only views a tiny portion of it at any point in time. The map can be panned by dragging the workspace, clicking on arrow icons just outside its boundaries, and pressing the arrow keys. Zooming is controlled by clicking on a horizontal imagemap, clicking on Zoom In and Zoom Out buttons, or pressing Page Up and Page Down.

OpenRico Search Demo

The OpenRico Search Demo (<http://openrico.org/rico/yahooSearch.page>) is a *Cross-Domain Proxy* (Chapter 10) providing an Ajax interface to Yahoo! Search (Figure 15-24). Its philosophy was best summed up by OpenRico developer Bill Scott as “Death to Paging!” (<http://looksgoodworkswell.blogspot.com/2005/06/death-to-paging-rico-livegrid-released.html>). Instead of wading through a sequence of pages, you’re presented with a single table containing all results. The results are a Virtual Workspace, and the table is a view into that space. Each time you navigate within the table, new results are pulled down from the server.

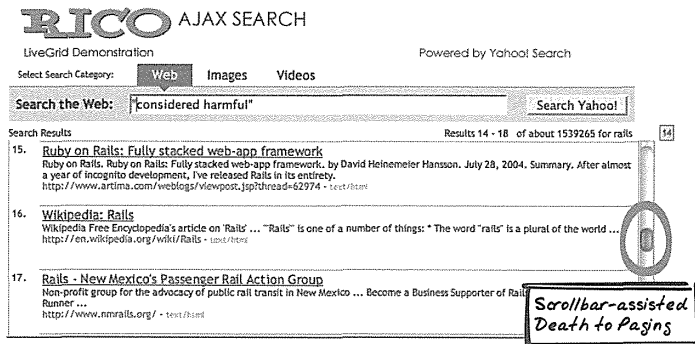


FIGURE 15-24. OpenRico Search Demo

Giant-Ass Image Viewer (GSV) library

Michael Magurski's GSV (<http://mike.teczno.com/giant/pan/>) is a library that lets web developers show an image of any size and allow the user to pan and zoom within it. The home-page contains a working demo.

Dunstan Orchard's blog

Dunstan Orchard's blog (<http://www.1976design.com/blog/colophon>) presents a slick panoramic, cartoonish view from his home. When the mouse rolls anywhere on the image, a couple of transparent *Popups* (see earlier in this chapter) appear on each side of the panorama. Rolling onto either of those and keeping the mouse there causes the banner to pan in that direction. Note that this is a different form of Virtual Workspace, since the whole workspace is local.

Code Refactoring: AjaxPatterns OpenRico Search

The OpenRico Search Demo is based on the OpenRico's LiveGrid API. In *Data Grid* (Chapter 14), the code example shows how to use the API. This example covers some of the API internals, specifically regarding the inclusion of a Virtual Workspace. Note that OpenRico uses Prototype (<http://prototype.conio.net/>) to allow for a more object-oriented coding style.

In OpenRico, the user's view is a `GridViewPort` object, which has a fixed row height and also tracks the view's starting position, i.e., the index that the top row of the `GridViewPort` corresponds to.

```
Rico.GridViewPort.prototype = {  
  initialize: function(table, rowHeight, visibleRows, buffer, liveGrid) {  
    ...  
    this.rowHeight = rowHeight;  
    this.div.style.height = this.rowHeight * visibleRows;  
    ...  
    this.startPos = 0;  
  },  
};
```

The results might include thousands of virtual rows, but the table itself is only about 20 rows (as determined by `visibleRows`). How, then, is the scrollbar created to make it appear as if there were thousands of rows? OpenRico creates a custom scrollbar. The trick is to create a 1-pixel-wide div whose height matches the height of the Virtual Workspace. The virtual height can be calculated since the scrollbar has access to the visible table height (`visibleHeight`), the number of virtual rows (`metaData.getTotalRows()`), and the number of rows in the view (`metaData.getPageSize()`). The scrollbar div's height is set to this virtual height, so the browser will render a scrollbar that appears to scroll across the entire virtual table:

```
createScrollbar: function() {  
  var visibleHeight = this.liveGrid.viewPort.visibleHeight();  
  // create the outer div...  
  this.scrollerDiv = document.createElement("div");  
  ...  
}
```

```

// create the inner div...
this.heightDiv = document.createElement("div");
this.heightDiv.style.width = "1px";
this.heightDiv.style.height = parseInt(visibleHeight *
    this.metaData.getTotalRows()/this.metaData.getPageSize()) + "px" ;
this.scrollerDiv.appendChild(this.heightDiv);
this.scrollerDiv.onscroll = this.handleScroll.bindAsEventListener(this);

var table = this.liveGrid.table;
table.parentNode.parentNode.insertBefore(
    this.scrollerDiv, table.parentNode.nextSibling);
},

```

Events on the scrollbar are dispatched to `handleScroll`. After any scrolling behavior has occurred, this function calculates the portion of virtual space being viewed. The algorithm determines the new virtual row that should appear on top of the viewport. For instance, if the row height is 10 pixels, and the user has scrolled to 50 pixels from the top, then the virtual row is 5; the viewport will need to be refreshed such that the new top row is the fifth virtual row.

```

handleScroll: function() {
    ...
    var contentOffset = parseInt(this.scrollerDiv.scrollTop / this.viewPort.rowHeight);
    this.liveGrid.requestContentRefresh(contentOffset);
    this.viewPort.scrollTo(this.scrollerDiv.scrollTop);
    ...
},

```

`requestContentRefresh` fetches the required content into a *Browser-Side Cache* known as *Buffer*. With the buffer in place, you can smoothly scroll back to previously seen results without any call required:

```

fetchBuffer: function(offset) {
    ...
    var bufferStartPos = this.buffer.getFetchOffset(offset);
    this.processingRequest = new Rico.LiveGridRequest(offset);
    this.processingRequest.bufferOffset = bufferStartPos;
    ...
    callParms.push('id=' + this.tableId);
    callParms.push('page_size=' + fetchSize);
    callParms.push('offset=' + bufferStartPos);
    ...
    ajaxEngine.sendRequest.apply( ajaxEngine, callParms );
    ...
},

```

Eventually, the viewport table is populated with the new set of rows:

```

refreshContents: function(startPos) {
    ...
    for (var i=0; i < rows.length; i++) { //initialize what we have
        this.populateRow(this.table.rows[i + contentOffset], rows[i]);
    }
    ...
},

```

Alternatives

Virtual Magnifying Glass

Instead of showing a partial view, you could present the entire thing in low detail and offer a “Virtual Magnifying Glass” or “Virtual Fish-Eye Lens” to zoom in on the detail. This might sound like it could only be applied to an image, but it is also applicable to tables and other interfaces.

Related Patterns

Browser-Side Cache

Browser-Side Cache (Chapter 13) is very important with respect to the Virtual Workspace—if the user moves 1 percent down, you don’t want to download the entire view again.

Predictive Fetch

It’s useful to perform a *Predictive Fetch* (Chapter 13) on regions of the Virtual Workspace that the user is likely to navigate to next. Panning is a common task, so it’s worthwhile caching the regions neighboring the current view. You might also cache at the next and previous zoom levels.

Guesstimate

Sometimes you might be able to quickly satisfy a navigation action with a *Guesstimate* (Chapter 13) while waiting for the real data to return.

Multi-Stage Download

If the response is large, break it into more than one part; download the most important information first, then follow up with more refined content.

Drag-And-Drop

Drag-And-Drop (see earlier) is often used to let the user pan by enabling her to drag the entire workspace across the view.

Slider

The zoom control is often a *Slider* (Chapter 14).

Unique URLs

Views need to be associated with *Unique URLs* (Chapter 17) so that the user can highlight a particular part of the workspace rather than the entire thing. That way, the user can easily bookmark the view or mail it to a friend.

Metaphor

Did you see *The Truman Show* (<http://www.imdb.com/title/tt0120382/>)? It's the story of a man who has no idea his whole world is fake, completely architected for the purposes of a reality TV show. As he walks around town, props are adjusted and the "townspeople" are directed a few seconds ahead of his arrival. There's a similar type of perception management going in Virtual Workspace; the illusion is that there's a whole world of content beyond the view, but the reality is that it's all constructed on demand.

Want to Know More?

Death to Paging—Bill Scott's LiveGrid Announcement (<http://looksgoodworkswell.blogspot.com/2005/06/death-to-paging-rico-livegrid-released.html>).

Acknowledgments

Thanks to Bill Scott for pointing out this example and keeping me updated on the progress of OpenRico and the LiveGrid functionality. Bill and fellow OpenRico developer Richard Cowin helped explain some of the implementation.

Visual Effects

ARICH INTERFACE IS MORE THAN STATIC WIDGETS; THERE'S A DYNAMIC COMPONENT THAT HAS A BIG impact on usability. Used carefully, visual effects can guide the user's attention and help him understand what the application is doing. And don't underestimate the aesthetic benefits. It might be eye candy, but *good* eye candy (not just "how good am I" hacks) makes passionate users.*

The "One-Second" effects are fleeting effects that sometimes convey a permanent transition from one state to another, and other times are used just to highlight something. *One-Second Spotlight* is a more general version of the popular "Yellow Fading Technique" and involves changes to brightness and color. In *One-Second Mutation*, the element changes shape, and in *One-Second Motion*, the element moves around.

* Donald Norman, traditional defender of "serious" usability principles, has more recently argued that emotions like fun and excitement do make a big impact and should be considered by designers (Norman, 2003). "Passionate Users" is a reference to the excellent Creating Passionate Users blog at http://headrush.typepad.com/creating_passionate_users/.

Highlight is a more permanent effect where one or more elements are highlighted, typically because the user has selected them.

One-Second Spotlight



Attention, Effect, EyeCandy, Fade, Fading, Graphics, Refresh, Update, YellowFadeTechnique

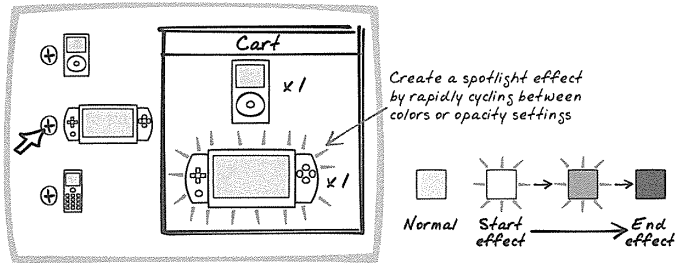


FIGURE 16-1. One-Second Spotlight

Goal Story

Tracy is checking out her portfolio, which shows each stock in its own row. Suddenly, “Neverland Mining Co.” stock plummets, and its row lights up to reflect the change. Tracy’s attention is drawn instantly to this visual effect, and she immediately begins to off-load her stake in the company.

Problem

How can you direct the user’s attention to spots on the page?

Forces

- To ensure that the user is working with current data, the browser display must be frequently updated.
- With Ajax, it’s possible to update only the portion of the screen that has changed, and if nothing has changed, the display will remain exactly as is.
- The screen can get cluttered with a lot of information, much of which might change at any time.
- While human vision is good at spotting changes, it’s easy to miss a sudden change, especially if it’s a subtle one.

Solution

When a display element undergoes a value change—or any other significant change—dynamically increase its brightness for about a second. As a device for communicating system activity, the One-Second Spotlight accomplishes several things:

- It highlights changes so the user doesn't have to actively monitor for changes and try to remember what—if anything—has changed.
- Because the spotlight lasts for a few seconds, the user has enough time to redirect attention to that item and study the details of the event—e.g., looking at an item's new value. In contrast, a standard transition would afford her no time whatsoever—if the user happened to blink, she could easily miss the change.
- When the animation technique follows certain conventions or mimics real-world phenomena, it can provide hints to the user about the nature of the event.

This pattern is a generalization of the technique 37signals introduced as the Yellow Fade Technique (<http://www.37signals.com/svn/archives/000558.php>). With that, an item lights up for a second after the user has edited it, and the lighting quickly fades away. Unlike a conventional application, in which a page refresh would occur, it's often unclear when an *XMLHttpRequest Call* has occurred. So a quick visual effect is almost essential for the user to understand what's happening.

From a technical perspective, the Yellow Fade Technique is usually achieved by suddenly switching the element's background-color style from white to yellow, then using *Scheduling* to run through a continuous spectrum of background colors, starting with yellow and ending with white.

In more general terms, One-Second Spotlight encompasses several types of animation. One or more of the following CSS styles are affected during the spotlight period:

color

Usually, the element's background changes, but font color is also a possibility. Though commonly assigned with Red-Green-Blue hex codes such as #ff00ff, animation algorithms can benefit from the percentage form, e.g., (100%, 0%, 100%). Often, the algorithm cycles through one or more of the Red, Green, and Blue components, e.g., gradually raising Red from 50 percent to 100 percent and lowering it back to 50 percent while holding blue and green constant. You might also prefer to represent the color as a combination of Hue-Saturation-Value (HSV) and cycle through one or more of those components. In this case, calculate the HSV values and convert them to RGB for display.

opacity

This style determines how transparent the element is; an opacity of zero means the item is solid, and an opacity of 50 percent means the browser background—or any items underneath the element—will be partially visible.

visibility

This style determines whether the element can be seen. Visibility does *not* affect page layout, so the element still occupies the same space on the page.

The possibilities are endless, but there are a few common effects. Firstly, there are straightforward transitions from one style setting to another:

Fade Out

An element suddenly brightens and then fades back to its original appearance. This is useful for general highlighting and may also indicate that the element has been “sucked into” the server, i.e., the data is now safely stored server side. Since many pages use a white background, fading the background out to white is a natural choice for general attention grabbing.

Fade In

An element is suddenly dimmed and then brightens up to its original appearance. This is useful for general highlighting and may also indicate that the element has been “pushed out” of the server.

Fade Away

An element fades further and further out until it disappears altogether. If the element is already somewhat dim, it may be helpful to brighten it at the start of the animation. This effect often indicates that the element (or its value) has been suspended or destroyed.

Materialize

From nothing, an element fades into its regular appearance. If its regular appearance is somewhat dim, it may be helpful to fade it in to a higher level, then fade back out a bit. This can indicate that the element has been created or retrieved.

Switch

An element gradually switches its appearance from one setting to another. Instead of fading or brightening, though, an element might shift from red to blue. This effect is usually used to indicate a state change.

The other category of effects involves rapid oscillation and includes the following:

Total Flash

In rapid succession, an element completely disappears and then reappears. The transition to and from a state of disappearance can be achieved by continuous fading, or by simply toggling the visibility style. This effect is useful as a general attention grabber, although it brings back memories of the dreaded `<blink>` tag; be sparing in your use of flashing elements.

Shifting Flash

An element’s appearance shifts several times from one setting to another. This can also be used as a general attention grabber if the element returns back to its original appearance. If the element permanently shifts from one setting to another, this effect will grab attention more effectively than *Switch*, which is a more subtle transition.

One typical implementation combines *Display Morphing* with *Scheduling*. To fade an item out, for instance, the following algorithm may be used:

1. Remember the element's current setting.
2. Set `element.style.color` to a bright setting.
3. Every 100 milliseconds:
 - a. Fade the element a bit. More precisely, drop color by 10 percent (of the bright setting). This applies individually to each color component (R, G, B).
 - b. Check if 1,000 milliseconds has already passed. If so, set the element back to its original setting (it should already be about that anyway).

The algorithm is based on interpolation. In moving from color A to color B, there are a number of discrete transitions that occur, say, once every 100 milliseconds. At each stage, the algorithm decides where on the spectrum from A to B the color should be. Since a color consists of three components—R, G, and B—the calculation occurs separately for each component.

So, let's say you're shifting from `#ffffff` to `#000044`. First, it's useful to represent these as decimal percentage settings: (100%, 100%, 100%) moving to (0%, 0%, 25%). And assume we're going to make 10 transitions. The red and green components will shift down 10 percent each step, and the blue component will shift down 7.5 percent each time. So it's just a matter of dropping the value for each color component at each transition, rounding off, and redisplaying the element.

If you want to make a transition to the background of the element, you can shift opacity instead of color. (Be aware that there are portability issues with opacity [<http://www.quirksmode.org/css/opacity.html>].)

The algorithm assumes a linear interpolation, where each transition drops by the same amount. You could also experiment with other transitions. An exponentially increasing interval, for example, would give an effect of an initially slow transition, leading to a quick burst at the end. A similar phenomenon is quite common with flash effects, where the flashing starts off slowly and speeds up at the end.

You probably won't have to hand code many of these effects because libraries like Scriptaculous (<http://script.aculo.us>; discussed below) are quite flexible and easy to incorporate.

Decisions

What events will trigger a spotlight?

An Ajax App undergoes multiple events; which warrant a spotlight? A spotlight can serve any of the following purposes:

- To draw the user's attention to a change
- To suggest that the user needs to do something
- To inform the user that a browser-server interaction has taken place
- To inform the user that another user has done something

Here are some examples:

- The user has just changed something. A *Fade Out* effect suggests that the change has been sent to the server.
- On the server, a new item has been created. A *Materialize* effect suggests that the item has just appeared.
- The user has forgotten to enter a form field. The field undergoes a *Shifting Flash* as its background shifts from white to red.
- An item has just changed value. A *Fade Out* effect suggests that the change was just picked up by the browser.

What color will be used?

Yellow, popularized by 37signals, is becoming something of a standard for fading effects. In general, though, the color has to take into consideration which element is affected, as well as surrounding elements and the page appearance in general.

Additionally, multiple colors can be used to good effect. For instance, each color can be used to represent a different type of event. In a multiuser context, JotSpot Live (<http://jotlive.com>) illustrates how each user can be associated with a different color so that their recent edits appear as distinctly-colored fade effects.

How long should the spotlight last? How long will each transition be?

Asking how long the spotlight should last might seem a bit ridiculous for a pattern named “One-Second Spotlight.” But the name, of course, only hints at one reasonable value. You’ll have to decide on the precise duration of the overall effect and also on how many transitions will take place in that time.

Here are some considerations:

- A longer duration will have a greater window of time to be noticed.
- A longer duration will appear less dramatic, so it might be interpreted as a less critical event.
- A longer duration increases the likelihood of multiple spotlights occurring simultaneously, which might be confusing, especially if they are flash effects.
- A longer duration increases the likelihood of a second spotlight effect occurring on the *same* element while the first effect is in progress. That’s not necessarily a problem in itself, but it does lead to an implementation issue; it’s worth ensuring that an element is undergoing only one effect at a time to prevent any nasty issues, like flickering.

As far as transitions go, the trade-off is fairly clear: faster transitions look smoother but slow down the application. Is the user likely to leave your application in a background tab and only check it once an hour? If so, avoid slowing down the whole browser with high-granularity effects. Likewise if the application is already process-intensive.

This is an area where different browsers will act differently. For instance, what's the lowest possible transition time you can set? That will depend on the user's system and the browser in question. You can experiment with different parameters on the Time Spotlight Demo (<http://ajaxify.com/run/time/periodicRefresh/spotlightCustom/>).

Real-World Examples

37signals Backpack

37signals Backpack (<http://www.backpackit.com/>) maintains items in a list (Figure 16-2). When you click the Edit button, the read-only item morphs into a textarea. On saving, it becomes read-only again and is accompanied by a yellow *Fade Out* effect. All quite fitting for the company that coined the term, “Yellow Fade Technique.”

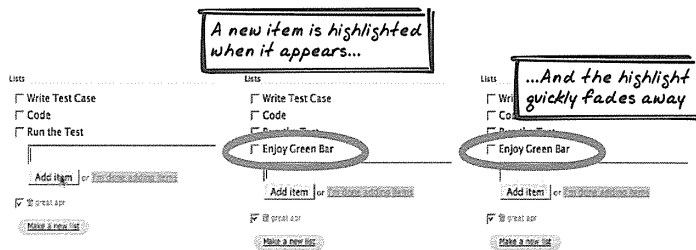


FIGURE 16-2. Backpack

chat.app

chat.app (<http://gamma.nic.fi/~jmp/chat/app.html>) shows text messages as you or others enter them. Each new message is highlighted with a yellow *Fade Out*.

Digg Spy

Digg Spy (<http://digg.com/spy>) shows new stories as they appear on the Web. Each time a block of new stories appears, it appears with a *Fade Out* effect—the background shifting from grey to white.

Coloir

Coloir (<http://www.couloir.org>) is a slideshow application. Each time you click next, an animated “loading” icon appears and a new photo fades in.

Odeo

Odeo (<http://odeo.com>) is a podcast subscription service. You can force Odeo to grab the latest feed for a podcast; during the delay, the entire document fades. An effect like this can be achieved with a partially transparent *Popup* element covering the whole document. It feels similar to the way Windows fades the entire screen when you're about to log out.

Scriptaculous library

Scriptaculous (<http://script.aculo.us>) is an excellent JavaScript library supporting many Ajax UI patterns. Its visual effects library is based on four fundamental effects:

- Altering opacity
- Altering scale
- Altering position
- “Parallel” effects—i.e., multiple effects at the same time

Combining these leads to any number of animation sequences, and many of the useful sequences are prepackaged.

The visual effects demo (<http://script.aculo.us/visual-effects>) provides many spotlight effects.

Fade Anything Technique (FAT) library

FAT (<http://www.axentric.com/posts/default/7>) is a library allowing easy scripting of a transition from one color to another.

Code Refactoring: AjaxPatterns Spotlight Time

The Time Periodic Refresh demo (<http://www.ajaxpatterns.org/run/time/periodicRefresh/>) shows the server time, which is updated every few seconds. It's not clear exactly when the time has changed, however, so this refactoring adds a spotlight effect each time a new result arrives from the server. There are actually a couple of refactorings here: one is hand-built and the other illustrates reuse of the Scriptaculous library.

Hand-built shift effect

The Time Spotlight Demo (<http://www.ajaxpatterns.org/run/time/periodicRefresh/spotlightCustom>) lets the user set a few parameters and applies the corresponding effect (Figure 16-3). A fader object encapsulates the fading logic. The fade duration and transition interval are modifiable attributes:

```
fader.durationTime = parseInt($("#fadeDuration").value);
fader.transitionInterval = parseInt($("#transitionInterval").value);
```

The application asks fader to fade the div as soon as it's been modified:

```
function showCustomTime(text) {
    var customTimeLabel = $("#customTimeLabel");
    customTimeLabel.innerHTML = text + "." + callingContext;
    fader.fade(customTimeLabel, $("#startColor").value, $("#endColor").value);
}
```

fader then kicks off the first transition, passing in a timeSoFar parameter of zero. The function is simpler using an array of percentage color values, but hex is more common, so a simple conversion takes place:

Fade Parameters

Start Color (RGB): #996666

End Color (RGB): #

ffffff

Total Duration (ms): 10000

e.g. 200 for a quick flicker, 2000 for a slow burn.

Transition Interval (ms): 1000

e.g. 20 for very smooth, 200 to save resources.

Default Time:

Jan 6 19:51:27

Refresh

Customised Time:

Fri Jan 6 19:51:27 EST 2006.55

Refresh

FIGURE 16-3. Time Spotlight Demo

```
fade: function(element, startRGB, endRGB) {  
    this.nextTransition(  
        element, this.rgbToPercents(startRGB), this.rgbToPercents(endRGB),0);  
    }  
}
```

Each transition involves deciding how far along the fade it is and calculating the corresponding color style using a linear interpolation (as explained in the Solution for this pattern):

```
nextTransition: function(element, startColor, endColor, timeSoFar) {  
    ...  
    var currentColor = "rgb(";  
    for (component=0; component<3; component++) {  
        var currentComponent = Math.round(startColor[component] +  
            proportionSoFar * (endColor[component] - startColor[component]));  
        currentColor+=currentComponent + "%" + (component<2 ? ", " : "");  
    }  
    element.style.backgroundColor = currentColor;  
    ...  
}
```

Then it increments timeSoFar and finishes the fade if fade time has expired:

```
timeSoFar+=this.transitionInterval;  
if (timeSoFar>=this.durationTime) {  
    this.durationTime+ "\n";  
    element.style.backgroundColor =  
        "rgb("+endColor[0]+"%", "+endColor[1]+"%", "+endColor[2]+"%";  
    return;  
}
```

```

var nextCall = function() {
    fader.nextTransition(element, startColor, endColor, timeSoFar);
}
setTimeout(nextCall, this.transitionInterval);

```

fader is coded for simplicity, and it is possible to make a couple of improvements. First, the repeated color calculation would be more effective if it worked on the pretransition delta instead of calculating the absolute color each time. Also, the timing could be adjusted to take into account the time taken to run the function itself.

Fading with Scriptaculous

The Time Scriptaculous Demo (<http://www.ajaxpatterns.org/run/time/periodicRefresh/spotlightScriptaculous>) shows how easy it is to reuse an existing effects library. Here, the time displays show a Materialize effect when a new result appears. They transition from completely invisible (showing only the document background color) to the appearance determined by their class's style.

The refactoring is trivial. Include the Scriptaculous effects library, effects.js, and prototype.js, a general-purpose JavaScript utility used by Scriptaculous:

```

<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript" src="effects.js"></script>

```

Then, as with the hand-built fader presented earlier, invoking the effect is a one-liner:

```

new Effect.Appear("defaultTimeLabel");

```

Alternatives

One-Second Mutation and One-Second Motion

One-Second Mutation and *One-Second Motion* (see both later in this chapter) are also used to draw attention to a significant event. One-Second Spotlight can be used as both a general attention-getting device as well as to hint at a specific type of event, whereas those two patterns tend to be slightly more about depicting specific events. The effects can also be combined, e.g., shrinking an object as it fades away.

Related Patterns

Periodic Refresh

A change arising from a *Periodic Refresh* (Chapter 10) is often marked with a One-Second Spotlight.

Timeout

A *Timeout* (Chapter 17) event, and any prior warnings, can be marked with a One-Second Spotlight. In community sites that show details about other users, the effect can also be used to mark changes to users' online status.

Fade Parameters

Start Color (RGB): #996666

End Color (RGB): #

ffffff

Total Duration (ms): 10000

e.g. 200 for a quick flicker, 2000 for a slow burn.

Transition Interval (ms): 1000

e.g. 20 for very smooth, 200 to save resources.

Default Time:

Jan 6 19:51:27

Refresh

Customised Time:

Fri Jan 6 19:51:27 EST 2006.55

Refresh

FIGURE 16-3. Time Spotlight Demo

```
fade: function(element, startRGB, endRGB) {  
    this.nextTransition(  
        element, this.rgbToPercents(startRGB), this.rgbToPercents(endRGB),0);  
    }  
}
```

Each transition involves deciding how far along the fade it is and calculating the corresponding color style using a linear interpolation (as explained in the Solution for this pattern):

```
nextTransition: function(element, startColor, endColor, timeSoFar) {  
    ...  
    var currentColor = "rgb(";  
    for (component=0; component<3; component++) {  
        var currentComponent = Math.round(startColor[component] +  
            proportionSoFar * (endColor[component] - startColor[component]));  
        currentColor+=currentComponent + "%" + (component<2 ? ", " : "");  
    }  
    element.style.backgroundColor = currentColor;  
    ...  
}
```

Then it increments timeSoFar and finishes the fade if fade time has expired:

```
timeSoFar+=this.transitionInterval;  
if (timeSoFar>this.durationTime) {  
    this.durationTime+ "n";  
    element.style.backgroundColor =  
        "rgb("+endColor[0]+"%,"+endColor[1]+"%,"+endColor[2]+"%";  
    return;  
}
```



```

var nextCall = function() {
  fader.nextTransition(element, startColor, endColor, timeSoFar);
}
setTimeout(nextCall, this.transitionInterval);

```

fader is coded for simplicity, and it is possible to make a couple of improvements. First, the repeated color calculation would be more effective if it worked on the pretransition delta instead of calculating the absolute color each time. Also, the timing could be adjusted to take into account the time taken to run the function itself.

Fading with Scriptaculous

The Time Scriptaculous Demo (<http://www.ajaxpatterns.org/run/time/periodicRefresh/spotlightScriptaculous>) shows how easy it is to reuse an existing effects library. Here, the time displays show a Materialize effect when a new result appears. They transition from completely invisible (showing only the document background color) to the appearance determined by their class's style.

The refactoring is trivial. Include the Scriptaculous effects library, effects.js, and prototype.js, a general-purpose JavaScript utility used by Scriptaculous:

```

<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript" src="effects.js"></script>

```

Then, as with the hand-built fader presented earlier, invoking the effect is a one-liner:

```

new Effect.Appear("defaultTimeLabel");

```

Alternatives

One-Second Mutation and One-Second Motion

One-Second Mutation and *One-Second Motion* (see both later in this chapter) are also used to draw attention to a significant event. One-Second Spotlight can be used as both a general attention-getting device as well as to hint at a specific type of event, whereas those two patterns tend to be slightly more about depicting specific events. The effects can also be combined, e.g., shrinking an object as it fades away.

Related Patterns

Periodic Refresh

A change arising from a *Periodic Refresh* (Chapter 10) is often marked with a One-Second Spotlight.

Timeout

A *Timeout* (Chapter 17) event, and any prior warnings, can be marked with a One-Second Spotlight. In community sites that show details about other users, the effect can also be used to mark changes to users' online status.

Metaphor

The pattern's name is the key here: a spotlight can be used to direct audience attention.

Acknowledgments

37signals introduced the Yellow Fade Technique into its products, and its Yellow Fade Technique article (<http://www.37signals.com/svn/archives/000558.php>) helped spread the meme.

One-Second Mutation



Auto-Update, Sync, Synchronize, Real-Time

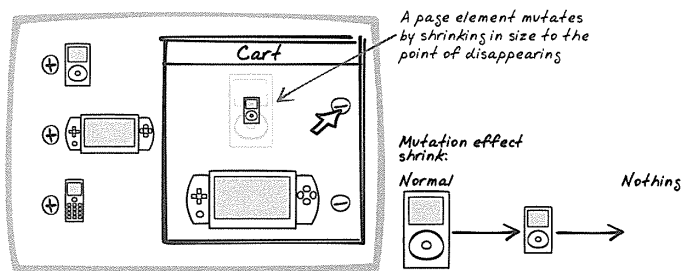


FIGURE 16-4. One-Second Mutation

Goal Story

Doc monitors a patient's condition, watching some histograms shrink and grow to reflect changes in the patient's body.

Problem

There's a lot of activity in an Ajax App; how can you help the user digest it all?

Forces

- To ensure that the user is working with current data, the browser display must be frequently updated.
- With Ajax, it's possible to update only the portion of the screen that has changed; if nothing has changed, the display remains exactly as is.
- The screen can get cluttered with a lot of information, much of which could change at any time.
- While human vision is good at spotting changes, it's easy to miss a sudden change, especially if the change is a subtle one.
- There are different types of changes, and those differences are important to users.

Solution

When a page element undergoes a value change or some other significant event, dynamically mutate its shape for about a second. In most cases, the mutation is used to reflect a permanent change. Instead of flipping from one form to another, the element gradually transitions into another form. The effect serves a few purposes:

- The mutation and the extended transition time make it likely that the user will notice that something has happened.
- The type of mutation hints at what sort of action has taken place.
- The gradual shift in appearance feels more natural than an instant transformation.

Mutation combines *Display Morphing* with *Scheduling*. A loop runs while the second of mutation progresses, with each transition occurring perhaps once every 50 milliseconds. Each transition will likely involve a change to at least one of the following CSS properties:

width

An object can be thinned or fattened by changing its width value.

height

An object can be heightened or shortened by changing its height property.

backgroundImage

Instead of just resizing the element, it's possible to swap in a completely different image. Just like a movie, the DOM element quickly flips through various images to give the illusion of mutation. See the DHTML Lemmings walkthrough in *Sprite* (Chapter 15) for more details.

position

Manipulating position with *position*, *left*, *right*, *top*, and *bottom* is possible for certain mutation effects.

overflow

Overflow can be set to *hidden* to enable the entire element to be present on the page while showing only a certain portion of it.

The first category of mutation involves making an object appear. The following are effective ways to communicate that the object has just been created or retrieved from the server.

Form

An item “forms” from nothingness into its standard appearance—typically, by incremental, outward growth. It might grow from the center of the object, a corner, or some other position entirely. Typically, width and height increase at the same time.

Slide Out

An item slides out. If the item slides in from the side, it appears to be pushed out like a sliding door. If the item comes in from the top, it appears like a garage door closing. If coming in from the bottom, the item appears to rise from the ground. Usually, either

width or height is fixed, while the other dimension gradually increases. The item's position changes simultaneously.

Reveal

An item is gradually revealed. It appears that the item has been there all along, and that a cover is gradually being lifted off. Here, one or both of width and height incrementally increases, while `overflow` is set to `hidden` so that part of the element is covered.

The next category of mutation involves some common disappearing acts—the opposite of the effects above—and a special effect for disappearing:

Disappear

An item changes from its standard appearance to nothingness.

Slide In

An item slides in from the top, side, or bottom.

Cover Up

An item is gradually covered up in one or both dimensions.

Blow Up

An item expands outward from its current side and simultaneously fades to nothingness.

There are also a few mutations that involve changing from one thing to another:

Grow

An item grows from one size to another. Height or width (or both) increases.

Shrink

An item shrinks from one size to another. Height or width (or both) decreases.

Metamorphise

An item shifts from one appearance to another. This is typically achieved with a change in background image. See *Sprite* (Chapter 15) for more details.

Most of these effects can be combined with *One-Second Spotlight* to give a 3-D effect. For example, an item can *Materialize* and *Form* at the same time. The human visual system uses both ambience and size as cues for distance, so the effect is that the item is reaching outward from behind the screen toward the user.

Note that you probably won't have to hand code effects, because libraries like *Scriptaculous* (discussed in the following "Real-World Examples") are quite flexible and are usually easy to incorporate into typical scripts.

Decisions

Most of the decisions in *One-Second Spotlight* are relevant here too. As with *One-Second Spotlight*, the "One-Second" part of the name is only a guideline.

Real-World Examples

TiddlyWiki

Jeremy Ruston's TiddlyWiki (<http://tiddlywiki.com>), like many of its spin-offs, uses a Grow effect to introduce new content when you click on a *Microlink* (Chapter 15). The Microlink itself is the starting point. The new content block appears to leap out and, as it flows outward, grows into a full *Malleable Content* (Chapter 15) block (Figure 16-5). Actually, there are a few visual effects at work here:

- *One-Second Spotlight* (see earlier in this chapter) is used to fade in the *Malleable Content* being opened, in parallel to the element leaping out.
- *One-Second Mutation* is used to grow the transition element from its initial state as a small piece of link text into a full block of Malleable Content.
- *One-Second Motion* (see later in this chapter) is used to move the element from the *Microlink* position into the Malleable Content below.

There are a few possible variations depending on whether the content is already open and where it's placed.

Scriptaculous library

As mentioned in *One-Second Spotlight*, Scriptaculous is a general-purpose JavaScript library. Its visual effects demo (<http://script.aculo.us/visual-effects>) provides many of the mutation effects described here.

DHTML Lemmings

DHTML Lemmings is a full-featured implementation of the Lemmings PC game, which uses DOM manipulation to show the lemming creatures performing typical lemming-like activities, such as walking, digging, and clutching an umbrella to make a safe descent from the top of a cliff. As the characters move about, their appearance is animated using quick mutations. While the usage is somewhat different from the usual type of mutation, which tends to focus on highlighting particular pieces of information, it is still of particular interest because it involves manipulation of images instead of just CSS style. See the code walkthrough in *Sprite* (Chapter 15) for more details.

Code Example: TiddlyWiki

Let's look at how TiddlyWiki shows the Grow effect discussed earlier. In doing so, we'll see how its generic animation engine works.

TiddlyWiki delegates animation to an Animator, which is capable of animating according to a list of Strategy (Gamma et al., 1995) objects, or animations. Each encapsulates the strategy for a particular animation. At this time, there are just a couple of animation objects: a Zoomer and a Slider.

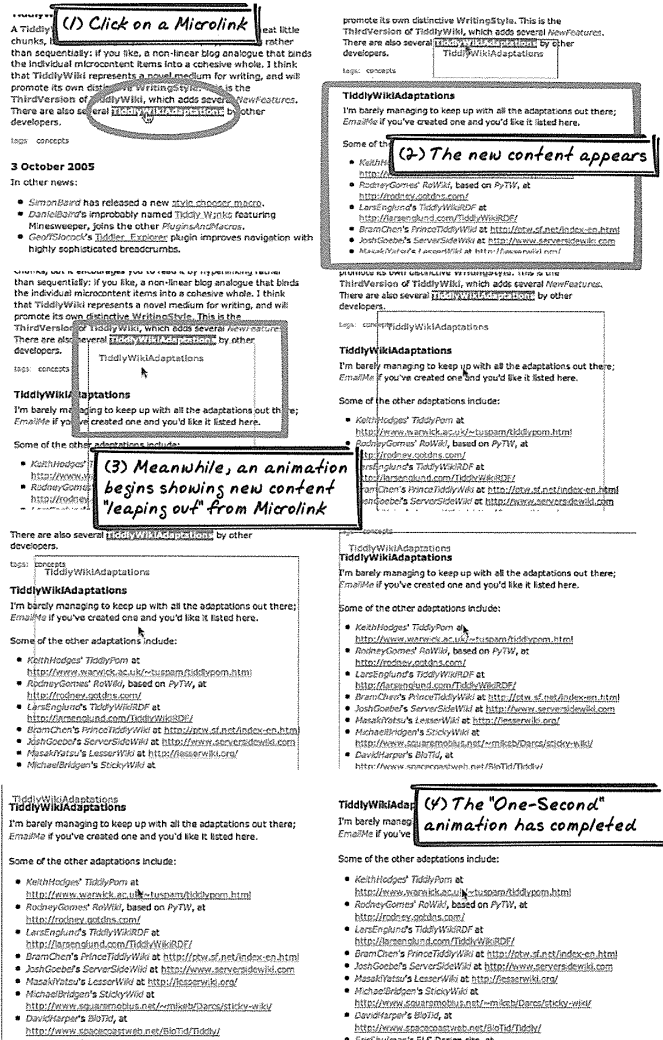


FIGURE 16-5. TiddlyWiki motion-mutation sequence

When a *Microlink* is clicked, the Animator is passed a Zoomer strategy, which is injected with the Microlink DOM object from which the content will leap out (src), a string title of the *Malleable Content* (title), the Malleable Content element in which the content will end up (theTiddler), and a flag indicating if the animation should be slow or not (I assume this is for debugging, and you can activate it by holding down Shift or Alt as you click on the link):

```
anim.startAnimating(new Zoomer(title,src,theTiddler,slowly));
```

startAnimating() pushes the new animation onto the list of running animations. If it's not already running, it sets up an animation loop:

```
Animator.prototype.startAnimating = function(animation)
{
  this.animations.push(animation);
  if(this.running++ == 0)
  {
    var me = this;
    this.timerID = window.setInterval(function() {me.doAnimate(me);},25);
  }
}
```

The loop runs every 25 milliseconds. It calls a tick() function on each running animation. By analyzing Zoomer's tick(), you can see how the One-Second Mutation is achieved. Upon construction, Zoomer has already set up a bunch of variables to support this function, as well as an element (element) to be shown during the transition. The variables hold position information about the start and end elements. tick(), then, is left with the task of interpolation: "if the item starts at point A and ends at point B, how should it look at a particular point in the journey?" The Animator provides a value f between 0 and 1 to define how far along the animation the object is, which makes things much easier. So, if the value is 0.5, the item's width will be halfway between the original width and the item's final width (Figure 16-6). The same idea applies for height and position:

```
this.element.style.left =
  this.startLeft + (this.targetLeft-this.startLeft) * f + "px";
this.element.style.top =
  this.startTop + (this.targetTop-this.startTop) * f + "px";
this.element.style.width =
  this.startWidth + (this.targetWidth-this.startWidth) * f + "px";
this.element.style.height =
  this.startHeight + (this.targetHeight-this.startHeight) * f + "px";
```

A *One-Second Spotlight* effect is also used here. The target object—the place the object is "leaping" toward—gradually fades in by way of the opacity property (and filter, for compatibility). Finally, the window is scrolled to show the target object:

```
this.targetElement.style.opacity = this.progress;
this.targetElement.style.filter = "alpha(opacity:" + this.progress * 100 + ")";
window.scrollTo(0,this.startScroll + (this.targetScroll-this.startScroll) * f);
```

The purpose of mutations is generally to provide a smooth transition from one state to another. Therefore, the final state should generally be reached simply by running the animation for the right period. No special handling needs to take place at the end, right?

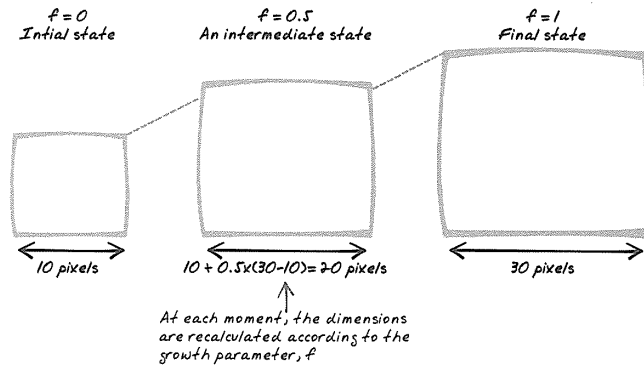


FIGURE 16-6. Growth factor

Well, in practice, there are rounding errors and approximations that make it good practice to explicitly set the final state. Thus, the Zoomer completes the animation by ensuring that the precise opacity is precisely 1 (in fact, the filter should ideally be set here too as a further security against rounding errors). The function also deletes the transient element it created to leap from source to target:

```
Zoomer.prototype.stop = function()
{
    this.element.parentNode.removeChild(this.element);
    this.targetElement.style.opacity = 1;
    window.scrollTo(0, this.targetScroll);
}
```

Alternatives

One-Second Spotlight and One-Second Motion

One-Second Spotlight (see earlier) and *One-Second Motion* (see the next pattern) are also used to draw attention to a significant event. One-Second Mutation is particularly suited for indicating that certain types of changes, such as object creation and removal, have occurred. It is not as well suited for general-purpose attention grabbing.

Related Patterns

Sprite

Sprites (Chapter 15) are often changed in rapid succession to give the impression of a smooth transition.

Metaphor

Many of the individual effects have their own metaphors—that's why they're used! Slide Out looks like a physical object sliding outward, Grow looks like an object growing, and so on.

Acknowledgments

Many of the effects discussed in the preceding Solution are based on an analysis of the Scriptaculous implementation (<http://script.aculo.us>). Jeremy Ruston's TiddlyWiki code was also instructive, and Jeremy helped clarify part of the code.

One-Second Motion



Copy, Displace, Duplicate, Jump, Leap, Move, Movement, Motion, Rearrange, Transfer

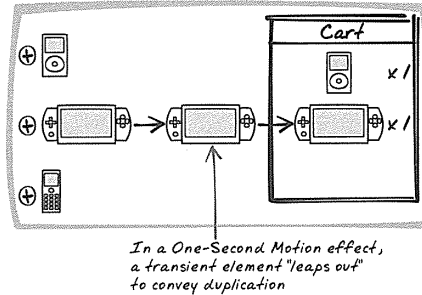


FIGURE 16-7. One Second Motion

Goal Story

Bill is adding a new monthly bank order. The existing orders are shown in the usual way—as a table with one row per order, sorted by cost. Once he is verified the new order form, it morphs to become a table row, then slowly drifts down the table. It soon reaches its rightful place in the sorted table, which is ordered by amount, and comes to rest after bouncing up and down a little due to reverberation.

Problem

How can you communicate that an element is shifting from one position to another?

Forces

- An element's location is important, as it can provide cues about its prominence and its relationship to other elements.
- An element sometimes needs to move from one place to another.
- Similarly, a new element sometimes needs to be created and then move from where it was created to another location.
- When an element changes position, the user often needs to know about it.

Solution

Incrementally move an element from one location to another, or temporarily move it, to communicate that an event has occurred. In most cases, the motion of the element involves a permanent change in its position. It can also be used to show a temporary displacement or gesture, such as a vibration effect.

There are several benefits of showing the motion explicitly:

- The motion and the extended transition time make it likely that the user will notice that something has happened.
- The motion will highlight the place where the element is moving to (it won't get lost on the page).
- The motion will feel more natural than an abrupt change.

The effect combines *Page Rearrangement* (Chapter 5) with *Scheduling* (Chapter 7). A loop runs for the duration of progresses, with each transition occurring perhaps once every 50 milliseconds. Each transition will likely involve a gradual change to the positioning style settings, usually to top and left. The position style can be set as absolute or relative, depending on the overall application design.

When an element has to move from point A to point B, in most cases it's logical for the trajectory to be the shortest distance from A to B, i.e., for the element to move as the crow flies. The algorithm in this case uses interpolation—for example, one-quarter of the time into the effect, an element will be one-quarter of the way between A and B—so, when moving from left to right, the left style of the element will be $(0.25 * x(A) + 0.75 * x(B))$, where $x()$ represents the source and destinations' horizontal coordinate.

This assumes a linear interpolation. You can also experiment with other algorithms, e.g., you might like to accelerate the element as it proceeds, which is often a more realistic model.

Note that you probably won't have to hand code motion, because libraries like Scriptaculous (discussed in the "Real-World Examples") are quite flexible and usually easy to incorporate into typical scripts.

Decisions

What events will trigger a motion effect?

One-Second Motion is not very common (yet), but there are several potential applications. First, a permanent movement can show any of the following:

- A straightforward change from one place to another.
- One element being spawned from another, and then jumping out to a new location.
- An element being introduced into a space. Instead of making the element just appear, or fade in, you could make it arrive from a logical starting point. For example, a new data point in a chart can "fly in" from the corner.

A temporary movement has its own applications:

Attention-grabbing

For general attention grabbing, motion is a killer app—humans are wired for sensitive motion detection as a means of survival. We’ve all seen enough annoying banner ads to know that. Extreme cases involve rapid vibration and rapid circuitous movement, such as tracing out a square 10 times a second. Of course, these extremes are rarely justified, and you can often use more subtle approaches to get the same effect (while causing a lot less annoyance).

Aesthetic effect

People expect objects to move in certain ways based on their experiences in the physical world as well as learned expectations from conventions in movies, cartoons, and video games. For example, when a small object bumps into a big object, you expect the small object to fly off in the other direction and the big object to shake a little.

Gesture

Some forms of motion rely on anthropomorphization—endowing the element with human-like characteristics—something users are surprisingly prone to do (“The computer doesn’t like me today!”). In the Scriptaculous demo (discussed in this pattern’s Code Example), the search form (<http://script.aculo.us/visual-effects>) shakes left and right when a validation error occurs, similar to a person shaking her head from side to side. Another effect would be to depict desire by showing an element temporarily “reaching” in a certain direction and pulling back again. This could be used as a hint to the novice user that he needs to drag the element in that direction.

Since motion can be distracting, you will likely want to limit its usage to the following situations:

- When the event is important enough that the user would probably appreciate having her attention diverted to it.
- When the user is likely to be focused on the motion already. Here, the motion is not used to attract attention, but rather to convey what’s happening to the element.
- When visualization and motion are central to the application style; for example, if the application was tracking a car’s movement every few seconds, it would probably be worthwhile to animate the move between sample points.

Also consider whether you should use motion at all. The conventional wisdom is that motion is more natural as a transition because it’s based on the physical world. But how about the massive population that grew up with computers and might be perfectly comfortable with the sudden jumps that are only possible in a digital realm? Maybe not everyone will appreciate motion effects.

What will the element do when it reaches its destination?

In every example I've seen, the element stops suddenly at its destination. However, it might appear more natural if the element reverberates when stacked against another element.

Real-World Examples

Scriptaculous library

As mentioned in *One-Second Spotlight* (see earlier in this chapter), Scriptaculous is a general-purpose JavaScript library. The visual effects demo (<http://script.aculo.us/visual-effects>) supports motion-based effects.

TiddlyWiki

As detailed in *One-Second Mutation* (see earlier in this chapter), clicking on a TiddlyWiki *Microlink* (Chapter 15; see <http://tiddlywiki.com>) causes the entire content to “leap out” from the link and form in a new position.

Backbase Portal

The Backbase Portal Demo (<http://projects.backbase.com/RUI/portal.html>) contains a page full of *Portlets* (Chapter 15) arranged in three columns, each with its own menu. The menu contains all of the Portlets on the page, and choosing a Portlet causes it to move over to the top of the column. *One-Second Motion* is used to animate the Portlet's movement, allowing users to see that the Portlet has “flown in” to the top of a column from elsewhere on the page.

DHTML Lemmings

As detailed in *Sprite* (Chapter 15), DHTML Lemmings involves the movement of Lemming creatures through a game space. Each Lemming is represented as a DOM element.

MS-Windows

MS-Windows uses a form of One-Second Motion. When minimizing a window, the whole window appears to leap downward to its handle in the taskbar, like a large object being packed into a small container. The reverse process occurs upon activating the window. This is not an Ajax App, but is still significant, as it's frequently cited as a rationale for the kind of animation this pattern details.

Code Example: Scriptaculous Effects

Motion is one of the effects offered by the Scriptaculous effects library (<http://script.aculo.us/visual-effects>). Examining its implementation gives us an opportunity to look at the overall Scriptaculous effects engine, which also provides *One-Second Spotlight*– and *One-Second Mutation*–style effects.*

* Note that the engine code looks a little different to most JavaScript applications because it uses Prototype (<http://prototype.conio.net/>) to allow for a more object-oriented coding style.

The generic engine component controls the flow by continuously delegating to an individual effect object to perform an incremental animation. This is an example of the Strategy pattern (Gamma et al., 1995). The initialization sequence clears the frame count, computes the total effect time, and kicks off the loop:

```
start: function(options) {
  ...
  this.currentFrame = 0;
  this.startOn      = new Date().getTime();
  this.finishOn     = this.startOn + (this.options.duration*1000);
  ...
  if(!this.options.sync) this.loop();
}
```

The loop runs until the predicted finish time, at which point the animation cleans up and terminates. The most important thing here is the pos calculation, which determines how far along the animation is. The calculation is essentially timeSoFar/timeRemaining. So, one-quarter of the way through the animation, it will be 0.25. Next, a calculation takes place to support the fps option, which lets the caller state the maximum number of animation frames per second. If an animation is indeed required, the engine's render() function is called:

```
loop: function() {
  var timePos = new Date().getTime();
  if(timePos >= this.finishOn) {
    this.render(1.0);
    ...
    return;
  }
  var pos = (timePos - this.startOn) / (this.finishOn - this.startOn);
  frame = Math.round(pos * this.options.fps * this.options.duration);
  if(frame > this.currentFrame) {
    this.render(pos);
    this.currentFrame = frame;
  }
  this.timeout = setTimeout(this.loop.bind(this), 10);
},
```

The main purpose of render() is to delegate to the effect strategy to perform the update. The strategy will receive a value between 0 and 1 to tell it how far the animation has proceeded.

The MoveBy effect takes a DOM element and tracks its start and end positions. The element's style is set to relative using a call to the effect engine's makePositioned() function. Because positioning is relative, you construct a MoveBy effect with relative arguments. To move an object 5 right and 10 down, you'd pass in 5 and 10 as parameters. The effect is to increase left by 5 and top by 10:

```
initialize: function(element, toTop, toLeft) {
  this.originalTop = parseFloat(this.element.style.top || '0');
  this.originalLeft = parseFloat(this.element.style.left || '0');
  this.toTop      = toTop;
  this.toLeft     = toLeft;
```

```

    Element.makePositioned(this.element);
    ...
}

```

Remember that the effects engine delegates to `update()` for effect-specific behavior, passing in a progress ratio between 0 and 1. In the case of `MoveBy`, the algorithm performs the necessary interpolation calculation to see how far along the object should be (`topd` and `leftd`). Having made the calculation, all that remains is to update the DOM element's style:

```

update: function(position) {
    topd = this.toTop * position + this.originalTop;
    leftd = this.toLeft * position + this.originalLeft;
    this.setPosition(topd, leftd);
},
setPosition: function(topd, leftd) {
    this.element.style.top = topd + "px";
    this.element.style.left = leftd + "px";
}

```

The `MoveBy` effect in itself is useful for getting an object from A to B. But you can build on it to create effects such as motion displacements. One such effect, included in the Scriptaculous library, is `Shake()`, which swings an element left and right a few times. With the framework in place, the effect is easily defined as a sequence of moves:

```

Effect.Shake = function(element) {
    return new Effect.MoveBy(element, 0, 20,
        { duration: 0.05, afterFinish: function(effect) {
            new Effect.MoveBy(effect.element, 0, -40,
                { duration: 0.1, afterFinish: function(effect) {
                    new Effect.MoveBy(effect.element, 0, 40,
                        { duration: 0.1, afterFinish: function(effect) {
                            ...
                        }
                    }
                }
            )
        }
        }
    );
}

```

Alternatives

One-Second Spotlight and One-Second Mutation

One-Second Spotlight and *One-Second Mutation* (see both earlier in this chapter) are also used to draw attention to a significant event. *One-Second Motion* is suited for indicating that an object's state has changed in the case where there is some geometric mapping to each object's state. When used as a temporary displacement effect, *One-Second Motion* is sometimes an alternative to these other patterns.

Related Patterns

Sprite

A *Sprite* (Chapter 15) often undergoes motion similar to *One-Second Motion* and can make use of similar interpolation calculations.

Guesstimate

In some cases, the motion is a form of *Guesstimate* (Chapter 13). When an object is moving around according to state information, the motion effect is effectively an estimate of what's happening between positions.

Metaphor

Until “Beam Me Up” teleportation technology hits the markets, every visible movement in the physical world is an illustration of this pattern.

Highlight



Auto-Update, Sync, Synchronize, Real-Time

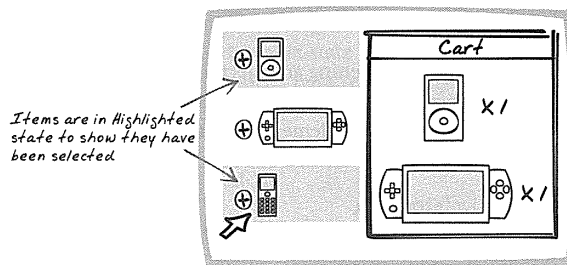


FIGURE 16-8. Metaphor

Goal Story

Sasha notices that a few spam comments have appeared on an old blog posting. When she clicks on the comments, they are successively Highlighted—the background color switches from white to yellow. Because she’s already logged in, there’s a Delete Spam button below the comments, which she clicks to send them where they belong.

Problem

How can you make elements stand out?

Forces

- An Ajax App can enable a lot of information to be shown at once—often information with many with different states.
- There is usually one element which the user is working on or about to activate.
- In addition, the user often needs to group several elements together to perform a common function. These elements must stand out from the crowd.

Solution

Highlight elements by rendering them in a consistent, attention-grabbing format. This pattern has been applied to dynamic web sites prior to Ajax but is particularly important in the context of rich displays and interaction.

Consistency is important here; when items are selected, they should look the same. One straightforward way to achieve this is with selected and deselected CSS classes. By dynamically switching the element's `className` property, you can easily Highlight and de-Highlight.

Highlighting is useful in the following ways:

- To show which particular element has input focus.
- To show which elements are selected.
- When the user rolls over an element, to indicate its boundaries and hint that some action will occur by clicking on it.
- To indicate that an element is particularly important.
- To indicate that an element is undergoing change.
- To prompt the user to perform some kind of work on the element.

Decisions

How will the Highlight look?

You want the Highlight to be noticeable but not distracting. Here are some guidelines for deciding how the Highlight should look:

- Tone down the Highlight if it's likely to occupy a large proportion of the page. This would happen if there are quite a few Highlighted elements or if each element is relatively large.
- Users often need to read and edit Highlighted elements, so ensure that the display is usable whether or not the element is Highlighted.
- Avoid Highlighting techniques that displace other elements on the page. For instance, increasing font size might increase the element's size, in turn pushing away other elements. The Highlight should ideally appear as an on-off toggle, only affecting the element in question.

How will the Highlight appear and disappear?

The most obvious thing to do is make Highlight state binary—an element is either Highlighted or it's not. But consider using visual effects to make the transition smoother and support visual pattern recognition (see the following "Teacher!" example).

Real-World Examples

Gmail

Gmail presents a list of mail messages one row at a time. You select messages by clicking on the checkbox, which causes the entire row to change to a light tan background color. The appearance of various tan-colored rows makes it easy to spot selected items, and the checkboxes also provide a clue.

A9

A9 provides search results in different “Columns.” Visit the Preferences page, then request to add more Columns. If you have an account, you’re allowed to choose multiple Columns at once. Each Column is shown as a horizontal block in the Preferences page, and when you click the Add button, the block Highlights to confirm your choice. The interface uses Highlighting effectively, but it would be even more helpful if *all* added elements remained Highlighted instead of just the most recent one.

Teacher!

Teacher! (<http://teacherly.com/>) shows a table of student grades and Highlights whichever row your mouse is hovering over. What’s novel here is that when you move to another row, the Highlight doesn’t just disappear, but slowly fades away (*One-Second Spotlight*). An AjaxPatterns demo replicates the effect (<http://ajaxify.com/run/time/periodicRefresh/spotlightCustom/highlight/>).

Whitespace

Whitespace (<http9rules.com/whitespace/>) prominently Highlights links on the side menu as you roll the mouse over them. This is a common technique used on many web sites and can be achieved entirely by using CSS styling to modify the anchor tag, absent of any JavaScript.

Code Example: AjaxPatterns Wiki

The Wiki demo (<http://ajaxify.com/run/wiki>) uses Highlighting in a couple of ways:

- As the user rolls the mouse over a message and then clicks on it, the message is Highlighted to indicate its current status (Figure 16-9).
- When the message has changed and is being buffered for submission, the background color changes. Once it has been submitted, the color returns to normal.

The first example is detailed in *Malleable Content* (Chapter 15), in which I explain the buffering Highlight. When you click outside the message area, the application checks whether you changed any text from the version originally downloaded, and if so, that text is Highlighted and prepared for uploading:

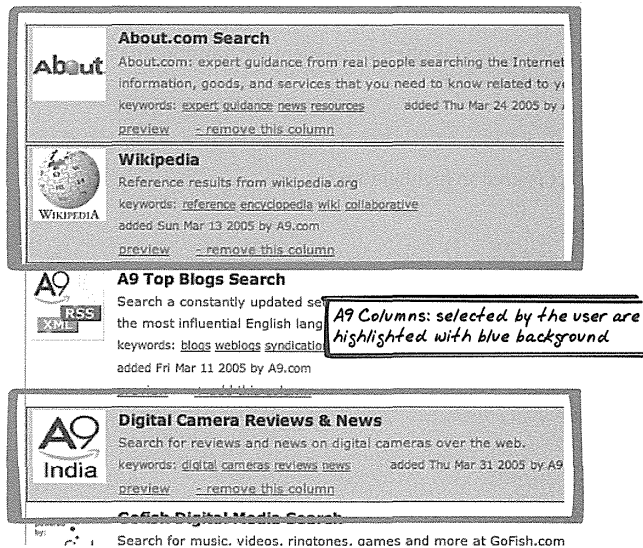


FIGURE 16-9. A9 columns

```
function onMessageBlur(event) {
    ...
    var initialMessageContent = message.serverMessage.content;
    if (message.value != initialMessageContent) {
        pendingMessages[message.id] = true;
        message.style.backgroundColor = "#ffcc33";
    }
    startPeriodicSync();
}
```

The sync is started again, so the item will soon be uploaded. If any further edits begin in the next few seconds, the upload will be placed on hold and the second message alteration will also be Highlighted. At any point in time, then, all of the buffered messages are Highlighted. When the messages are eventually uploaded, they revert to their normal, read-only color:

```
for (messageId in pendingMessages) {
    ...
    $(messageId).style.backgroundColor = "#cccccc";
}
```

Related Patterns

Status Area

The *Status Area* (Chapter 15) is a convenient place to explain why an element is Highlighted.

One-Second Spotlight

One-Second Spotlight (see earlier) is often convenient for transitioning into and out of longer-duration Highlight periods.

Metaphor

A fluorescent marker is used to highlight text.

Functionality

T HIS CHAPTER CONTAINS A NUMBER OF HIGHER-LEVEL PATTERNS THAT ARE MORE GEARED TOWARD offering new functionality than modifying the user interface. Technical barriers have prevented these from being used much on the Web, but Ajax technologies are helping to break down the barrier.

Lazy Registration addresses the issue of managing the lifecycle of a user profile. Instead of a binary “you’re registered or you’re not” perspective, *Lazy Registration* advocates a more gradual accumulation of user data and authentication credentials. A related pattern is *Direct Login*, which explains how to let the user securely log in without forcing a page refresh. It incorporates browser-side encryption, a theme at the heart of the next pattern, *Host-Proof Hosting*. The purpose of this pattern is to encrypt and decrypt data within the browser so that it can’t be inspected by the hosting organization.

The next two patterns address monitoring the user’s activity level. *Timeout* uses event-handling and scheduling to decide when the user is no longer active. *Heartbeat* is similar, but it brings the server into the equation; the browser keeps sending a heartbeat message to tell the server that the user is still active and that the application is still sitting in the browser.

Finally, *Unique URLs* restores some functionality that Ajax, to some degree, takes away. Bookmarkable URLs and browser history have always been an issue on the Web, but Ajax exacerbates the problem when developers choose not to use page refreshes. The *Unique URLs* pattern identifies a range of tricks to handle bookmarking, deep linking, the Back button, and related concerns.

Lazy Registration



Account, Authentication, Customisation, Customization, Incremental, Login, Password, Personalisation, Personalization, Profiling, Registration, User, Verification

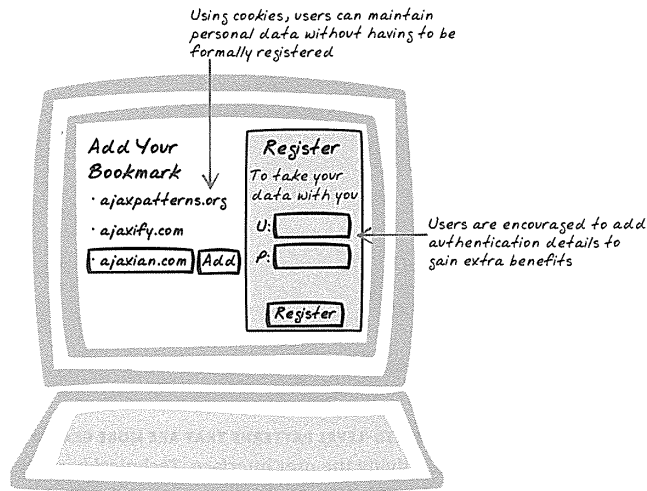


FIGURE 17-1. Lazy Registration

Goal Story

It's Saturday afternoon, and Stuart is busy planning the evening's activities. He visits a band listings web site, where he clicks on a map to zoom into his local area. Even though he has never visited the site before, a profile, which he can see on the top of the site is already being constructed. At this stage, the profile guesses at his location based on his actions so far. As he browses some of the jazz bands, the profile starts to show an increasing preference for jazz bands, and some of the ads reflect that. Since the jazz thing is a one-time idea, he goes into his profile and tweaks some of those genre preferences but leaves the location alone since the system's guess was correct. Finally, he decides to make a booking, at which point he establishes a password for future access to the same profile including his address, which is posted back to the profile.

Problem

How can the user customize the site while deferring formal registration?

Forces

- Public web sites thrive on registered users. Registered users receive personalized content, which means that the web site is able to deliver greater value per user. And registered users can also receive more focused advertising material.
- For nonpublic web sites, such as extranets used by external customers, registration may be a necessity.
- Most users don't like giving their personal information to a web server. They have concerns about their own privacy and the security of the information. Furthermore, the registration process is often time consuming.
- Many users spend time familiarizing themselves with a site before registering. In some cases, a user might interact with a site for years before formally establishing an account. There is a lot of valuable information that can be gained from this interaction that will benefit both the web site owner and the user.

Solution

Accumulate bits of information on the user as they interact while deferring formal registration. As soon as the user visits the web site, a user account with auto-generated ID is immediately created for her and set in a cookie that will remain in the browser. It doesn't matter if she never return; unused IDs can be cleared after a few months.

As the user interacts with the application, the account accumulates data. In many cases, the data is explicitly contributed by the user, and it's advisable to expose this kind of information so that the user can actually populate it. In this way, the initial profile may be seen as a structure with lots of holes. Some holes are eventually filled out automatically and others by the user himself. The user is also free to correct any of the filled-in data at any time (Figure 17-2).

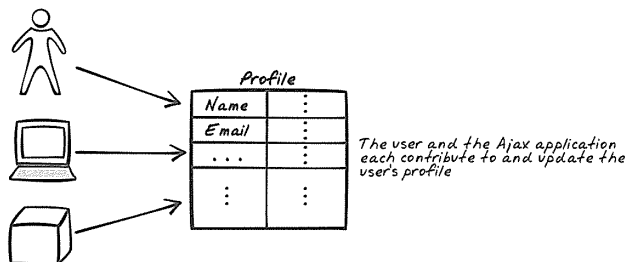


FIGURE 17-2. User Profile

Two particularly notable “holes” are a unique user identifier and a password. It is this combination of attributes that allows the user to access the profile from another machine or a different browser. They will also preserve the profile in the case that cookies are deleted from the user’s browser. So, while this pattern is generally about gradual accumulation of profile data, there remains a significant milestone in the user/application relationship—the moment at which user ID and password are established.

Do the user ID and password have to be provided simultaneously? No. Even that can be incremental as long as you make the email address the unique identifier. In fact, this is pretty common nowadays. Email is usually required anyway, and it’s unique, so why not make it the user ID? In the context of Lazy Registration, though, there’s an additional benefit, as the email might be accumulated in the natural flow of events—the site might add the user to an announcements list, for example. In some cases, the email might even be verified during this process.

Sceptics may wonder why a user would want to actively work with her profile. The answer was formulated in a web usability pattern called “Carrot and a Stick” (http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/Kane/perzel_kane.pdf):

Determine what users consider to be a “valuable” carrot. Offer the end user a portion of that carrot before you request personal information. The content is withheld (“the stick”) until the requested information is provided.

Thus, users will only enter information if there is a perceived benefit to them. There is plenty of evidence that this occurs—witness the social bookmarking phenomenon, where thousands of users make public their personal links. By exposing their profiles, many of those users are hoping the system will point them in the direction of related resources they have not yet heard of.

Some web sites have used this pattern for years, so what does it have to do with Ajax? *Lazy Registration* aims for a smooth approach in which the barrier is low for each new user contribution. For instance, you sign up for a web site’s mailing list, and your email is automatically added to your profile and shown on the side of the page. With Ajax, there’s no need to break the flow. No more “just go over there for a few minutes, then come back here, and if you’re lucky, you might be looking at something similar to what you can see now.” That’s a big win for web sites aiming to drop the barrier of registration, and it’s great for users, too.

It’s standard practice for web sites to collect data about users. The aim of this pattern is to empower them to contribute to this. Instead of covertly building up a corpus of data on a user, you invite him to add value to his own experience by contributing and maintaining the data himself.*

* The Attention Trust (<http://www.attentiontrust.org/about>) is an organization which promotes this idea.

Several technologies are involved in Lazy Registration:

Database

You clearly need a persistent data store in order to retain user profiles.

XMLHttpRequest

Passing profile information back and forth with *XMLHttpRequest Calls* is the key to the smooth interaction mode you are seeking to achieve with this pattern.

Cookie manipulation and session tracking

A cookie, associated with your domain and identifying a unique session ID, must reside in the user's browser. The session ID can serve as a key on the server side to locate details about the user each time she accesses the web site. In conventional applications, the cookie is pushed from the browser to the server as a header in the response. That's fine for Ajaxian Lazy Registration when the user first accesses the system, though sometimes it may be convenient to use a more Ajax-oriented approach. The first such approach is to manipulate the cookie in JavaScript (<http://www.netspade.com/articles/javascript/cookies.xml>). The second is to set the cookie using the response from an *XMLHttpRequest Call*, which all browsers are apparently happy to deal with in the same way as they deal with cookies in regular page reloads. In practice, you're unlikely to be playing with cookies anyway. Most modern environments contain session-tracking frameworks that do the low-level cookie manipulation for you (also see *Direct Login*). They generally use either URL rewriting or cookie manipulation, and you need the latter to make this pattern work most effectively. Since responses from *XMLHttpRequest* set cookies in the same way as do entire page reloads, you should be able to change session data while servicing *XMLHttpRequest Calls*.

Decisions

What kind of data will the profile contain?

Usability and functionality concerns will drive decisions on what data is accumulated. By envisioning how users will interact with the web site, you can decide what kind of data must be there to support the interaction. For example:

- Do you want to provide localized ads? You'll need to know where the user lives.
- Do you want to use collaborative filtering? You'll need to capture the user's preferences.

In addition, consider that some users, such as employees working on an intranet web site, use certain Ajax Apps all day long. For that reason, the profile might also contain preferences similar to those on conventional desktop applications. Many options will be application-specific, but a few generic examples include:

- Enabling and disabling visual effects, such as *One-Second Spotlight* (Chapter 16).

- Setting *Heartbeat*-related (see earlier in this chapter) parameters, e.g., setting how long timeout will be and whether the system will prompt the user when it's coming up.
- Customizing *Status Area* (Chapter 15) display.

One issue that arises with Lazy Registration is the clearing of data. What if a user visits once and never comes back? You probably don't want to keep that data sitting there forever. Typically, you will probably have a script running daily to delete (or archive) the records of users whose last login was, say, three months ago.

How can the profile be accumulated?

You might know what data you need, but are users willing to give it to you? This comes back to the carrot-and-stick argument: you need to provide users a service that will make it worthwhile for them to provide that data. In addition, you need to communicate the benefit, and you must be able to assure them that the data will be safe and secure.

The least imaginative way to gain user data is to pay them for it, or, more deviously, pay others for it. Giving away a T-shirt in exchange for data was fine during the dot-com boom, but hopefully you can do better than that. Give the user a service they really need. For example:

- If you want the user to provide his email, offer to send email notifications.
- If you want the user to provide an ID and password, help him understand the benefits: he can log in from anywhere and the data will survive a hard-drive crash.
- If you want the user to provide his physical address, provide localized search features.
- If you want the user to rate your product, provide recommendations based on his ratings.

How much data should be stored in cookies?

How much you store in cookies depends on your general approach to the Ajax implementation: is the application browser-centric or server-centric? A browser-centric choice would be to pack as much as possible into the browser's local state so as to optimize performance, while running a full-fledged JavaScript application with a little server-side synchronization. A server-centric approach would rely only on data held server-side, with the browser accessing additional data on a need-to-know basis using *XMLHttpRequest Calls*.

One special concern is the security of cookies. If users access the application from a public PC, there's the risk of unauthorized access. In this case, it's especially advisable not to store sensitive information in the browser and to offer the possibility of cleaning cookies at the end of the session. (For instance, call the option "I'm on a public terminal.")

Real-World Examples

Memeflow

Steve Lacey's MemeFlow (<http://memeflow.com>) is a portal with RSS-backed *Portlets*. Its use of Lazy Registration is characteristic of several other portals (Figure 17-3). You can immediately build up a collection of your favorite feeds, and when you provide your username and password later on, those feeds will remain.

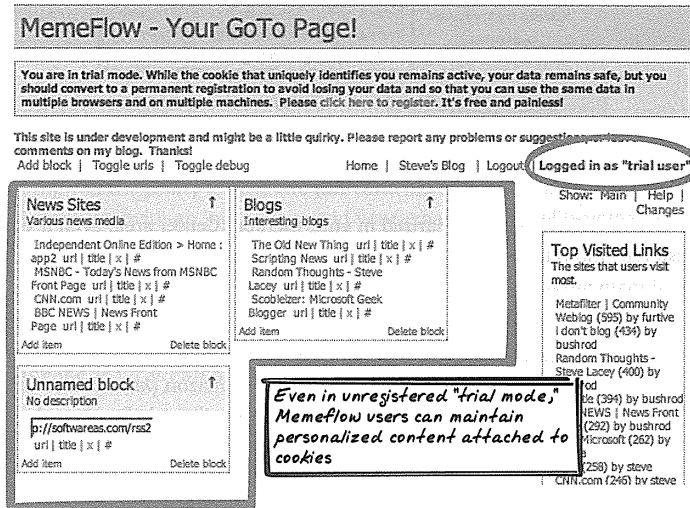


FIGURE 17-3. Memeflow

Blummy

Alexander Kirk's Blummy (<http://blummy.com>) is a bookmarklet manager (Figure 17-4). You can start adding bookmarklets to a personal "Blummy" container straightaway (this has a unique URL). When you register, you'll get a URL with your own name, but the old URL remains valid, so you can keep the bookmark you created before registering.

Kayak

Kayak (<http://kayak.com>) is a travel search engine that retains queries you've made. A query history is available for nonregistered users and becomes part of your profile once registered.

Palmsphere

Palmsphere (<http://palmsphere.com/store/home>) showcases Palm applications for download and purchase. Each item has a Favorite button—if checked, the item is one of your

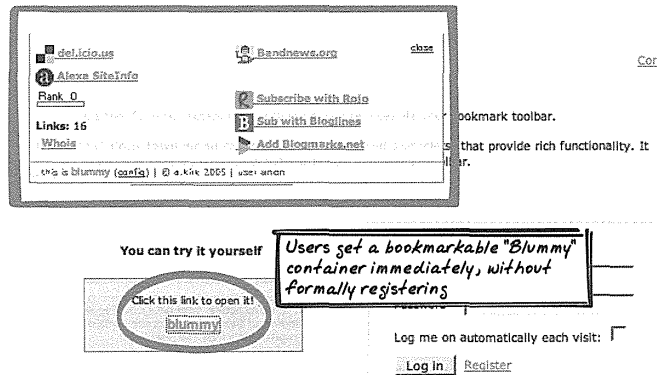


FIGURE 17-4. Blummy

Favorites. The Favorites list is summarized in your Member Center area, even if you've never registered, and retained in a cookie for the next time you visit.

Amazon.com

Amazon (<http://amazon.com>) has begun incorporating Ajax features only recently, but it blazed the trail for Lazy Registration a long time ago. Visit Amazon (<http://amazon.com>) as a new user, browse for just a few seconds, and here's what you'll see before even beginning to register or log in:

- Shopping cart, to which you can add items
- Recently Viewed Items
- Page You Made—showing recent views and bookmarks related to those
- Ability to update your history by deleting items you viewed
- Ability to turn off Page You Made
- Ability to search for a friend's public wish list

Code Example: AjaxPatterns Shop Demo

The Ajax Shop Demo (<http://ajaxify.com/shop>) illustrates the kind of user interface described by this pattern (Figure 17-5).

When you run the demo, you'll notice a few things:

- You can add items to your cart right away.
- The application guesses your favorite category by watching what you're looking at. If you prefer, you can override the application's guess, and the application will no longer attempt to adjust it.

Cart	Clear	Mail Contents
Am I Ever Gonna See Your Face Again?	1	
Hackers and Painters	1	
Being Digital	1	

Books [Explore](#)
[Add To Cart](#) *Accidental Empires*
[Add To Cart](#) *Being Digital*
[Add To Cart](#) *Hackers and Painters*

Songs [Explore](#)

Movies [Explore](#)

User Id:
37823
My Best Category:
Songs [▼](#)
Email:

Password:

[Login](#) [Register](#)

FIGURE 17-5. Ajax Shop Demo

- By offering you the ability to send the cart contents to your email address, the application provides you with an incentive to add your email to the profile without yet formally registering.
- The password and email verification process itself is unintrusive—the main flow of the site is uninterrupted. Even while you’re waiting for verification mail, you can continue to play around with the main content. Figure 17-6 shows the application when the user has gone as far as entering an email address and password.

Cart	Clear	Mail Contents
Am I Ever Gonna See Your Face Again?	1	
Hackers and Painters	1	
Being Digital	1	

Books [Explore](#)
[Add To Cart](#) *Accidental Empires*
[Add To Cart](#) *Being Digital*
[Add To Cart](#) *Hackers and Painters*

Songs [Explore](#)

Movies [Explore](#)

User Id:
37823
My Best Category:
Songs [▼](#)
Email:
michael@mahemoff.com
Password:

Demo Registration
1. Please ensure email address is correct, then verify your password below.
Verify Password:

[Send Email](#) [Cancel](#)

FIGURE 17-6. Ajax Shop Demo with email address and password entered

To keep things simple, it doesn’t actually use a persistent data store; all information is held in the session. That’s definitely not advisable for a real system, because you don’t want to store passwords and other sensitive data there. Also, it means that the user, in theory, could bypass the email verification by inspecting the cookie. Nevertheless, the application demonstrates Lazy Registration from the user’s perspective, and the underlying code provides some illustration of what’s required to develop such an application.

Following are some of the features and how they were achieved.

Retrieval of categories and items

The application maintains the flow by avoiding any page reloads when categories and items are accessed. No information about categories or items is hardcoded; generic REST services are used to extract the data and are rendered locally in JavaScript.

Cart management

Again, the only real relevance of cart management is that page reloads are avoided. The cart contents are tracked in the session, so they should be present when the user resumes using the web site. When the user adds something to the cart, the JavaScript cart is *not* directly altered. Instead, the new item is posted to the server as XML:

```
function onAddItemClicked(item) {
    var vars = {
        command: 'add',
        item: item
    }
    ajaxCaller.postForXML("cart.phtml", vars, onCartResponse);
}
```

And likewise when the cart is cleared:

```
function onCartClearClicked() {
    var vars = {
        command: 'clear'
    }
    ajaxCaller.postForXML("cart.phtml", vars, onCartResponse);
}
```

For both operations, the server retrieves the session cart and alters its state:

```
$cart = $_SESSION['cart'];

if (isset($_POST["command"]) && $_POST["command"]=="add") {
    $item = $_POST["item"];
    $cart->add($item);
} else if (isset($_POST["command"]) && $_POST["command"]=="clear") {
    $cart->clear();
}
```

Then, the server outputs the final state as an XML response:

```
header("Content-type: text/xml");
echo "<cart>";
$content = $cart->getContents();
foreach (array_keys($content) as $itemName) {
    echo "<item>";
    echo "<name>$itemName</name>";
    echo "<amount>".$content[$itemName]."</amount>";
    echo "</item>";
}
echo "</cart>";
```

In the browser, onCartResponse is registered to render the cart based on the resulting XML.

Mailing cart contents

The profile block contains, along with several other fields, the user's email. There's also a clickable Mail field on the cart:

```
<div>
  <div class="userLabel">Email:</div>
  <input type="text" id="email" name="email" />
</div>
...
<span id="cartMail">Mail Contents</span>
....
$("cartMail").onclick = onCartMailClicked;
```

When the user clicks on cartMail, the server checks that the email has been filled in and simply uploads a POST message for the mail to occur. In this case, there's no feedback to the web user, so the callback function is blank:

```
vars = {
  command: "mailCart",
  email: email
}
ajaxCaller.postForPlainText("cart.phtml", vars, function() {});
```

The server receives not only the command, but the email address itself, since this might not be in the user's profile yet. Just prior to sending the mail, the server retains the address as part of the user's session:

```
function mailCart() {
  ...
  $email = $_POST["email"];
  // Add mail to the profile - it's part of the Lazy Registration.
  $_SESSION['email'] = $email;
  ...
}
```

Then, it's a simple matter of constructing a message from the server-side cart state and sending the email to the specified address using standard server-side libraries.

Tracking favorite categories

There's a fixed "favorite category" selector in the HTML. It begins empty and is populated when the categories are loaded:

```
<div id="favoriteCategoryInfo">
  My &nbsp;&nbsp;Best &nbsp;&nbsp;Category: <select id="favoriteCategory"></select>
</div>

function onAllCategoriesResponse(xml, ignoredHeaders, ignoredContext) {
  ...
  categoryExplores[category] = 0;
```

* Again, this is a simplification, because retaining the address would not be necessary—or desirable—if the user was already logged in.

```

        favoriteCategoryOption = document.createElement("option");
        ...
    }

```

There's also a mode variable to indicate whether the favorite category selection is automated. It begins in automated mode:

```

var isFavoriteCategoryAutomated = true;

```

If in automated mode, the script watches each time the user explores an item. Each category is tracked according to how many times the item was explored, and the selector is altered if a new maximum is reached:

```

var categoryExplores = new Array();
...

function onExploreClicked(category) {
    ...
    if (isFavoriteCategoryAutomated) {
        categoryExplores[category]++;
        favoriteCategory = $("favoriteCategory").value;
        favoriteCategoryExplores = categoryExplores[favoriteCategory];
        if (categoryExplores[category] > favoriteCategoryExplores) {
            $("favoriteCategory").value = category;
        }
    }
}

```

If the user decides to overwrite this *Guesstimate* by manually setting the preference, it will stay manual permanently:

```

$("favoriteCategory").onclick = function() {
    isFavoriteCategoryAutomated = false;
}

```

For the sake of simplicity, this field is not actually tracked in the server, though it could easily be incorporated into the user's profile.

Verifying password and email

Now for the most important part. The user is finally willing to verify her password and email. These could potentially be broken into two separate verification activities, but since they fit together as a formal registration step, they are combined in the demo.

The trick is to manage the process with a little state transition logic. The registration is broken into a few states with transitions between them. Each state requires you to handle events in a slightly different way. Each transition involves altering the UI a little to reflect what the user can do.

registerState holds the current state:

```

/*
    "start": When page is loaded
    "mustSendMail": When instructions and verify password field shown
    "mustVerifySecretNumber": When email sent and user must enter secret

```

```

        number inside email
        "verified": When user is successfully logged in
    */

    var registerState = "start";

```

The HTML for this demo contains all the necessary fields and buttons. Their visibility is toggled based on the current state. For example, following are the password and password verification fields. The password field is always shown until the user is at the “verified” stage, whereas the “verify password” field is only shown after the user initiates the registration process:

```

<div id="passwordInfo">
  <div class="userLabel">Password:</div>
  <input type="password" id="password" name="password"/>
</div>

<div id="verifyPasswordInfo">
  <div id="regHeader">Demo Registration</div>
  <div class="regInstructions">
    <strong>1.</strong> Please ensure email address is correct and
    password is <strong>not</strong> confidential, then verify your
    password below.
  </div>
  <div class="userLabel">Verify Password:</div>
  <input id="verifyPassword" type="password" name="verifyPassword" />
</div>

```

All three buttons are declared and, again, their visibility will change depending on the current state:

```

<input type="button" id="login" value="Login"></button>
<input type="button" id="register" value="Register"></button>
<input type="button" id="cancel" value="Cancel"></button>

```

What’s most important here is the Register button, which drives the process through each state. The Cancel button returns the state back to start, which causes the display to return to its initial state too. The purpose of the Login button is purely for demonstration. The Register button is present until the user is verified, and its label changes at each stage of the registration process. Its event handler remains the same throughout; the handler decides what to do based on the current state:

```

function onRegisterClicked() {
  if (registerState=="start") {
    registerState = "mustSendMail";
  } else if (registerState=="mustSendMail") {
    var submissionOK = sendMail();
    if (submissionOK) {
      registerState = "mustVerifySecretNumber";
    } else {
      return;
    }
  }
}

```



```

    } else if (registerState=="mustVerifySecretNumber") {
        verifySecretNumber();
    }
    onRegistrationStateChanged();
}

```

And `onRegistrationStateChanged()` exists purely to reveal and hide fields, and to change the button label based on the current state:

```

function onRegistrationStateChanged() {
    if (registerState=="start") {
        $("userForm").reset();
        $("login").style.display = "inline";
        $("verifyPasswordInfo").style.display = "none";
        $("secretNumberInfo").style.display = "none";
        $("verifiedInfo").style.display="none";
        $("cancel").style.display = "none";
        $("register").value="Register";
    } else if (registerState=="mustSendMail") {
        ...
    } else if (registerState=="mustVerifySecretNumber") {
        ...
    } else if (registerState=="verified") {
        ...
    }
}

```

Related Patterns

Direct Login

Direct Login (see the next pattern) is a companion pattern, since some dynamic behavior can allow for login and registration to appear on the same form.

Live Form

It's useful to maintain the profile details in a *Live Form* (Chapter 14) so that the user can easily add to them and the server can synchronize state and provide opportunities for further enhancement to the profile.

Timeout

When data is held in cookies, it's important to expire the cookies if there's a risk that others may gain access to the browser. *Timeout* (see later) helps the server decide whether the client is still active. If it's not, it may be wise to ensure that any sensitive data is wiped from the cookies held in the browser.

Guesstimate

Lazy Registration can sometimes involve inferring information about the user's profile by monitoring his behavior. Thus, it embraces the same nebulous principles as *Guesstimate* (Chapter 13), where a guess is acknowledged to be imprecise but better than no guess at all.

Metaphor

A good salesperson works the same way. While assumptions might be made based on a prospect's behavior, the salesperson is always listening; her assumptions are always open to challenge. (Malcolm Gladwell depicted this pattern of successful salespeople in *Blink* [Little, Brown, 2005]).

Want to Know More?

- OECD privacy guidelines (http://www.oecd.org/document/18/0,2340,en_2649_201185_1815186_1_1_1_1,00.html)
- Personalization versus Customization (http://www.clickz.com/experts/archives/mkt/precis_mkt/article.php/814811)

Acknowledgments

The idea to handle Lazy Registration in this Ajaxian manner was originally proposed by Chris Were ("Tahpot") (<http://tahpot.blogspot.com/2005/06/lazy-registration-with-ajax.html>).

Direct Login



Access, Account, Authentication, Login

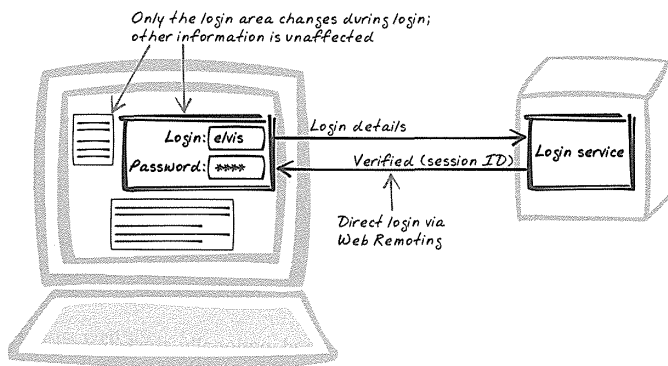


FIGURE 17-7. Direct Login

Goal Story

When Stuart logs in to perform an online exam, he is presented with a standard username-password form. He enters his username and password, but a few seconds later, the form becomes red and shows an error message underneath. He switches off Caps Lock and re-enters his credentials. This time, they're accepted. The form morphs to show his name and balance, and a new menu appears alongside it.

Problem

How can users present their credentials to the server?

Forces

- Login is a necessary evil—it should be as transparent as possible.
- Casual users may not bother to log in if the process interrupts their browsing experience.
- Login requires the browser to interact with the server in order to validate the username and password.
- The password should not travel in plain-text; it could be intercepted while traveling to the server.

Solution

Authenticate the user with an XMLHttpRequest Call instead of with a form-based submission, hashing in the browser for improved security. The essence of this pattern is a routine transformation from a submission-based approach to an Ajaxian, Web Remoting (Chapter 6) interaction style. But, in this pattern, I also discuss a very useful, though optional, technique that involves JavaScript-based hashing and is specific to the login process.

Conventional authentication usually requires the user and password to be uploaded as a standard form submission. The server usually converts the password to a hash (or “validator”) value and checks it against a stored hash value in the database. If they match, the user is in.

There are two problems with this approach. Firstly, flushing the page can be a distraction. It might not take long, but it will usually leave the user in a different context, which will discourage her from logging in. Even more troublesome is the stream of pages that ensue when a password must be recovered (e.g., having to provide your maiden name, last purchase date, and the name of your favorite pet canary), especially in this security-conscious era. The other problem relates to security of the transmission; if the password is uploaded as plain-text, there’s a risk of interception.

Direct Login addresses the page refresh problem and, optionally, the transmission problem too. In the simplest approach, you can implement *Direct Login* by simply sending the username and password in plain-text to the validation service using an XMLHttpRequest POST. Then, the server behaves similarly to a conventional server: it checks whether the password matches using a hash function and prepares for session management. XMLHttpRequest deals with cookies as it does regular form handling; this allows the session to be established in the same way as a conventional form submission. The only difference is the response content: instead of outputting a new HTML page, the server outputs an XML or plain-text acknowledgment, as well as any personalized content.

Passing the credentials over with XMLHttpRequest will improve usability, but as long as the password is being transferred in plain-text, there's still a security threat. Ensuring the whole transaction runs over HTTPS is always the best measure, as this generally makes the transaction secure from interception. However, many web sites don't provide such a facility. Fortunately, there's a compromise that can prevent transmission of plain-text passwords. The technique, strictly speaking, is orthogonal to the Direct Login approach; you could apply it to conventional submission-based authentication as well. But since it makes heavy use of browser-side processing, it fits nicely with Direct Login.

The trick is to perform hashing in the browser. JavaScript is fast enough to transform the password into a hash value (not to mention that it's capable of doing it), and there are libraries available to implement popular algorithms.

The naïve way of hashing is to simply hash the password to match what should be in the database. But any interceptor would then be able to perform a *replay attack*—it could log in using the same details. So we need a more sophisticated approach, such as the following *double hashing* algorithm.

With double hashing, the server generates a one-time random seed (S). The browser then hashes twice: first, it hashes the password (P), hopefully to yield what is stored on the database (Ha; Hash attempt). But instead of sending that, the browser combines it with the one-time seed to form a new hash (Da; Double-hash attempt). This new hash is sent to the server. The server then pulls out the stored hash (H) from the database and combines it with the original one-time seed (S) to form a new hash, which must match the hash (Da) that was uploaded. This works because, in both cases, the initial password (P) has been passed through the same two hash functions. In the browser, the user's attempt is passed through a fixed hash function, and the result is immediately passed to a new hash function with one-time seed. In the server, the database already holds the result of hashing the real password using the fixed hash function. As long as the server uses the same seed and algorithm as the browser used to perform a second hashing, the two results should match. The server is also responsible for clearing the one-time seed after a successful login; otherwise, an interceptor could log in later on by uploading the same data. Here's a summary of the algorithm:

1. User visits web site
2. Server outputs initial page.
 - a. Server generates one-time seed (S) and stores it.
 - b. Server outputs page, including login form, with one-time seed embedded somewhere on the page (e.g., in a JavaScript variable).
 - c. User enters username (U) and password (P)
3. Browser handles submission.
 - a. Browser hashes password (P) using permanent hash function in order to arrive at the attempted hash value (Ha), which should be held in the database.

- b. Browser combines attempted hash (Ha) with one-time seed (S) to create one-time, double-hashed value (Da).
 - c. Browser uploads username (U), double-hashed value (Da), and one-time seed (S).
- 4. Server authenticates
 - a. Verifies that one-time seed (S) is valid.
 - b. Server extracts stored hash for this user (H) and combines it with the seed (S) to get one-time, double-hashed value (D).
 - c. Server compares the double-hashed values (D and Da). If successful, it logs the user in (e.g., creates a new session and outputs a successful response code) and clears the one-time seed (S). If it is not successful, it either generates a new seed or decrements a usage counter on the existing seed.

Decisions

What hashing algorithm will be used?

You're going to be hashing in the browser as well as in the server, so you'll need a portable algorithm. Two popular standards are MD5 and SHA-1; both have implementations on JavaScript and on just about any server-side language you're likely to use.

How will you manage the one-time seed?

The double-hashing algorithm hinges on the one-time seed being used only once and on ensuring that the user authenticates with the seed that the server provided. You have to make a few decisions regarding this:

How does the seed expire?

In theory, the seed's lifetime shouldn't matter much since it will only be used once—there's no risk of someone intercepting a successful upload and reusing that data to authenticate. However, you'll probably want to clear any unused seeds periodically, e.g., once a day. More important than lifetime is number of validation attempts—perhaps you want to allow only three login attempts against the same seed. In this case, you'll need to associate a counter with the seed.

Is the seed uploaded back to the server?

The algorithm above requires that the seed be uploaded, but the server could instead track the session with a unique session ID and use that to look up the most recent seed it sent out. It's probably better to upload the seed in most cases, as it keeps the conversation as stateless as possible. With the seed having already been downloaded in plain-text, there's no significant threat posed by uploading it again.

Real-World Examples

NetVibes

The NetVibes (<http://netvibes.com>) portal handles the entire login process without any page refreshes.

Protopage

The Protopage (<http://protopage.com>) portal pops up a login box without opening a new page, though it sends you to a new page after the credentials are submitted.

Treehouse

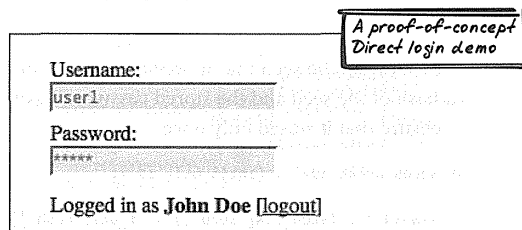
Treehouse Magazine (<http://treehousemagazine.com/>), has a sidebar with a login *Microlink*. When clicked, it expands to form a login area, which can in turn morph into a registration area. It also degrades to use standard form submission if JavaScript is disabled.

Code Examples: Ajax Login Demo

James Dam's Ajax Login (http://www.jamesdam.com/ajax_login/login.html) presents a standard HTML form (Figure 17-8). Submission is disabled and handled instead by callback methods registered on initialization:

```
<form action="post" onsubmit="return false">
  <div id="login" class="login">
    <label for="username">Username: </label>
    <input name="username" id="username" size="20" type="text">
    <label for="password">Password: </label>
    <input name="password" id="password" size="20" type="password">
    <p id="message">Enter your username and password to log in.</p>
  </div>
  <label for="comments">Comments:</label>
  <textarea rows="6" cols="80" id="comments"></textarea>
</form>
```

Demonstration



A proof-of-concept
Direct login demo

Username:
user1

Password:

Logged in as John Doe [logout]

FIGURE 17-8. Direct login demo

As soon as the user signals his intent to authenticate, which is indicated by form field focus, a random, one-time seed is retrieved from the server if there isn't already one

present. The response comes in two parts: an ID for the seed and the seed itself, both of which are saved as JavaScript variables. The server can later use the ID to retrieve the seed it sent:

```
function getSeed() {
    ...
    if (!loggedIn && !hasSeed) {
        http.open('GET', LOGIN_PREFIX + 'task=getseed', true);
        http.onreadystatechange = handleHttpGetSeed;
        http.send(null);
    }
    ...
}

function handleHttpGetSeed() {
    ...
    if (http.readyState == NORMAL_STATE) {
        results = http.responseText.split('|');
        // id is the first element
        seed_id = results[0];
        // seed is the second element
        seed = results[1];
    }
    ...
}
```

The seed is then used to hash the password upon submission. Notice that `hex_md5()`, the double-hashing operation, is used twice.

```
// validateLogin method: validates a login request
function validateLogin() {
    ...
    // compute the hash of the password and the seed
    hash = hex_md5(hex_md5(password) + seed);

    // open the http connection
    http.open('GET',
        LOGIN_PREFIX +
        'task=checklogin&username='+username+'&id='+seed_id+'&hash=
'+hash, true);
    ...
}
```

The server then validates by locating the seed that it previously sent out, and checking if the hash value matches a hash of the seed and the stored password hash. If it does, the server deletes the seed to ensure that it's used only once:

```
sql = 'SELECT * FROM seeds WHERE id=' . (int)$_GET['id'];
...
if (md5($user_row['password'] . $seed_row['seed']) == $_GET['hash']) {
    echo('true|' . $user_row['fullname']);
    ...
    mysql_query('DELETE FROM s WHERE id=' . (int)$_GET['id']);
}
```

After calling for validation, the browser receives a response and the form is morphed to show whether login was successful.

Related Patterns

Lazy Registration

Lazy Registration (see earlier) is geared toward first-time registration as well as deferred login, and it makes use of Direct Login.

Host-Proof Hosting

Host-Proof Hosting (see the next pattern), like Direct Login, uses JavaScript to perform encryption-related functionality.

Timeout

Apply a *Timeout* (see later) to log users out.

Acknowledgments

- The idea for this pattern comes from James Dam's demo and write-up (http://www.jamesdam.com/ajax_login/login.html).
- Peter Curran of Close Consultants (<http://closeconsultants.com>) provided valuable feedback that helped me clarify my explanation of the algorithm.

Host-Proof Hosting



ASP, DataCloud, Key, Secure, Untrusted

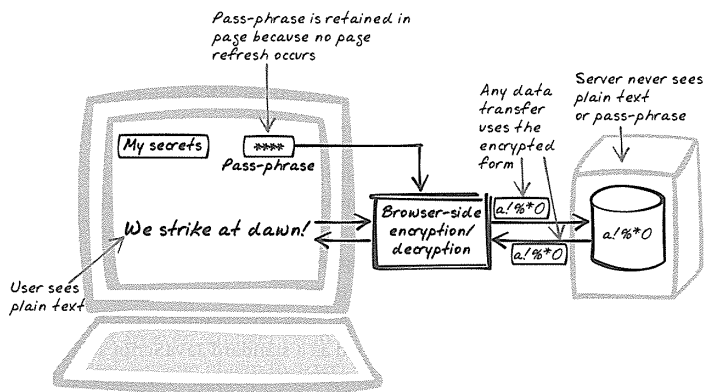


FIGURE 17-9. Host-Proof Hosting

Goal Story

Reta is dismayed to learn that a malicious hacker managed to download a chunk of the company's database containing personal details about all the customers in her store. Fortunately, she also learns that the pass-phrase she always entered was actually used to encrypt all that data, so the hacker won't be able to make sense of any of the contents.

Problem

How can you mitigate the effects of unauthorized access to your application data?

Forces

- Web apps require that some form of persistent data hold information about users, business state, past events, and so on.
- Security restrictions—using standard web technologies—prohibit users from storing web apps on the user's own hard drives. Even with *Richer Plugins* that allow it, many benefits of using a web app in the first place are lost. This means that persistent data is usually stored server side.
- Cookies allow some data storage within the browser, but cookie data is also transmitted to the server where it's vulnerable.
- Server-side storage is open to abuse: the administrators, along with anyone who is able to gain access to it, are able to extract sensitive information by reading the data, as well as effect malicious changes by tampering with it. The abuse can occur within an organization's own IT department or be inflicted by a third-party hosting company entrusted with the data.

Solution

Host sensitive data in encrypted form so that clients can only access and manipulate it by providing a pass-phrase that is never transmitted to the server. The server is limited to persisting and retrieving whatever encrypted data the browser sends it and never actually sees the sensitive data in its plain form. All encryption and decryption takes place inside the browser itself.

Just what does secure hosting have to do with Ajax? The Ajaxian twist comes in the maintenance of the pass-phrase. You could use the browser-side encryption with a conventional application, but the pass-phrase would have to be entered upon each page refresh, since no JavaScript state survives a reload. With page refreshes occurring every few seconds, the pass-phrase is completely unusable. However, using Ajax to avoid page refresh means you can retain all session state in the browser, so the pass-phrase only needs to be entered at the start. After being entered, it can be retained as a standard JavaScript string and will disappear from the browser when the user quits the browser or visits another site. Suddenly, Host-Proof Hosting becomes usable.

Incidentally, don't take this pattern to be a new "Ajax-HTTPS" protocol. The issue here is how the data is actually stored, not how it's transmitted. In theory, the data itself need not

travel over a secure connection because it's already encrypted. In practice, a secure connection might be worthwhile in order to reduce some of the vulnerabilities described later in this section.

Before you rush off to upload all your trade secrets to Shonky Hosting Inc., you should be aware that this idea isn't foolproof. On the one hand, the host is assumed to be inherently untrustworthy. But on the other hand, the script for the browser application is held right there on the server, and the browser runs whatever scripts come down from that URL. This leaves open the possibility that the host will tamper—to evil ends—with either the code itself or the outgoing HTML and JavaScript.

What if a rogue administrator from the hosting company decided to quietly add a small monitoring function to a JavaScript file and append its execution to a `window.onload` function. Then, the evil monitoring function could be made to run once a minute within the browser. It might, for instance, serialize the entire DOM and upload a summary back to the server with Web Remoting (Chapter 6), where more malicious code would log it somewhere convenient. A third-party hacker sitting between the browser and the server could also inject a script and could upload data to a remote site with one of several established techniques—for instance, by exporting browser data as CGI variables on the source URL of an external image under the hacker's control. In both scenarios, the application can continue as normal, and the poor user is none the wiser.

The threat of script injection certainly weakens the claim for this pattern, but it doesn't invalidate it altogether. While script injection is theoretically possible, it does require some skill on the host's part and is also detectable if you know what the code should and should not be doing. If you happened to discover that your application is uploading DOM details every sixty seconds (using a tool similar to those described in *Traffic Sniffing* [Chapter 18]), there's a good chance that something's blatantly wrong.

For practical purposes, also consider what happens if a hacker gains unauthorized access for a short time. She might well grab as much data as possible, but the data will be safely encrypted. In the unlikely event such a hacker was sophisticated enough to use script injection, she would only be able to gain pass-phrases of users who happen to be logged in during the time the server is under her control.

So pragmatic considerations suggest that the technique is safer than hosting the data in plain form, though it's by no means perfect. But is it so much safer as to warrant the extra performance overhead and coding effort and the constraint of zero page refreshes? That's a decision you'll need to make on a case-by-case basis, bearing in mind the critical nature of the data—the likelihood of the various types of attack.

In theory, there's an even stronger claim in favor of this approach. It might be possible to develop a general-purpose plugin to detect script injection. For a given application, such a plugin would have access to a certified copy of the source code. Then, it could monitor traffic and caution you about any unexpected activity. If such a plugin could be developed,

the only way for script injection to succeed would be a conspiracy between the host, the code certifier, and the plugin manufacturer.

Decisions

What encryption algorithm and framework will be used?

You'll need an algorithm that's available in JavaScript as well as in your server-side environment. If the data is accessed by other clients, they obviously must have access to the algorithm too. A search reveals that several algorithm implementations are available, each with its own strengths and weaknesses. All of the following are open source.

- RC4, AES, Serpent, Twofish, Caesar and RSA are provided by Michiel Van Everdigen's open source package (<http://home.zonnet.nl/MAvanEverdingen/Code/>).
- TEA (Tiny Encryption Algorithm) is provided by Moveable Type UK (<http://www.movable-type.co.uk/scripts/TEAblock.html>).
- Blowfish is provided by farfarfar.com (<http://www.farfarfar.com/scripts/encrypt/encrypt.js>).

When will the pass-phrase be requested?

The browser will need to query for the pass-phrase as soon as encrypted data must be rendered. However, that might not be immediately. To make the encryption less intrusive, you might consider using something like the *Lazy Registration* pattern, where regular data is shown as soon as the user accesses the application, with encrypted data only accessible after the pass-phrase has been entered.

Real-World Examples

There are no public real-world examples to my knowledge. One precedent is Hushmail (<http://hushmail.com>), which uses a Java applet to allow access to email encrypted on the server.

Code Example: Host-Proof-Hosting Proof-Of-Concept

Richard Schwartz provides a proof-of-concept demo (<http://smokey.rhs.com/web/test/AjaxCryptoConceptProof.nsf/blowfish?OpenPage>) (Figure 17-10). For encryption, it delegates to a JavaScript Blowfish library. The application accepts a pass-phrase, a message key, and some message content. It then encrypts the message and uploads it. It also uploads the key, along with an encrypted version of the key, which can be used later to check that the user has the correct pass-phrase. The application then shows that the server is holding only the encrypted content and the key. You can then pull the encrypted content back down and decrypt it with the original pass-phrase.

The application itself is quite simple: it manipulates the "display" style settings of a series of forms in order to show or hide them. Thus, the user's pass-phrase remains in the pass-phrase input field at all times. This is the important thing about the demo; there's no form submission, so the pass-phrase needs to be entered only once.

Ajax Crypto Proof-of-Concept

*A message is encrypted
with the user's pass-phrase*

Test #1 Blowfish

Here is the encrypted data that was saved on the server

518C5BAE5386ABB8C32CCA7FF956631F07AAF292D6748AA9A3604A549FE633A0

Click [here](#) to view the saved data on the server and verify that it is encrypted. (Opens a new window.)

Click [here](#) to go to the next step 

FIGURE 17-10. Host-Proof Hosting

When the application is ready to upload the encrypted data, it sets up the global variables required by the Blowfish library (ideally, the library would accept these as parameters). `encodetext()` is called (with "2" to specify the Blowfish algorithm), and it outputs the encrypted version in the form of a global variable:

```
saveCryptoText() {  
  ...  
  inpdata=window.document.inputForm.plaintextInput.value;  
  passwd=window.document.inputForm.password.value;  
  // invoke blowfish  
  encodetext(2);  
  data = data + "&check=" + outdata ;  
  ...  
}
```

The encrypted key is also attached:

```
inpdata=window.document.inputForm.check.value;  
encodetext(2);  
data = data + "&check=" + outdata ;
```

The data can now be sent to the server:*

```
url = "http://smokey.rhs.com/web/test/AjaxCryptoConceptProof.nsf/  
SaveBlowfishDoc?OpenAgent" + data  
httpPost(url)
```

Later on, the application provides a list of message keys that have been sent to the server. When the user chooses one, the application requests an XML document from the server containing the message and key details (in practice, the message key could be verified before the body is downloaded). It first performs a check that the key is valid for this pass-phrase, then decrypts the message itself. Finally, a DOM object is morphed to display the decrypted text:

```
inpdata = "";  
inpdata = getElementText(valuenode);
```

* RESTful principles suggest that, in practice, the data would be uploaded with a POST, because the data would affect the server state. Another reason to use POST is that it's not practical to encode an entire message in the URL. However, this application is a proof-of-concept, so GET is used here instead.

```

outdata = ""
decodetext(2);
...
window.document.all.decryptedText.innerHTML = stripNulls(outdata);

```

Alternatives

Richer Plugin

You might consider exploiting the increased permissions of a *Richer Plugin* (Chapter 8) to access a local data store. However, with this there is still a risk of malicious script injection from the server. Furthermore, you'll lose several key benefits of holding the data server side:

- Users will not be able to access the data from remote locations.
- Each local data store will need its own backup process. Uploading to a backup server will defeat the purpose of the local store.
- Each local data store must be protected against unauthorized access.
- The data stores will sometimes need to be migrated as the system changes.

Another application of *Richer Plugin* would be to hold the data server side but use the Richer Plugin in place of the JavaScript code to manage the local encryption and decryption. A plugin like this could also retain the pass-phrase.

Related Patterns

Direct Login

Direct Login, like Host-Proof Hosting, also involves using JavaScript for encryption-related activity.

Timeout

Apply a *Timeout* to clear the key from browser state after an idle period. This will help you prevent unauthorized users from accessing the encrypted data.

Metaphor

Caesar wonders whether he can entrust his aide with the top-secret recipe for victory wine. Fortunately, he's a pioneer of cryptography, so he just hands over the recipe in encrypted form.

Want to Know More?

Check out Richard Schwartz's blog entries:

- Overview of the idea and pertinent conversation in the comments section (<http://smokekey.rhs.com/web/blog/PowerOfTheSchwartz.nsf/d6plinks/RSCZ-6C5G54>)

- Practical implementation issues (<http://smokey.rhs.com/web/blog/PowerOfTheSchwartz.nsf/plinks/RSCZ-6CCMCD>)
- Introducing the proof-of-concept application (<http://smokey.rhs.com/web/blog/PowerOfTheSchwartz.nsf/plinks/RSCZ-6CATX6>)
- A hypothetical scenario elaborating the idea (<http://smokey.rhs.com/web/blog/PowerOfTheSchwartz.nsf/plinks/RSCZ-6CHL5J>)
- My blog entry summarizing the idea (<http://www.softwareas.com/ajax-and-the-great-data-cloud-in-the-sky>)
- Chris Hammond-Thrasher comments on Ajax as a general solution to encryption issues, e.g., digital signatures and secure timestamps (<http://thrashor.blogspot.com/2005/05/more-on-ajax-and-secure-web.html>)

Acknowledgments

Richard Schwartz's blog entries provided the idea for this pattern, and its name is attributed to Richard Schwartz, Michael Griffes, and their colleagues at eVelocity. I am also grateful to others who have commented on the approach, notably Alex Russell, who has cautioned on the vulnerabilities of this approach, such as script injection.

Timeout



Logout, Screensaver, Session, Suspend, Timeout

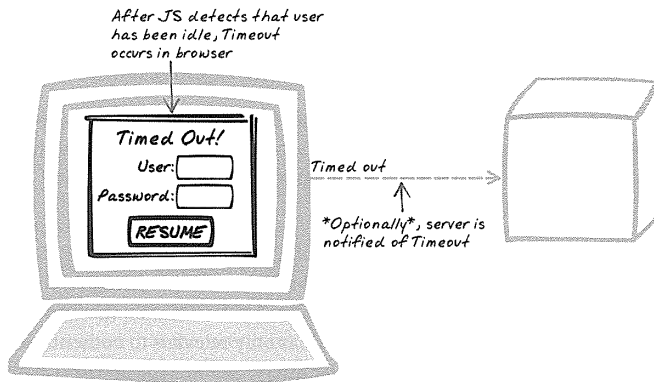


FIGURE 17-11. Timeout

Goal Story

Tracy was so keen on leaving the office Friday afternoon that she forgot to shut down the market news web site. Fortunately, the web site monitors mouse movements, meaning that if she doesn't interact with the page for an hour, it will suspend itself, saving a lot of

wasted refreshes. On Monday morning, Tracy walks in and sees a screensaver-like animation in the browser, with a message indicating that updates have ceased. She clicks a mouse button and sees the screen update with the latest prices.

Problem

How can you tell whether the user is still working with the application?

Forces

- Users often leave an application in a browser window for hours or days without interacting with it. Many times, they abandon it altogether. The practice is becoming more prevalent as all major browsers now support multitab browsing.
- Ajax Apps often use *Periodic Refresh* or *HTTP Streaming* to keep informed about server state. The user may not care about the application anymore, but if it's still in the browser, a naïve implementation will continue to poll the server indefinitely—a massive waste of bandwidth and resources.
- Ajax Apps often contain sensitive data which is at risk if the user forgets to log out when leaving a public terminal.
- Ajax Apps often involve collaboration and communication between users. To cultivate an awareness of other users and their activities, the server must track the state of individual browsers.

Solution

Have the browser Timeout the user after a period of inactivity and, optionally, inform the server. After a period of inactivity, the application is either suspended, requiring the user to manually resume it or shut down, requiring the user to restart.

This pattern raises the whole question of sessions in Ajax. Conventional applications usually have server-side session objects which represent the user's current session. Typically, this means short-lived information such as shopping cart contents or recently viewed items.

In Ajax, server-side sessions are not so useful. In fact, if you rely solely on *RESTful Services*, you may not need those session objects at all. In browser-centric Ajax Apps, the browser is a full-fledged application with all of the session state held within. So the browser state *is* the session. The server-side session, if used at all, is only relevant to authentication and has no impact on the response to any particular query. See *RESTful Service* (Chapter 9) for more details.

How is all of this relevant to Timeout? Well, consider this familiar nightmare scenario:

The user opens up a form, spends two hours populating it while researching the answers, and clicks submit. The browser then responds with “Your session has expired” and adds insult to injury by presenting a new blank form.

Stories like this are all too common because server-side session Timeouts ignore what’s happening in the browser. If the user hasn’t submitted a form in, say, 30 minutes, the session times out and all the data is lost. That’s a problem if the user has been actively working on the browser side. With an Ajax App, the server has a better chance of staying in sync with the browser thanks to *XMLHttpRequest Calls* (Chapter 6). In most environments, the server-side session will be automatically refreshed when an *XMLHttpRequest Call* comes in. However, this is not optimal either because the server still doesn’t know why browser calls are occurring. For applications using *Periodic Refresh* (Chapter 10), for example, a server-side session will stay alive indefinitely even if the user has left the building.

By relying instead on browser-based Timeout detection, you can be more intelligent about how Timeouts will occur. Browser-based Timeout detection is based on *Scheduling* (Chapter 7). A timer is established to count down to Timeout state. Each significant activity cancels the timer and starts a new one in its place.

What activities are significant enough to restart the timer? Mouse movement is one candidate. You can monitor each mouse movement on the page; one of the following Code Refactoring illustrations below does exactly that. Mouse movements are a very broad indicator, and you might be concerned about “false positives”: These may be suggestions that a user is still working with an application when they’re in fact not, e.g., they just happened to move the mouse across the browser window while cycling through each open window. You’ll also get some “false negatives”: A user working only with the keyboard is liable to be timed out. If you’d prefer to keep the session going only if the user is actively changing data, you can instead catch events such as button clicks and keypresses within input fields.

Sudden Timeouts can be frustrating for users, so consider these feedback measures:

- Unobtrusively show Timeout status, e.g., number of minutes remaining. You might highlight the status as Timeout approaches, and also offer some background and instruction on avoiding the Timeout.
- When Timeout is approaching, offer a warning and an opportunity to explicitly restart the timer.

Once you’ve detected that the user is inactive, what do you do? Timeout can be used in several ways:

Stop Periodic Refresh

Conventional applications tend to do nothing when the user is idle, but many Ajax Apps continuously poll the server. Thus, the Timeout should trigger cancellation of any *Periodic Refresh* timers.

Clear data

Sometimes an inactive application is a hint that the user may no longer be in control of a terminal. There's a risk that someone else could gain access to the user's data and permissions. After a Timeout, `document.url` can be pointed to the main application URL in order to refresh the page. In addition, you might decide to delete cookies holding sensitive data.

Save data

If you clear data, you might also consider first *saving* it on the server. There's nothing more annoying than arriving back from lunch and finding that a partly filled-out form has been cleared for security purposes. With an *XMLHttpRequest Call*, you have the option of saving the user's progress as a "draft."

Inform user

If you're performing Timeout activities, you'll need to provide some feedback to the user about what's happening and what he should do about it.

Inform server

Although the browser controls session Timeouts, the server can nevertheless be kept informed, and it can use server-side sessions for this purpose. There are quite a few applications of server-side session tracking:

Invalidating server-side session

The server can invalidate any server-side session data for security purposes.

Historical records

The server can retain Timeout data to help personalize the web site, to provide feedback to users, and also to inform you, the site operator.

Multiuser awareness

In multiuser systems and especially in any sites hoping to foster a community, users should be aware of each others' activities. Indeed, the "live web"—or, as Technorati puts it, "What's Happening on the Web Right Now?"—is becoming a key phenomenon on the Web. If the server is able to track what users are working on and how long they have been idle, this information (subject to privacy considerations) can be conveyed to other users. That's especially important when users are collaborating on the same workspace—for instance, in an Ajaxian wiki environment.

Pessimistic locking

Pessimistic locking is a technique used to prevent two users from working on the same thing at once. In a wiki, for example, a user can lock the article and thereby have sole access to it until it's saved or until the user is idle for a while. While this can be open to abuse on the public web, it would actually be quite a good model for many intranet applications. However, it hasn't been practical because of the risk that the user might walk away and leave the lock intact. Server-side session Timeouts don't help much because the user might be busily working in the browser but not be ready yet to submit content. But once the server is aware of what the user is

doing in the browser, pessimistic locking starts to become practical. If the user hasn't done anything for the last 30 minutes, for example, she loses the lock. Again, Ajax helps with this, because with *Periodic Refresh* (Chapter 10), you can immediately inform the user that she has lost the lock.

Work scheduling

With this variant of *Predictive Fetch* (Chapter 13), it's possible for the server to proactively prepare for future actions. By tracking active users, the process becomes more focused. For example, a blog aggregator could run a periodic background process to build a list of unread articles for each active user should they decide to view the list.

So informing the server of a Timeout is a good thing. But there's still a small problem with this approach: What if the user quits the browser or moves away to another site? Then a Timeout will never occur and the server will blissfully continue assuming that the user is logged in for all eternity. To alleviate this problem, have the server refresh the user's record upon each incoming request that suggests user activity. The rest remains, however, of inadvertently timing out a user who is performing browser-only activity. So what we need is a way for the browser to keep telling the server that the user is still active, explicitly noting that a Timeout hasn't actually occurred. That's exactly what the *Heartbeat* pattern is about—see later in this chapter for more details.

Decisions

How long will the Timeout period be?

The Timeout period will depend on the purpose for having the Timeout period in the first place. For example, if the purpose is primarily to stop *Periodic Refresh* (Chapter 10), you need to weigh the benefits of reduced bandwidth and server costs against the frustration caused to users who will have to reactivate the page and wait a few seconds for the latest data. In practice, there are probably several reasons to use Timeout, and the needs in each situation must be taken into consideration.

How will Timeout affect the user interface?

A Timeout can impact the interface in different ways. You may choose to use any of the following:

- Wipe the entire display and show a message that the user was timed out or the session suspended. This is worthwhile if the data is sensitive.
- As a variant on the previous option, reload the entry point for the application (as if the user had typed in the main URL).
- Perform a *Page Rearrangement* to insert a message about the Timeout into the page.

In addition, if you want to be really sure that users know they've been timed out, you could produce an alert. Use Timeout alerts with caution: it's quite obtrusive, and you could probably make the Timeout equally obvious with a carefully considered page element. A *Popup* (Chapter 15) is a less obtrusive mechanism.

Real-World Examples

Many conventional web sites have a session Timeout feature, though it's implemented server side rather than the way I've described in this section.

Lace Chat

Brett Stimmerman's Lace Chat (<http://www.socket7.net/lace/>) is an Ajax chat app. As such, it requires a *Periodic Refresh* in order to keep showing new messages, a serious bandwidth concern if the user is no longer interested in the conversation. So after some idle time, a dialog box appears indicating that Lace has stopped; it includes a button allowing the user to resume (Figure 17-12). Lace also has a *Status Area* (Chapter 15) that always shows whether the application is running—i.e., whether the server is being polled. In fact, you can pause and resume it manually too. The Timeout mechanism hooks into this state by forcing the application to pause upon Timeout.

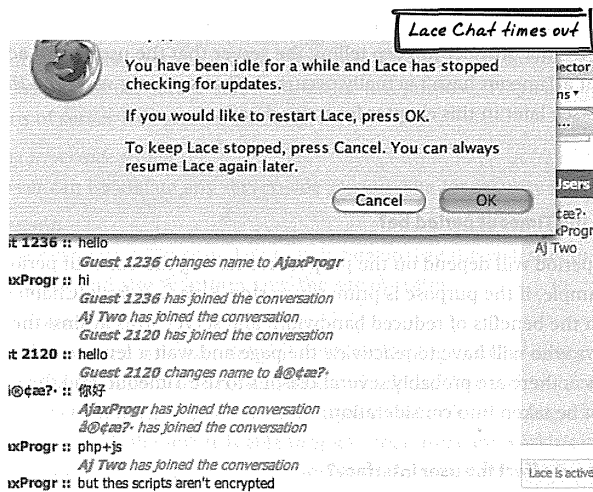


FIGURE 17-12. Lace Chat Timeout

Pandora

Pandora uses Flash to stream music into a browser. After a long idle period, it stops playing and produces a dialog to confirm that the user is still listening.

Session Warning Demo

Eric Pascarello has explained a demo system which has a standard server-side session Timeout but which also pops up a warning near Timeout allowing the user to renew the Timeout (<http://radio.javaranch.com/pascarello/2005/07/05/1120592884938.html>). There are implementations in both VB.net and JSP.

Operating system Timeouts

A precursor to Ajaxian Timeout is evident in operating systems, such as Apple OSX, that show a warning when they are about to enter standby mode. A pop-up dialog says something like “You haven’t used the mouse or keyboard for 15 minutes. Standby mode will start in 30 seconds.” The dialog will count down, and the user can prevent the Timeout with a keyboard or mouse action or by explicitly hitting a Cancel button. A similar precursor is present in remote terminal systems that output an idle warning to the console.

Code Refactoring: AjaxPatterns Timeout Wiki

Introducing Timeout to the wiki

Timeout is a very useful pattern for a wiki on several counts. Firstly, it cuts down on *Periodic Refresh* (Chapter 10) wastage. Secondly, it helps users understand what others are up to. Thirdly, it improves security if users are logged on. For all of these reasons, we will now add some Timeout functionality to the Basic Wiki Demo (<http://ajaxify.com/run/wiki>). (The Periodic Refresh Time Demo [<http://ajaxify.com/run/time/periodicRefresh>] also has a couple of similar Timeout refactorings under it.)

Initial refactoring: unconditional Timeout

This initial version, Timeout Wiki Demo (<http://ajaxify.com/run/wiki/timeout>), is merciless. Log into the wiki and, no matter what you do, you’re timed out a few seconds later (Figure 17-13). The point, of course, is to introduce and prove the basic Timeout functionality, which can be built on later.

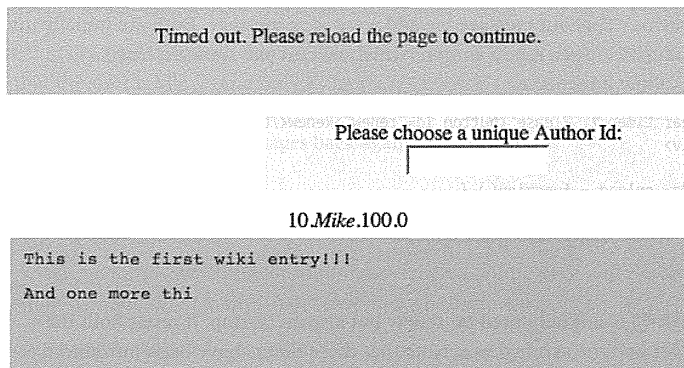


FIGURE 17-13. A timed-out wiki

A Timeout message that will initially be hidden is added to the initial HTML:

```
<div id="timeoutMessage">
  Timed out. Please <a href=".">reload the page</a> to continue.<br/>
</div>
```

The script begins by hiding the Timeout message and kicking off a Timeout timer:

```
window.onload = function() {  
    ...  
    $("timeoutMessage").style.display = "none";  
    timeoutTimer = setTimeout(onTimeout, TIMEOUT_TIME);  
}
```

When Timeout occurs, the Timeout message is made visible (with some fanfare, courtesy of a Scriptaculous effect [<http://script.aculo.us>]). Most importantly, the periodic sync stops, so the server is no longer polled. All the messages are removed from the page as well:

```
function onTimeout() {  
    new Effect.BlindDown($("timeoutMessage"));  
    stopPeriodicSync();  
    removeMessages();  
}  
...  
function removeMessages() {  
    while ($("messages").hasChildNodes()) {  
        $("messages").removeChild($("messages").firstChild);  
    }  
}
```

Warning that Timeout is pending

If only for the selfish reason that we want to avoid being flamed by the user, it would be nice to warn them that a Timeout is about to occur. And nicer still to let them prevent it from occurring. So this demo (<http://ajaxify.com/run/wiki/timeout/warning>) introduces a warning mechanism.

To the previous Timeout message we add a warning message. Both are initially invisible. The warning includes a renew button, which will invoke `renewSession()`.

```
<div id="warningMessage">  
    Near timeout. Please <button id="renew">Renew</button> your session now.  
</div>  
  
window.onload = function() {  
    ...  
    $("renew").onclick=renewSession;  
    ...  
}
```

`renewSession()` is not only used on renew but also on startup. It resets both the `timeoutTimer` and the `warningTimer`. Note that these timers have been introduced as variables in the script, precisely so that we can cancel them in this method. In addition, `renewSession()` kicks off the standard wiki sync timer and hides the warning and Timeout messages in the case that either is showing:

```
var timeoutTimer = null;  
var warningTimer = null;  
...  
function renewSession() {
```

```

    $("#warningMessage").style.display = "none";
    $("#timeoutMessage").style.display = "none";
    clearInterval(warningTimer);
    clearInterval(timeoutTimer);
    warningTimer = setTimeout(onWarning, WARNING_TIME);
    timeoutTimer = setTimeout(onTimeout, TIMEOUT_TIME);
    startPeriodicSync();
}

```

So as long as the user clicks on the renew button while it's showing, he can force all timers to be restarted. But if he doesn't click on it, Timeout will proceed. We simply have to ensure that the warning message is shown before the Timeout message, i.e., WARNING_TIME must fall short of TIMEOUT_TIME. When Timeout occurs, we can assume that the warning is already being shown, so we replace it with the Timeout message. And as before, we stop synchronization and remove all messages:

```

function onTimeout() {
    new Effect.BlindUp($("#warningMessage"));
    new Effect.Appear($("#timeoutMessage"));
    stopPeriodicSync();
    removeMessages();
}

```

Monitoring mouse movements

A warning's better than nothing, but it does have the slight odor of techno-centeredness. From the user's perspective, why should she have to explicitly renew the session. What's this Timeout business anyway? For most users, it's better to handle Timeout on their behalf. This demo (<http://ajaxify.com/run/wiki/timeout/monitoring/>) builds on the initial Timeout demo to suspend activity when the user is idle.

Here, we'll make an assumption that the user is working with the mouse. That is, we can assume that the user is idle if (and only if) he hasn't used the mouse for a while. As before, we have only one Timeout message, which we've altered to help the user understand how to bring all of those messages back again:

```

<div id="timeoutMessage">
    Timed out. Please wiggle your mouse to continue.<br/>
</div>

```

The key to this pattern is watching for mouse movements on the page and renewing the session when one occurs. Actually, that's easy enough to achieve:

```

window.onload = function() {
    ...
    document.getElementsByTagName("body")[0].onmouseover = function(event) {
        renewSession();
    }
    ...
}

```

Calling a function on each mouse movement could get grossly inefficient. In real life, you might want to be just a little more intelligent about this. For example, use some sort of

throttling algorithm to ensure that `renewSession()` isn't called more than once every few seconds.

`renewSession()` itself works similarly to the preceding warning demo: it hides the Timeout message and kicks off `sync` and Timeout timers.

```
function renewSession() {
  if ($("#timeoutMessage").style.display!="none") {
    new Effect.BlindUp($("#timeoutMessage"));
  }
  clearInterval(timeoutTimer);
  if (!syncTimer) {
    startPeriodicSync();
  }
  timeoutTimer = setTimeout(onTimeout, TIMEOUT_TIME);
}
```

Related Patterns

Heartbeat

Heartbeat (see the next pattern) is a companion pattern that helps the server monitor whether the user is still working in the browser.

Periodic Refresh

For Ajax Apps, a big motivation for Timeout functionality is the prospect of reducing *Periodic Refreshes* (Chapter 10).

Progress Indicator

Progress Indicator (Chapter 14) is normally used for feedback during an *XMLHttpRequest Call*. However, another application of it would be to count down toward the Timeout. The display could be present as a permanent fixture, or, alternatively, you could introduce it near Timeout.

Direct Login

If your Timeout requires reauthentication for users to continue working with the system, you probably want to offer a *Direct Login* (see earlier).

Status Area

You can show Timeout-related messages and countdowns in a *Status Area* (Chapter 15).

Popup

You can show Timeout-related messages and countdowns in *Popups* (Chapter 15).

Metaphor

If a car engine remains idle long enough, you're probably going to assume that it's dead and give up trying to start it. To keep going would only be a waste of energy.

Heartbeat



ACK, Announcement, Flash, Heartbeat, Monitor, Signal

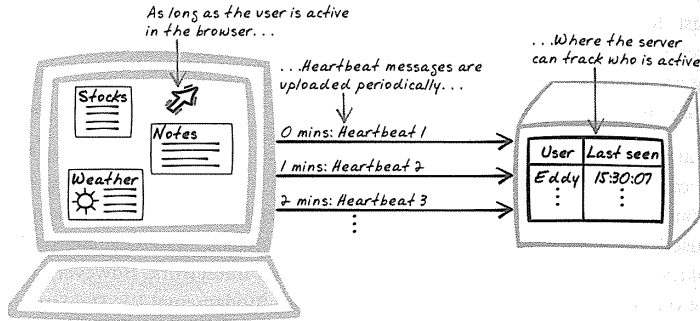


FIGURE 17-14. Heartbeat

Goal Story

For auditing purposes, Frank must run the factory decision support system all day long, so the server needs to ensure that it's always alive in the browser. But requests won't come in very often because it's a fat client, with the business logic in JavaScript. To ensure that the server keeps getting requests, the browser explicitly uploads a Heartbeat message every 10 minutes.

Problem

How do you know the user still has an application in the browser and is actively working with it?

Forces

- It's useful for the server to track whether the user is active, with several applications highlighted in the Solution of *Timeout*.
- Due to the stateless nature of HTTP, the server doesn't know when the user has quit the browser, experienced a browser crash, or surfed away to a different URL. Note that you could use the JavaScript `onunload` event to catch the last of these, but it won't work for the first two. In tracking the user's activity, the server may assume the user is still active when he has in fact abandoned the application.
- The user sometimes spends a long time working in the browser, but in a manner which yields no calls to the server—for example, filling out a long form or playing a game. The server has no way of knowing that the user's still around if no calls are made.

Solution

Have the browser periodically upload Heartbeat messages to indicate that the application is still loaded in the browser and the user is still active. The server keeps track of each user's "last Heartbeat." If the Heartbeat interval is 10 minutes and the user's last Heartbeat was longer than 10 minutes ago, the server knows that the user is no longer active. The objective is to help the server track which users are active. Session tracking isn't essential in Ajax Apps, where it's possible to hold all session data as JavaScript state, but, as described in the Solution of *Timeout*, there are still reasons to do it.

Heartbeat messages are uploaded to a special "Heartbeat service" using *XMLHttpRequest Calls*. Because they affect server state, they should be POSTed in. The Heartbeat service will update the user's last Heartbeat record, but how does it associate a Heartbeat message with a user? Heartbeat relies on some form of session management. One approach is to use cookies, either directly or via a cookie-based session framework. However, if you do that, the conversation will be stateful, thus violating a fundamental *RESTful Service* principle. A cleaner approach is to explicitly include the session ID in the Heartbeat body. Note that you shouldn't just upload the user ID, as others could easily fake the user ID.

Heartbeat is closely related to *Timeout*, but they work in different ways. You can use either independently, but Heartbeat works best as a supplement to *Timeout*. The point of *Timeout* is to stop the browser application after an idle period for security and bandwidth reduction. Notifying the server of *Timeout* events yields extra benefits, but it only works if the application is still sitting in the browser and working fine. That's why we use Heartbeat messages, which are, in a sense, the inverse of *Timeouts*, and therefore a good supplement. Whereas a *Timeout* message has the browser announce when a timeout has occurred (Figure 17-15), a Heartbeat message has it continuously announced that a *Timeout* *hasn't* occurred (Figure 17-16). As soon as the server detects a missed Heartbeat, it can assume the application is no longer running.

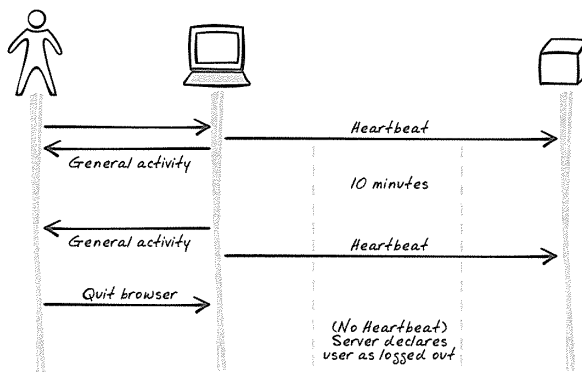


FIGURE 17-15. Heartbeat sequence

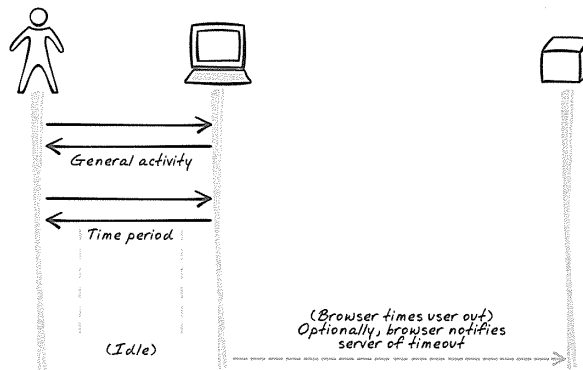


FIGURE 17-16. Timeout sequence

As an alternative to Heartbeat, you can maintain a “Last Seen” or “Last Request” field to track the last time the user issued a request. Thus, not only Heartbeat messages but any other messages will refresh the user’s record. The Heartbeat is just a backup. Using these fields might paint a more meaningful picture if you’re showing the timestamp to other users or feeding it into analysis.

This pattern is purely speculative in the context of Ajax applications. However, Heartbeats are commonplace in enterprise messaging systems, where they are used to monitor the status of components throughout a network.

Decisions

How will you maintain user records?

There are two main ways to maintain user records:

In memory

The data is not persisted and will be lost once the user is timed out. In most environments, this will happen if you rely on standard session objects.

Directly in the database

Within persistent user records. There’s some extra storage and data maintenance involved, but the benefit is that you always know when the user was last seen.

What, if anything, will cause the browser application to stop sending Heartbeats?

One option is for the browser application to *always* send Heartbeats. This lets the server track whether the application is still sitting in the browser, which may be useful for analysis purposes. More likely, though, you probably want to send Heartbeats only if the user is actively working with the browser. Thus, as mentioned in the preceding Solution, the Heartbeat is effectively a message that says “the user has not yet timed out.”

How much time between Heartbeats? How much delay until a user is declared inactive?

You need to decide on the time period between Heartbeats. If it's too long, the information will not be very useful. If it's too short, you'll be placing a strain on the network, as well as impacting browser and server performance.

The appropriate figure could vary from subseconds to up to 30 minutes or more depending on the following factors:

Application

In some applications, up-to-the-second information is more critical. In a multiuser system, for example, users' work may be dictated by whoever else is present. If you wait 10 minutes to tell Alice that Bob has quit the chess game, she might be annoyed that she wasted the last 10 minutes thinking about her next move.

Available resources

Ideally, the period should be as short as possible, though you can't always justify this. For an intranet application, you're likely to use a shorter period in recognition of better resources per user.

Real-World Examples

As this pattern is speculative, there are no real-world examples at this time.

Code Refactoring: AjaxPatterns Heartbeat Wiki

The *Timeout* pattern (see earlier in this chapter) refactored the wiki to produce the Timeout Wiki Demo (<http://ajaxify.com/run/wiki/timeout>), and two further refactorings from there. The present refactoring (<http://ajaxify.com/run/wiki/timeout/heartbeat>), as seen in Figure 17-17, creates a third version of the basic Timeout demo, introducing a Heartbeat. Note that Heartbeat can work independently of *Timeout* but works better in tandem, as this pattern demonstrates.

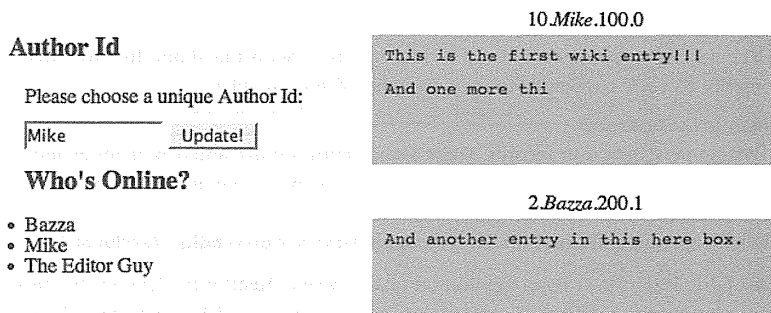


FIGURE 17-17. Showing user status via Heartbeat

To illustrate Heartbeat, I'll show one of its main applications—an application that lets users see who else is "currently online", i.e., who else has a recent Heartbeat registered.

Thus, each user is assigned a random ID, and a Heartbeat mechanism is used to maintain the “Currently Online” list.

But first, let’s look at the Heartbeat mechanism. Server side, the Author class contains a `lastRequest` property:

```
class Author {
  private $id;
  ...
  private $lastRequest;
  ...
}
```

The Heartbeat service accepts a message with an author ID and updates the user’s `lastRequest`. Note that a production system should accept a session ID rather than a user ID. The service extracts the author from the database and updates its `lastRequest`. If the author does not yet exist, a new record is created with `lastRequest` set to the present. Finally, the record is persisted:

```
if (isset($_POST['renew']) && $_POST['renew']=="true") {
  $authorId = $_POST['author'];
  $now = time();
  $author = $authorDAO->fetch($authorId);
  logInfo("Got author record for $authorId");
  if ($author) {
    logInfo("Updating timestamp for user $authorId");
    $author->setLastRequest($now);
  } else {
    logInfo("Adding author $authorId");
    $author = new Author($authorId, $now, $now);
  }
  $authorDAO->persist($author);
}
```

A periodic loop is set up to access the Heartbeat service. Because the Heartbeat call is “fire-and-forget,” an empty callback function is used:

```
window.onload = function() {
  ...
  heartbeatTimer = setInterval(sendHeartbeat, HEARTBEAT_PERIOD);
  ...
}
function sendHeartbeat() {
  vars = {
    renew: true,
    author: authorId
  };
  ajaxCaller.postForPlainText("session.phtml", vars, function() {});
}
```

And that’s the basic Heartbeat pattern. But there’s a bit more to do because we’re integrating it into *Timeout*. We want to stop sending Heartbeats when a timeout has occurred in

the browser. In the preceding code, we created a variable, `heartbeatTimer`, to track the periodic Heartbeat calls. Since we have a handle on the periodic process, we are able to cancel it upon timeout:

```
function onTimeout() {
    ...
    clearInterval(heartbeatTimer);
    ...
}
```

The other feature here is the list of online users, which illustrates one way to use the Heartbeat data. `session.phtml` exposes an XML list of currently online users (<http://ajaxify.com/run/wiki/timeout/heartbeat/session.phtml?current>). The `fetchRecentlyActive` method accepts a parameter indicating just *how* recently active the records should be. In this case, we ask for users with a request in the past 10 seconds:

```
if ($_GET="current") {
    header("Content-type: text/xml");
    echo "<authors>";
    foreach ($authorDAO->fetchRecentlyActive(10) as $author) {
        echo "<author>{$author->getId()}</author>";
    }
    echo "</authors>";
}
```

In the browser, there's a div to contain the list:

```
<h1>Who's Online?</h1>
<div id="currentlyOnline"></div>
```

A timer is established to perform a *Periodic Refresh* on the current data. Every few seconds, the XML is requested and the callback function transforms it into HTML for display:

```
currentlyOnlineTimer = setInterval(updateCurrentlyOnline, CURRENTLY_ONLINE_PERIOD);

function updateCurrentlyOnline() {
    ajaxCaller.getXML("session.phtml?current", function(xml) {
        $("currentlyOnline").innerHTML = "<ul>";
        var authors = xml.getElementsByTagName("author");
        for (var i=0; i<authors.length; i++) {
            var authorName = authors[i].firstChild.nodeValue;
            $("currentlyOnline").innerHTML += "<li> " + authorName + "</li>";
        }
        $("currentlyOnline").innerHTML += "</ul>";
    });
}
```

As with the Heartbeat timer, the “currently online” timer is cancelled upon timeout, so no more *Periodic Refreshes* will occur:

```
function onTimeout() {
    ...
    clearInterval(currentlyOnlineTimer);
    ...
}
```

Related Patterns

Timeout

Timeout (see earlier) is a companion pattern, as discussed in the preceding “Solution.”

Submission Throttling

Heartbeat resembles *Submission Throttling* and *Periodic Refresh* (Chapter 6) insofar as all have a continuous *XMLHttpRequest Call* cycle. However, Heartbeat has a more specific aim—namely, informing the server of browser state.

Metaphor

The name “Heartbeat” is, of course, taken from a metaphor with the biological heart.

Want to Know More?

Go to http://www.mindspring.com/~mgrand/pattern_synopses3.htm#Heartbeat for a brief summary of Enterprise Java Heartbeat pattern. See Mark Grand’s “Java Enterprise Design Patterns” for the full pattern.

Unique URLs



Address, Bookmarks, Cut-And-Paste, Distinct, Favorites, Hyperlink, Location, Link, Mail, Share, Unique, URL

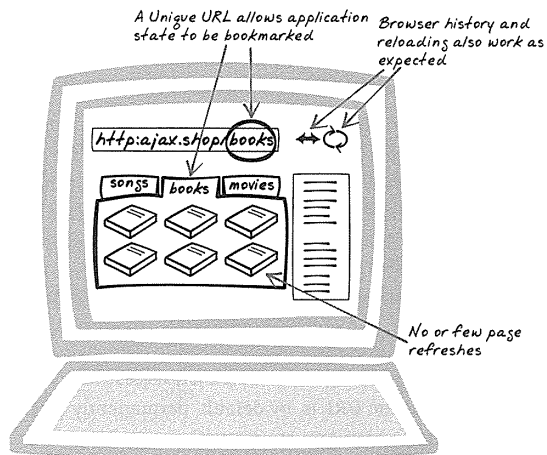


FIGURE 17-18. Unique URLs

Goal Story

Sasha is exploring a recent news event by browsing a map of the area. There's a Unique URL for each location, so she's able to link to it in her blog and add it to her social bookmarking account.

Problem

How can you assign distinct URLs to different parts of your application?

Forces

- Expressive URLs are vital. The ability to link from one site to another is arguably what made the Web successful. Furthermore, web surfers like to bookmark pages for later on. And these bookmarks are now a social phenomenon, with the advent of "link-blogs" and social bookmarking software like del.icio.us (<http://del.icio.us>).
- Browsers retain URLs in their histories, and the Back button will only work if URLs vary. The Back button is a big habit for many users. It can mean "Go back to where I was a minute ago." For many users, it can also mean "Undo my last action," and that mental model must be respected even though some experts deem it to be technically incorrect.
- A single address for your whole site is not enough; you need to support *deep linking* for links to be effective. That is, each distinct component of a web site should have its own URL.
- In conventional web sites, the URL changes when the page is refreshed—e.g., when the user submits a form or follows a link. But in Ajax Apps, the page never refreshes, so the browser won't change the URL.

Solution

Provide Unique URLs for significant application states. Each time a significant state change occurs—e.g., the user opens a new product in an e-commerce system—the application's URL changes accordingly. The objective is straightforward, but as discussed later in this section, the implementation requires a few unusual tricks. This Solution is fairly complex, because it delves into the details of building Unique URLs. Increasingly, though, reusable libraries will save you the effort, so be sure to check out what's available before directly applying any of the techniques here. The Real-World Examples identifies a couple of resources, and there will doubtless be others by the time you read this.

In Ajax, most server communication occurs with XMLHttpRequest. Since *XMLHttpRequest Calls* don't affect the page URL, the URL is, by default, permanently stuck on the same URL that was used to launch the Ajax App, no matter how many transfers occur. The only option you have to manipulate the URL is to use JavaScript. And, in fact, it's very easy to do that with the `window.location.href` property:

```
window.location.href = newURL; // Caution - Read on before using this.
```

Very easy, but there's one big problem: under normal circumstances, the browser will automatically clear the page and load the new URL, which sort of defeats the purpose of using Ajax in the first place. Fortunately, there's a cunning workaround you can use. It was originally conceived in the Flash world and has more recently been translated to an Ajax context. Mike Stenhouse's demo and article (<http://www.contentwithstyle.co.uk/Articles/38/fixing-the-back-button-and-enabling-bookmarking-for-ajax-apps>) provides a good summary of the approach, and the Solution here is based heavily on his work.

The cunning workaround relies on fragment identifiers—those optional components of URLs that follow the hash character (#). In http://ajaxpatterns.org/Main_Page#Background, for example, the fragment identifier is *Background*. The point of fragment identifiers has traditionally been to get browsers scrolling to different points on a page, which is why they are sometimes called “in-page links” or “named links.” If you're looking at the Table of Contents on AjaxPatterns.org (<http://ajaxpatterns.org#toc>) and you click on the Background link—<http://ajaxpatterns.org/#Background>—the address bar will update and the browser will scroll to the Background section. Here's the point: no reload actually occurs. And yet, the browser behaves as if you clicked on a standard link: the URL in the address bar changes and the Back button sends you back from whence you came (at least for some browsers; more on portability later).

You can probably see where this is going. We want Ajax Apps to change the page URL without forcing a reload, and it turns out that changes to fragment identifiers do exactly that. The solution, then, is for our script to change only the fragment component. We could do this by munging the `window.location.href` property, but there's actually a more direct route: `window.location.hash`. So when a state change is significant enough, just adjust the URL property:

```
window.location.hash = summary;
```

Throughout this pattern, *summary* refers to a string representation of the current browser state—at least that portion of the state that's worth keeping. For example, imagine that you have an Ajax interface containing several tabs: Books, Songs, and Movies. You want to assign a Unique URL to each tab, so the summary will be the name of any of these tabs. If there were other significant variables, they would be integrated into the summary string. To keep the fragment synchronized with the current state, you need to identify when state changes occur and update the fragment accordingly. In the following tab example, you can do this by making the `onclick` handler set the hash property in addition to changing the actual content:

```
songTab.onclick = function() {  
    window.location.hash = "songs".  
    // ... now open up the tab  
}
```

So now the URL changes from <http://ajax.shop/> to <http://ajax.shop/#songs>. That was easy enough, but what happens when you mail the URL to a friend? That friend's browser will open up <http://ajax.shop> and assume *songs* is an innocent, old-fashioned, in-page link. It will try to scroll to a *songs* element and won't find one, but it won't complain about it

either; it will simply show the main page as before. We've made the URL change *after* a state change, but so far we haven't built in any logic to make the state change *after* a URL change.

State and URL need to be in sync. If one changes, the other must change as well. Effectively, there's a mini-protocol specific to your application that defines how state and URL map to each other. In the preceding example, the protocol says "The fragment always represents the currently open tab." Setting the fragment is one side of the protocol; the other side is reading it. After your friend opens *http://ajax.shop/#songs*, and after his browser unsuccessfully tries to scroll to the nonexistent *songs* element, we need to read the fragment and update the interface accordingly:

```
window.onload = function() {  
    initializeStateFromURL();  
}  
  
initializeStateFromURL() {  
    var initialTab = window.location.hash;  
    openTab(initialTab);  
}
```

Good. Now you can mail the link, bookmark it, and link to it from another page. In some browsers at least, it will also go into your history, which means the Back button will work as expected. For a simple implementation, that's good enough.

However, the solution still isn't ideal, because what will happen if the application is already loaded? The most likely situations occur when clicking the Back button to revert to a previous state and when retrieving a bookmark while already on that application. In both cases, the URL will change in the address bar, but the application will do absolutely nothing!

Remember, we're making the application manipulate fragments precisely because the browser doesn't reload the page when they change. When it comes to Back buttons and bookmarks and the like, we're talking about changes that are initiated by the user, yet the browser treats these with exactly the same inaction as when our script initiates them. So our onload won't be called and the application won't be affected in anyway.

To make Unique URLs work properly, we have to resort to something of a cheap trick: continuously polling the fragment property. Whenever the fragment happens to change, the browser script will notice a short time later and update state accordingly. In this example, we could schedule a check every second using `setInterval()`:

```
window.onload = function() {  
    initializeStateFromURL();  
    setInterval(initializeStateFromURL, 1000);  
}
```

In fact, we should only perform an action if the hash has recently changed, so we perform a check in `initializeStateFromURL()` to see if the hash has recently changed:

```

var recentHash = "";
function pollHash() {

    if (window.location.hash==recentHash) {
        return; // Nothing has changed since last polled.
    }
    recentHash = window.location.hash;

    // URL has changed. Update the UI accordingly.
    openTab(initialTab);

}

```

The polling adds to performance overhead, but we now have a URL setup that feels like the real thing.

One last thing on this technique. I mentioned earlier that you can set `window.location.hash` manually after state changes. Now that we've considered the polling technique, you can see that it's also possible to do the reverse: change the URL in response to user events, then let the polling mechanism pick it up and change state accordingly. That adds a delay, but one big benefit is that you can easily add a hyperlink on the page to affect the system state.

Here's a quick summary of the technique just described, which I'll call the "page-URL" technique:

1. Ensure that `window.location.hash` always reflects application state.
2. Create a function, e.g., `initializeStateFromURL()`, to set application state from `window.location.hash`.
3. Run `initializeStateFromURL` on startup and at periodic intervals.

But wait, there's more! I mentioned earlier that each URL goes into history, but only "for some browsers." IE is one of the outliers here, so the above technique will fail IE's history mechanism, and the Back button won't work as expected. The good news is there's a workaround for it. The bad news is that it's a different technique and won't work properly on Firefox. So if you can live with everything working but IE history, go with the hash-rewrite technique. Otherwise, read on....

The IE-specific solution relies on IFrames and on their source URLs, so I'll call it the "IFrame-URL" technique. While IE won't consider a fragment change historically significant, it *will* recognize changes to the source URL of an embedded IFrame. Each time you change the source URL of an IFrame, the entire page will be added to the history. Think of each page in the history as not just the main URL but the combination of main URL and URLs of any IFrames. It's the combination that must vary in order to create a new history entry.

So to get IE history working, we can embed a dummy IFrame and keep changing its source URL to reflect the current state. When the user clicks Back, the source will change to its previous state. In theory, the browser could keep polling the IFrame's source URL to

find out when it's updated, but, for some reason, reverted IFrames don't seem to provide the correct source URL. So a workaround is to make the actual IFrame body contain the summary of browser state, i.e., the string which was in the fragment identifier for the previous algorithm. Then, the polling mechanism looks at the IFrame body rather than its source URL.

For example, if the user clicks on the Songs tab, the IFrame source will change to `dummy.phtml?summary=Songs`. The body of `dummy.phtml` will be made (by the server) to contain this argument verbatim, i.e., the body of the IFrame will be "Songs." When the user later clicks somewhere else and then clicks the Back button, the browser will reset the IFrame's URL to `dummy.phtml?summary=Songs` and the body will once again be "Songs." The polling mechanism will soon detect the change and open the Songs tab.

Now we have history, but we haven't done anything about the URLs. So when the user clicks on the Songs tab, we not only change the IFrame source but also the main document URL using a fragment identifier like before. We also have to make sure that the polling mechanism looks at the URL as well as the IFrame source. If either has changed since last poll, the application state must be updated. In addition, the one that hasn't changed must also be updated, i.e., if the IFrame source has changed, the URL must be updated, and vice versa.

Here's a quick summary of the IFrame-URL technique:

- Embed an IFrame in your document.
- On each significant state change, update the IFrame's source URL using a query string (`dummy.phtml?summary=Songs`), and update the main page URL using a fragment identifier (`http://ajax.shop/#Songs`).
- Continuously poll the IFrame source URL and the main page URL. If either has changed, update the other one and update the application state according to the new value.

Note that this won't work on Firefox, because Firefox will include URL changes as well as IFrame changes in the history. Thus, each state change will actually add two entries to Firefox's history. As mentioned earlier, you'll probably need to implement both algorithms in order to fully support both browsers. A modified version of the second technique will do it as well. Also, be careful when using both approaches because there's a risk your polling mechanism will revert states too eagerly. If the user has begun changing a value and the polling mechanism kicks in, it's possible that the value will change back. If you separate input and output carefully enough, that can be avoided. A workaround is to suspend the polling mechanism at certain times. None of this is ideal, but a Unique URL mechanism is worth fighting for.

As a variant on changing the IFrame's source, Brad Neuberg—based on a hack by Danny Goodman—has pointed out that browsers will retain form field data (<http://codinginparadise.org/weblog/2005/08/ajax-tutorial-saving-session-across.html>). Instead of changing the IFrame source, you set the value of a hidden text field in the IFrame. That will

solve the history problem, but you'll still need to do something about the main page URL to give the page a Unique URL.

To summarize all of this, here are all the things we'd want from Unique URLs and an explanation of how each can be achieved in Ajax.

Bookmarkable, linkable, and type-in-able

`window.location.hash` makes the URL unique, so a user can bookmark it, link to it, and type it directly into her browser. `window.onload` ensures that the bookmark is used to set state. Polling or `IFrame` ensures that this will work even when the base URL (before the hash) is the same.

History and Back button

`window.location.hash` will change, and polling `this` property will let you detect changes to the URL in most browsers. A hidden `IFrame` is also required for portability; it can be polled too.

Back to Ajax App

After leaving an Ajax App, the user should be able to click the Back button and return to the final state just prior to signing off. The techniques in *Heartbeat* will support this behavior.

URL handling is one of the greatest complaints about Ajax, with myths abounding about how "Ajax breaks the Back button" and how "You can't bookmark Ajax Apps." That an Ajax App can include full page refreshes is enough to debunk these myths. But this pattern shows that even without page refreshes, Unique URLs are still do-able, even if somewhat complicated. This pattern identifies some of the current thinking on Unique URLs, but the problem has not yet been solved in a satisfactory manner. The best advice is to watch for new ideas and delegate to a good library where applicable.

Decisions

How will you support search engine indexing?

Search engines point "robot" scripts to a web site and have them accumulate a collection of pages. The robot works by scooting through the web site, finding standard links to standard URLs, and following them. It won't click on buttons or type in values as a user would, and it probably won't distinguish among fragment identifiers, either. So if it sees links to `http://ajax.shop/#Songs` and `http://ajax.shop/#Movies`, it will follow one or the other, but not both. That's a big problem, because it means an entire Ajax App will only have one search link associated with it, and you'll miss out on a lot of potential visits.

The simplest approach is to live with a single page and do whatever you can with the initial HTML. Ensure that it contains all of the info required for indexing, focusing on meta tags, headings, and initial content.

A more sophisticated technique is to provide a Site Map page that is linked from the main page and that links to all URLs you want indexed with the link text containing suitable

descriptions. There is one catch here: you can't link to URLs with fragment identifiers, so you'll need to come up with a way to present search engines with standard URLs even though your application would normally present these using fragment identifiers. For example, have the Site Map link to <http://ajax.shop/Movies> and configure your server to redirect to <http://ajax.shop/#Movies>. It's probably reasonable to explicitly check if a robot is performing the search, and if it is, to preserve the URL—i.e., when the robot requests <http://ajax.shop/Movies>, simply output the same contents as the user would see on <http://ajax.shop/#Movies>. Thus, the search engine will index <http://ajax.shop/Movies> with the correct content, and when the user clicks on a search result, the server will know (because the client is *not* a robot) to redirect to <http://ajax.shop/#Movies>.

Search engine strategies for Ajax Apps has been discussed in a http://www.backbase.com/#dev/tech/001_designing_rias_for_sea.xml: detailed paper by Jeremy Hartlet of Backbase. See that paper for more details, though note that some advice is Backbase-specific.

What will be the polling interval?

The solutions here involve polling either the main page URL, the IFrame URL, or both. What sort of polling interval is appropriate? Anything more than a few hundred milliseconds will cause a noticeable delay (though that's not a showstopper because users are used to delays in loading pages). From a responsiveness perspective, the delay should ideally be as short as possible, but there are two forces at work that will lengthen it:

Performance overhead

Continuous polling will have an impact on overall application performance, so consider 10 milliseconds the bare minimum. Somewhere between 10 and 100 milliseconds is probably the best trade-off.

Overeager synchronization

This is only a problem for the IFrame-URL approach and not the Page-URL approach. The IFrame-URL approach, as described in the Solution above, keeps Page URL, IFrame URL, and application state synchronized. Thus, there's the possibility that you will begin to change one of those things and it will "snap back" during a synchronization checkpoint. For example, you might begin changing application state and suddenly see it "snap back" because it's different than URL state. There are various pragmatic ways to avoid this sort of thing, but as long as you have to rely on polling, the problem might still arise for certain combinations of events. While far from ideal, a longer polling period will at least diminish the problem.

What state differences warrant Unique URLs?

In this pattern, I discuss "state changes" abstractly. When I say a "state change" should result in a new URL and a new entry in the browser's history, what kind of state change am I discussing? In an Ajax App, some changes will be significant enough to warrant a new URL, and some won't. Here are some guidelines:

- User preferences shouldn't change the URL—e.g., if a user opts for a yellow background, it shouldn't appear in the URL.
- When there's a particular thing the user is looking at—for example, a product in an e-commerce catalogue, or a general category—that thing is a logical candidate for a Unique URL.
- When it's possible to open several things at once, you'll probably need to capture the entire collection.

What will the URLs look like?

Unique URLs requires you to make like an information architect and do some URL design work. Possibly, you'll be controlling only the fragment identifier rather than the entire URL, but even the fragment identifier has usability implications. Here are some guidelines:

- Users sometimes hack URLs, which suggests that, in some cases, you might want to make the variable explicit. So instead of `/#Movies`, consider `#category=Movies` if you want to make the URLs more hack-friendly. Also, this format is more like a CGI-style URL, which may be more familiar to users. The downside is that you'll need a little more coding to parse such URLs.
- As mentioned earlier, search engine indexing is also a consideration, and the fragment identifier will sometimes be mapped to a search engine-friendly URL. Therefore, ensure that the fragment identifier is as descriptive as possible in order to attract searchers. For example, favor a product name over a product ID, since the name is what users are likely to search for.

Real-World Examples

PairStairs

Nat Pryce's PairStairs (<http://nat.truemesh.com/stairmaster.html>) is an Extreme Programming tool that accepts a list of programmer initials and outputs a corresponding matrix (Figure 17-19). The URL stays synchronized with the initials that are being entered—e.g., if you enter "ap ek kc mm," the URL will update so that it ends with `#ap_ek_kc_mm`.

Dojo binding library

Dojo Toolkit's `dojo.io.bind` (http://dojotoolkit.org/docs/intro_to_dojo_io.html) lets you specify (or autogenerate) a fragment-based URL as part of a web remoting call. After the response arrives, the application's URL will change accordingly.

Really Simple History library

Really Simple History (http://codinginparadise.org/weblog/2005_09_20_archive.html) is a framework that lets you explicitly manage browser history (Figure 17-20). A call to `dhtmlHistory.add(fragment, state)` will set the URL's fragment identifier to `fragment` and set

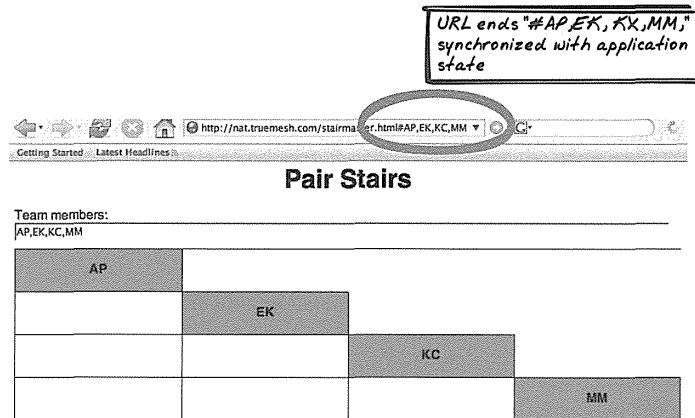


FIGURE 17-19. PairStairs

a state variable to state. A callback function can be registered to determine when the history has changed (e.g., the Back button was pressed), and it will receive the corresponding location and state. The library has recently been incorporated into the Dojo project.

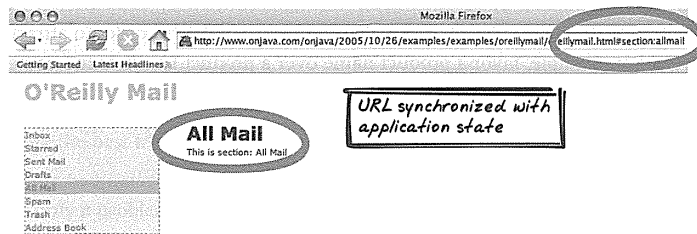


FIGURE 17-20. Really simple history demo

Code Refactoring: AjaxPatterns Unique URL Sum

This example refactors the Basic Sum Demo (<http://ajaxify.com/run/sum>) using both the Page-URL and IFrame-URL techniques. There are actually four implementations here:

- Unique URL Sum Demo (<http://ajaxify.com/run/sum/uniqueURL>) uses the Page-URL technique so that when state changes, the URL will change.
- Polling URL Sum Demo (<http://ajaxify.com/run/sum/uniqueURL/pollURL>) enhances the Unique URL Sum Demo by adding a polling mechanism, so that when the URL changes, the state will also change. Thus, the URL-state synchronization occurs in both directions.

- IFrame URL Sum Demo (<http://ajaxify.com/run/sum/uniqueURL/iFrame>) alters the Unique URL Sum Demo to use the IFrame-URL technique. In this version, the history will work correctly because the IFrame source is changed, but the main page URL is not actually updated.
- Full IFrame URL Sum Demo (<http://ajaxify.com/run/sum/uniqueURL/iFrame/updateURL/>) completes the IFrame-URL implementation by making the main page URL unique.

Unique URL Sum Demo

This first refactoring changes the fragment identifier (hash) when a sum is submitted. The fragment identifier always contains the main page state, which, in this case, is a comma-separated list of the three sum figures (e.g., #3,5,10):

```
function submitSum() {
    definedFigures = {
        figure1: $("figure1").value,
        figure2: $("figure2").value,
        figure3: $("figure3").value
    }
    hash =    "#" + definedFigures.figure1 + "," + definedFigures.figure2
            + "," + definedFigures.figure3;
    window.location.hash = hash;
    ajaxCaller.get("sum.phtml", definedFigures, onSumResponse, false, null);
}
```

This is only useful if the startup routine actually takes the fragment identifier into account. Thus, a new function, `setInitialFigures()`, is called on initialization. It inspects the fragment identifier, sets the parameters, and calls `submitSum()` to update the sum result:

```
function setInitialFigures() {
    figuresRE = /[0-9]*([0-9]*),([0-9]*),([0-9]*)/;
    figuresSpec = window.location.hash;
    if (!figuresRE.test(figuresSpec)) {
        return; // ignore url if invalid
    }
    $("figure1").value = figuresSpec.replace(figuresRE, "$1");
    $("figure2").value = figuresSpec.replace(figuresRE, "$2");
    $("figure3").value = figuresSpec.replace(figuresRE, "$3");
    submitSum();
}
```

Polling URL Sum Demo

The previous version will update the URL upon state change, but not vice versa. This demo rectifies the problem by polling the URL and updating state if the URL changes. The functionality for updating state from URL is already present in the `setInitialFigures()` function of the previous version. We simply need to keep running it instead of just calling it on startup. So, `setInitialFigures()` has been renamed to `pollHash()` and is run periodically:

```
window.onload = function() {
    ...
    setInterval(pollHash, 1000);
}
```


The function is the same as before, but with just one change: a `recentHash` variable is maintained to ensure that we only change state if the hash has actually changed. We don't want to change state more times than are necessary:

```
var recentHash = "";

function pollHash() {
    if (window.location.hash==recentHash) {
        return; // Nothing has changed since last polled.
    }
    recentHash = window.location.hash;

    // ... Same code as in previous setInitialFigures() function ...
}
```

IFrame Sum Demo

The previous versions won't work on IE because a fragment identifier change is not significant enough to get into IE's history. So here, we'll introduce an `IFrame` and change its source URL whenever the document changes.

The initial HTML includes an `IFrame`. This is the `IFrame` whose source URL we'll manipulate in order to add entries to browser history. It's backed by a PHP file that simply mimics the summary argument; the body of the `IFrame` always matches the source URL, so we can find the summary by inspecting the body. In fact, we should be able to just inspect the source URL directly, but there seems to be some bug in IE that fails to update the source property when you go back to a previous `IFrame` (even though it actually fetches the content according to that URL). Incidentally, the `IFrame` would normally be hidden via CSS but is kept visible for demonstration purposes.

```
<iframe id='iFrame' name='iFrame' src='summary.phtml?summary='></iframe>
```

Each time a sum is submitted, the `IFrame`'s source is updated:

```
function submitSum() {
    ...
    $("iFrame").src = "summary.phtml?summary=" + summary;
}
```

Instead of polling the URL, we're now polling the `IFrame`'s body, which contains the state summary. Remember that in this version, the main page URL is fixed—we're only making IE history work, and not yet making the application bookmarkable. Whenever we notice that the `IFrame` source has been changed, we assume that it's because the Back button has been clicked. Thus, we pull out the summary from the `IFrame` body and adjust the application state accordingly:

```
window.onload = function() {
    ...
    setInterval(pollSummary, 1000);
}

function pollSummary() {
```

```

summary = window["iFrame"].document.body.innerHTML;
if (summary==recentSummary) {
    return; // Nothing has changed since last polled.
}
recentSummary = summary;

// Set figures according to summary string and call submitSum()
// to set the result ...

}

```

The Back button will now work as expected on both Firefox and IE.

Full IFrame Sum Demo

This final version builds on the previous version by including Unique URLs for the main page so that you can bookmark a particular combination of figures. Note that this is an “IFrame URL” demo and won’t work properly on Firefox for reasons explained in the preceding Solution.

The page contains an IFrame as before, and since we’re now synchronizing the URL as well, there are actually three things to keep in sync:

- The application state, i.e., the three sum figures
- The page URL
- The IFrame URL

The algorithm is this: when one of these three changes, change the other two and update the sum result. To facilitate this, some convenience functions exist to read and write these properties:

```

function getURLSummary() {
    var url = window.location.href;
    return URL_RE.test(url) ? url.replace(/.*#(.*)/, "$1") : "";
}

function getIFrameSummary() {
    return util.trim(window["iFrame"].document.body.innerHTML);
}

function getInputsSummary() {
    return $("figure1").value + "," + $("figure2").value + "," + $("figure3").value;
}

function setURL(summaryString) {
    window.location.hash = "#" + summaryString;
    document.title = "Unique URL Sum: " + summaryString;
}

function setIFrame(summaryString) {
    $("iFrame").src = "summary.phtml?summary=" + summaryString;
}

```

The `updateSumResult()` is also straightforward:

```
function updateSumResult() {

    definedFigures = {
        figure1: $("figure1").value,
        figure2: $("figure2").value,
        figure3: $("figure3").value
    }
    ajaxCaller.get("sum.phtml", definedFigures, onSumResponse, false, null);

}

function onSumResponse(text, headers, callingContext) {
    self.$("sum").innerHTML = text;
}
```

Now for the actual synchronization logic. The first thing that might change is the application state itself. This occurs when the Submit button is clicked and will cause the IFrame and page URL to update:

```
window.onload = function() {
    self.$("addButton").onclick = function() {
        onSubmitted();
    }
    ...
}

function onSubmitted() {
    var inputsSummary = getInputsSummary();
    setIFrame(inputsSummary);
    setURL(inputsSummary);
    updateSumResult();
}
```

The other two things that can change are the IFrame source (e.g., if the Back button is clicked) or the URL (e.g., if a bookmark is selected). There's no way to notice these directly, so, as before, we poll for them. A single polling function suffices for both:

```
window.onload = function() {
    ...
    pollTimer = setInterval(pollSummary, POLL_INTERVAL);
}

function pollSummary() {

    var iframeSummary = getIFrameSummary();
    var urlSummary = getURLSummary();
    var inputsSummary = getInputsSummary();

    if (urlSummary != inputsSummary) { // URL changed, e.g., bookmark
        setInputs(urlSummary);
        setIFrame(urlSummary);
        updateSumResult();
    } else if (iframeSummary != inputsSummary) { //IFrame changed, e.g., Back button
        setInputs(iframeSummary);
    }
}
```

```

        setURL(iframeSummary);
        updateSumResult();
    }

}

```

One final point: The timer is suspended while the user edits a field in order to prevent the URL or IFrame source from reverting the application state:*

```

window.onload = function() {
    ...
    // Prevent iframe from triggering URL change during editing. The URL
    // change would in turn trigger changing of the values currently under edit,
    // which is why it needs to be stopped.
    for (var i=1; i<=3; i++) {
        $("figure"+i).onfocus = function() {
            clearInterval(pollTimer);
        }
        $("figure"+i).onblur = function() {
            pollTimer = setInterval(pollSummary, POLL_INTERVAL);
        }
    }
    ...
}

```

The application is now bookmarkable and has a working history. It's still not ideal, because if you unfocus an edit field, the figures will revert to a previous state. Also, there seems to be a browser issue (in both IE and Firefox), which means that the document URL isn't updated after you manually edit it. Thus, you can change the URL using a bookmark, but if you change it manually, it will not change again in response to an update of application state.

Alternatives

Occasional refresh

Some Ajax Apps perform page refreshes when a major state change occurs. For example, A9 (<http://a9.com>) uses a lot of Display Morphing (Chapter 5) and Web Remoting (Chapter 6) but nevertheless performs a standard submission for searches, leading to clean URLs such as <http://a9.com/ajax>. Usually, that's enough granularity, so no special URL manipulation needs to occur.

"Here" link

Some early Ajax offerings, like Google Maps and MSN Earth, keep the same URL throughout but offer a dynamic link within the page in case the user needs it. In some cases, there's a standard link to "the current page"; in other cases, the link is generated on demand. Various text is used, such as "Bookmark this Page" or "Link to this page." It's

* It's probably feasible to do something more sophisticated by tracking the history of URL, IFrame, and application state independently.

very easy to implement this but, unfortunately, it breaks the URL model that users are familiar with; thus users might not be able to use it. Time will tell whether users will adapt to this new style or whether it will fade away as Ajax developers learn how to deal with Unique URLs.

Want to Know More?

- “Uniform Resource Identifier (URI) Generic Syntax,” by Tim Berners-Lee, Roy Fielding, and Larry Masinter for the Internet Engineering Task Force (<http://www.gbiv.com/protocols/uri/rfc/rfc3986.html>)
- “Designing RIAs For Search Engine Accessibility,” by Jeremy Hartley of Backbase (http://www.backbase.com/#dev/tech/001_designing_rias_for_sea.xml)
- “Bookmarks and the Back Button in AJAX Applications,” by Laurens Holst of Backbase (http://www.backbase.com/#dev/tech/002_bookmarks.xml)
- “Ajax History Libraries,” by Brad Neuberg (http://codinginparadise.org/weblog/2005_09_20_archive.html)
- “Fixing the Back Button and Enabling Bookmarking for Ajax Apps,” by Mike Stenhouse (<http://www.contentwithstyle.co.uk/Articles/38/fixing-the-back-button-and-enabling-bookmarking-for-ajax-apps>)

Acknowledgments

The Solution is based heavily on Mike Stenhouse’s article (<http://www.contentwithstyle.co.uk/Articles/38/fixing-the-back-button-and-enabling-bookmarking-for-ajax-apps>).

Development Patterns

THE PATTERNS IN THIS PART ARE NOT “THINGS” YOU’D SEE IN AN AJAX APP—AS IN THE PREVIOUS parts—but processes you can use to aid development.* Diagnosis (Chapter 18) patterns help with troubleshooting and monitoring the health of an Ajax App. Testing (Chapter 19) patterns help with testing at various levels and are based on the agile practice of testing as the application evolves.

* If the idea of process patterns seems odd, feel free to treat them as general guidelines instead. There’s actually a relatively long history of patterns for software process and practice; for example, see Jim Coplien’s “A Development Process Generative Pattern Language” (<http://users.rcn.com/jcoplien/Patterns/Process/>).

CHAPTER EIGHTEEN

Diagnosis

IF THERE'S ONE CERTAINTY IN THE LIFE OF A DEVELOPER, IT'S BUGS. AJAX PROGRAMMING ISN'T ROCKET science, but you can still expect the odd moment of pain during development and maybe even in production. Fortunately, there are plenty of techniques, tools, and libraries available to help you isolate problems as they arise. This chapter breaks them into four categories.

Logging is just good old logging applied to the browser, where log messages can be shown in a popup, uploaded to the server, or shown in some browser-specific way. Likewise, *Debugging* is general-purpose debugging applied to JavaScript that is running within the browser.

The next two patterns are more Ajax-specific. *DOM Inspection* is about analyzing the dynamic state of the DOM. *Traffic Sniffing* is about watching traffic between browser and server.

Logging



Capture, Log, Message, Monitor, Record

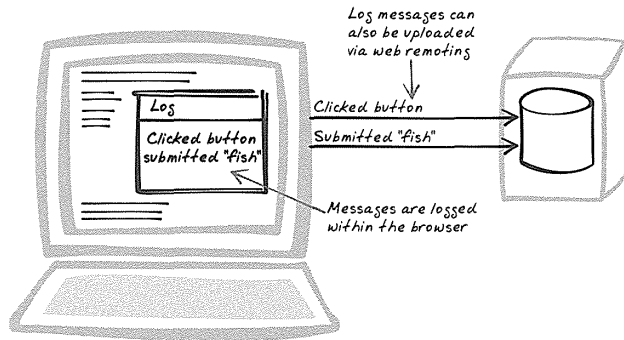


FIGURE 18-1. Logging

Developer Story

Dave's search engine keeps giving a blank result for certain queries. Fortunately, he previously added some Logging commands to see how the query is processed, so he makes the console visible and sets log level to verbose. After entering the query again, he now sees that one of the regular expressions is not matching the query as expected.

Problem

How can you track program process?

Solution

Instrument your JavaScript with log messages. Typically, the page contains a `div` element especially for Logging, and each significant action is appended there, e.g.:

```
$("#log").innerHTML += "User searched for " + query + "<br/>";
```

Even this simple implementation offers several benefits over the common alternative, debugging with an alertbox. Most importantly, the log element can be a *Popup* (Chapter 15): you can easily toggle visibility by dynamically switching CSS properties such as `display`, or even make it partly transparent. Further, Logging is unobtrusive—that is, there's no impact on application flow. Another benefit is that you have a history to consult if something goes wrong; there's no need to try replicating the problem.

Inspired by libraries such as Java's `log4j` (<http://logging.apache.org/log4j/docs/>), JavaScript Logging libraries usually provide some sort of filtering. Typically, a message is tagged with a priority level, e.g., "debug," or "info," or "error." You can then choose to only see messages above a certain level.

Logging impacts performance, not just in DOM manipulation but also in producing the messages themselves. You can end up with a memory problem as well unless some measure is taken to clear old messages, e.g., using a buffer to discard old messages.

Instead of Logging to a console on the page, some developers output messages to the browser status bar (using `window.status`), and Firefox developers also have the option of outputting to the browser console. However, this limits portability and requires some configuration (<http://www.make-believe.org/posts/05/10/24/0>).

Another approach that is gaining traction is to embrace Web Remoting (Chapter 6) and upload the data to a Logging web service.* This allows for permanent storage of the logs and can also be combined with server-side logs to paint a detailed picture of each interaction. Eric Pascarello, for example, has proposed the technique for usability testing (<http://radio.javaranch.com/pascarello/2005/11/01/1130878004388.html>). And log4js (http://jroller.com/page/stritti?entry=logging_in_javascript_log4js) supports an XMLHttpRequest-driven Logging strategy. The benefits of remote Logging must be balanced against concerns for users' privacy and consent.

Decisions

Will you log during production?

Most servers are configured to perform Logging during production as well as development, so should the browser log too? In the past, the answer was usually no. But Ajax makes the case for browser Logging more compelling for two reasons. Firstly, with more logic in the browser, there are more things that can go wrong and that need to be logged. Secondly, Web Remoting (Chapter 6) makes it possible to accumulate logs on the server in a completely unobtrusive manner. Still, remote Logging does consume application processing time as well as bandwidth, so you'll need to decide whether it's worth it, and if so, how much to log. In doing so, you'll also need to consider the user's privacy.

How will you change log settings between development and production?

In server-side *Logging* systems, log settings are usually altered by applying environment-specific filters. For example, in development, all messages are shown, whereas in production, only messages at information level and above are shown. But a familiar performance problem then arises: even though debug messages aren't being logged, the arguments must nevertheless be constructed, which takes time. A common solution is to include "if-then" statements—an unfortunate idiom that obscures the real point of code—to check the log level. Since JavaScript is generated by the server, you can do better than that: configure

* Ajax—more specifically, XMLHttpRequest—has been labelled a technique for "spying on users" (<http://www.devx.com/webdev/Article/28861>). However, techniques for remote *Logging* have been available for a long time. The main impact of Ajax is to increase rich activity in the browser, which might create further incentives to log user activity. See <http://www.softwareas.com/spying-on-users-with-xmlhttprequest>.

things so that log commands aren't even spit out in the first place. How you do this depends on the server-side environment. For example, a JSP developer could develop a JavaScript *Logging* tag whose implementation dynamically inspects the Logging configuration.

Tool Support

Lumberjack

Corey Johnson's Lumberjack (<http://gleepglop.com/javascripts/logger/>) is a JavaScript Logging framework that supports Logging at different levels. There's no setup required because it creates the Logging div itself, so you can immediately issue calls such as `Logger.info("User logged in.")`. The console is initially hidden, and you can toggle visibility with Alt-D.

fvLogger

David Miller's fvLogger (<http://www.fivevoltlogic.com/code/fvlogger/>) works similarly to Lumberjack. To use it, you just include a div with optional log level and make calls such as `error("No such record.");`.

log4js

log4js (http://jroller.com/page/stritti?entry=logging_in_javascript_log4js) is based more closely on log4j than other frameworks. In addition to various log levels, it supports pluggable Logging strategies. Logging strategies include: do nothing; log to pop-up window; and upload via *XMLHttpRequest Call*.

Mochikit

Bob Ippolito's Mochikit framework (<http://mochikit.com/doc/html/MochiKit/Logging.html>) has an API similar to those mentioned earlier and also adds features such as log listeners and a configurable message buffer. Interestingly, the standard way to launch the console is with a bookmarklet.

Code Example: Using Lumberjack

The Basic Ajax Pattern Reader (<http://ajaxify.com/run/reader/>) is refactored here to include Logging with Lumberjack (<http://gleepglop.com/javascripts/logger/>) (Figure 18-2). Lumberjack's *logger.js* is included, and the code is then instrumented to include log messages:

```
Logger.info("Received " + patternNames.length + " pattern names.");
...
Logger.debug("Received summary: " + summaryHTML.substring(0, 100) + "...");
...
Logger.info("Adding " + patternOption.value + " to playlist");
```

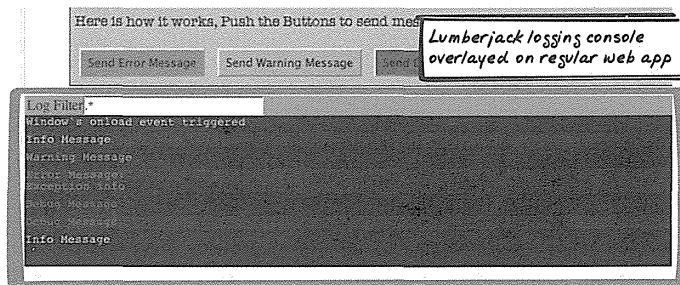


FIGURE 18-2. Lumberjack

Debugging



Break, Debug, Deduce, Fix, Inspect, Repair, Step, Test

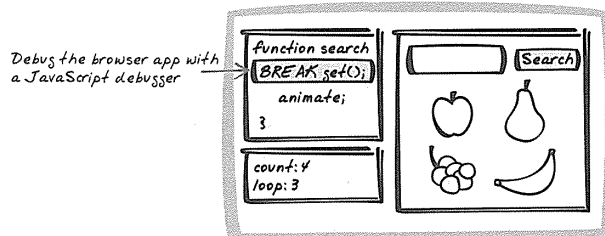


FIGURE 18-3. Debugging

Developer Story

Devi is scratching her head, wondering why a particular visual effect is so jerky; instead of growing gradually, the icon transitions from small to large in just a few, erratic, steps. Code analysis is getting her nowhere, so she fires up her JavaScript debugger, which allows her to step through the effect in her own time. After adding the box dimensions to a watchlist, she quickly diagnoses the cause as a number-rounding issue.

Problem

How can you diagnose errors and strange behavior?

Solution

Diagnose problems with a JavaScript debugger. JavaScript Debugging used to be as sophisticated as adding a few alert messages, but it has come a long way. A tool such as the Venkman debugger (<http://www.mozilla.org/projects/venkman/>), a popular Firefox extension, makes the point. Venkman has all the basic features you'd look for in a debugger of any

language: breakpoints, call stacks, step in/out/over, watches, and error and exception triggers. Also, an interactive session lets you type code for immediate execution as well as basic profiling support.

TO DEBUG OR NOT TO DEBUG?

That is the question many developers have asked over the years, with many deciding that there are often more productive things to do. The main problem with Debugging is that you have to invest time setting up breaklists and watches and thinking up strategies for diagnosis, yet all of that is thrown away after the problem's solved. The same problem might arise later on, and you'll have to set it all up again. Even if you can save breakpoints and watches—as Venkman lets you do—they are often too fragile to be useful, whereas if you can create a unit test to check the same thing, or introduce a log message to monitor state, you'll have created a permanent asset for yourself and your team.

For basic error reporting without a custom debugger, use Firefox's built-in JavaScript Console, and on IE, switch off the "Disable Script Debugging" options. You can also define `window.onerror` to show any errors that arise, as the event handler is notified of the error message, URL, and line number. You could then perform *Logging* or create an alert, e.g.:

```
window.onerror = function(message, url, lineNumber) {  
    alert("Error: " + message + ". At " + url + ", Line " + lineNumber);  
}
```

Tool Support

Venkman

Venkman (<http://www.mozilla.org/projects/venkman/>) is an open source Firefox extension with quite sophisticated *Debugging* support, as discussed in the preceding Solution (Figure 18-4). When you open the debugger, it shows a list of JavaScript files, which you can then open to set breakpoints.

Microsoft Script Debugger

Microsoft makes Microsoft Script Debugger (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sdbug/Html/sdbug_1.asp) available for no cost. It offers basic Debugging functions, e.g., breakpoints, a basic call stack, step in/out/over, and error triggers (Figure 18-5). While not as feature-rich as Venkman, it's still useful for investigating IE-specific bugs. The IE Blog (<http://blogs.msdn.com/ie/archive/2004/10/26/247912.aspx>) lists several options for IE Debugging.

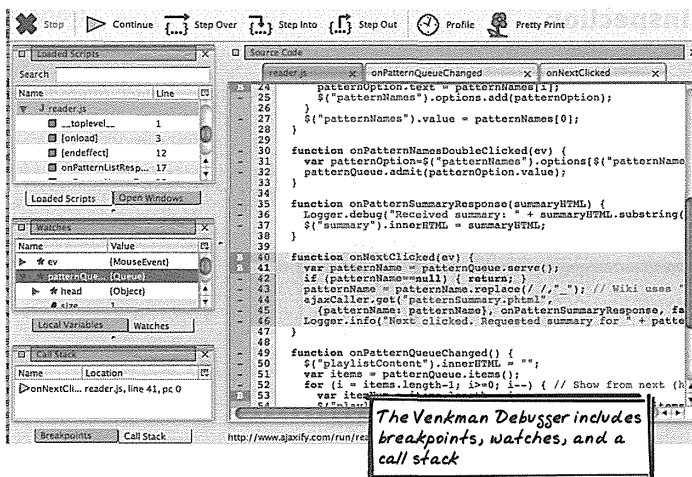


FIGURE 18-4. Venkman

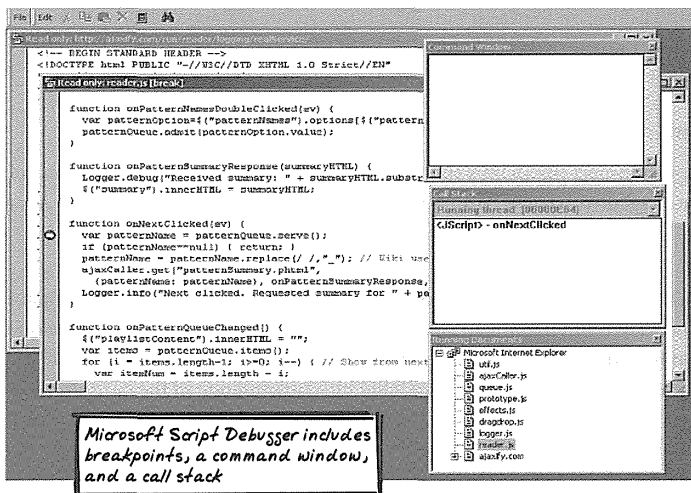


FIGURE 18-5. Microsoft Script Debugger

JavaScript HTML Debugger

JavaScript HTML Debugger (http://www.htmldebugger.com/javascript_debugger/javascript_debugger.asp) is a commercial tool from SplineTech. It's a standalone Windows application with functionality similar to Venkman but a greater emphasis on ease-of-learning.

DOM Inspection



DOM, Dynamic, Explore, Inspect, Interrogate, Investigate, Presentation, Render, Snapshot, State, View, Visualise, Visualize

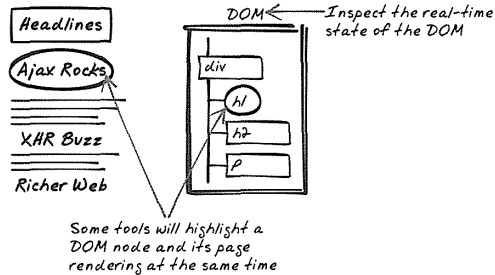


FIGURE 18-6. DOM Inspection

Developer Story

Devi keeps clicking on the Clear button, but the document text remains stubbornly unaffected. She fires up a DOM Inspection tool and discovers that the button's onclick event handler isn't present, which explains why nothing was happening.

Problem

How can you inspect the DOM's state at any time, taking into account DOM manipulations that may have occurred since the page was loaded?

Solution

Use a DOM Inspection Tool to explore the dynamic DOM state. The initial HTML for an Ajax App is often minimal and, in any event, likely to change over time due to DOM Manipulation. A DOM Inspection tool traverses the browser's current DOM model, rather than the initial HTML source, to build up a presentation of the current DOM. Many times, the tool is interactive, allowing you to drill down the hierarchy, search for keywords, and so on. All of this is very useful for checking assumptions and diagnosing problems, since many Ajax bugs arise because the programmer misunderstood the DOM state at a particular time.

Incidentally, DOM Inspection is useful beyond the fact that the DOM is dynamic. Even for static applications, an interactive exploration tool is often a good complement to the mound of text that makes up a complex HTML document.

Tool Support

Firefox DOM Inspector

Firefox ships with a DOM Inspector tool, accessible from the tools menu (Figure 18-7).^{*} The inspector is a popup window showing the DOM hierarchy on one side and facing a summary of the selected node. The summary shows node attributes such as registered event handlers, as well as CSS style information.

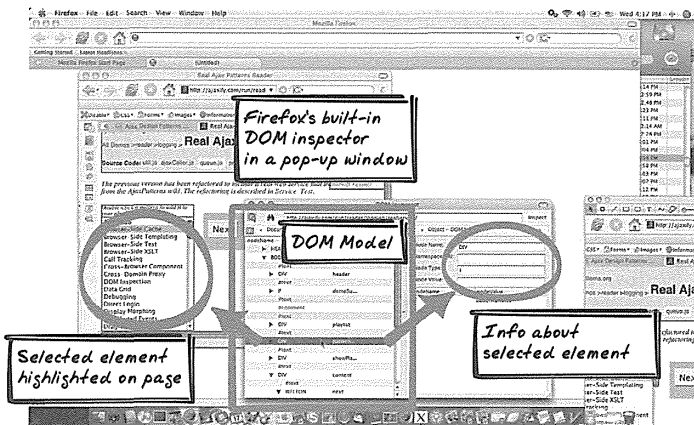


FIGURE 18-7. Firefox DOM Inspector

Firefox Web Developer Extension

Chris Pederick's Web Developer extension (<http://chrispederick.com/work/webdeveloper/>) is an invaluable Firefox toolbar for web development (Figure 18-8). Among its vast feature set are several commands especially relevant to DOM Inspection:

Display

These commands augment the page with DOM information, which makes a great alternative to exploring the DOM in a completely separate view. For example, *Display Topographic Information* alters the page to reveal the layout of DOM elements and *Display ID and Class Details* augments each element with its id and class attributes.

Outline

There commands—for example, *Outline Block Level Elements* or *Outline Table Cells*—draw outlines around elements of a particular type. In fact, you can define custom outlines to outline any element you want.

^{*} In Windows, you need to ensure that the Web Developer Tools option (<http://www.clagnut.com/blog/340/>) is selected during installation.

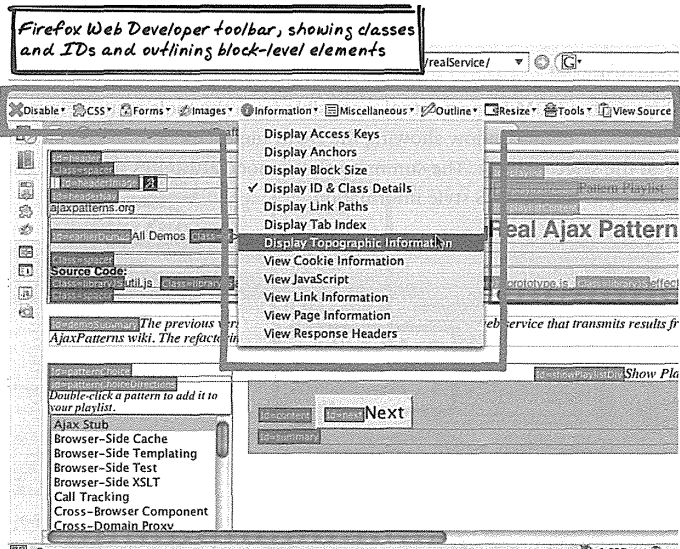


FIGURE 18-8. Web Developer toolbar

Form

These commands help expose form structure. *Display Form Details* augments the form with information about each control, while *View Form Information* opens up a summary page.

View Rendered Source

Jennifer Madden's View Rendered Source (<http://jennifermadden.com/scripts/View-RenderedSource.html>) is a Firefox extension that shows the entire DOM as an HTML document (Figure 18-9). Note that this is not the same thing as the good old View Source feature, because the rendered source reflects the DOM after any manipulation has taken place. The rendered source is shown in a pretty-formatted style, with colors and spacing to help convey the DOM structure. This is a free tool, and you can also purchase a slightly enhanced version for a small fee.

IE Developer Toolbar

Microsoft's free IE Developer Toolbar for IE6+ (<http://www.microsoft.com/downloads/details.aspx?FamilyID=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en>) has a number of DOM Inspection capabilities and is similar to the Firefox Web Developer Extension (Figure 18-10). It also includes a DOM Explorer similar to Firefox's DOM inspector (Figure 18-11).

You can find a list of IE DOM Inspectors at <http://blogs.msdn.com/ie/archive/2005/05/10/416156.aspx>.

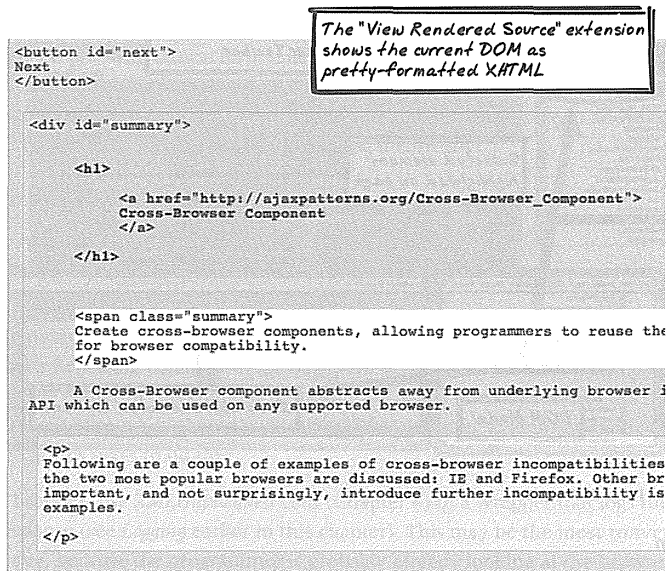


FIGURE 18-9. View Rendered Source

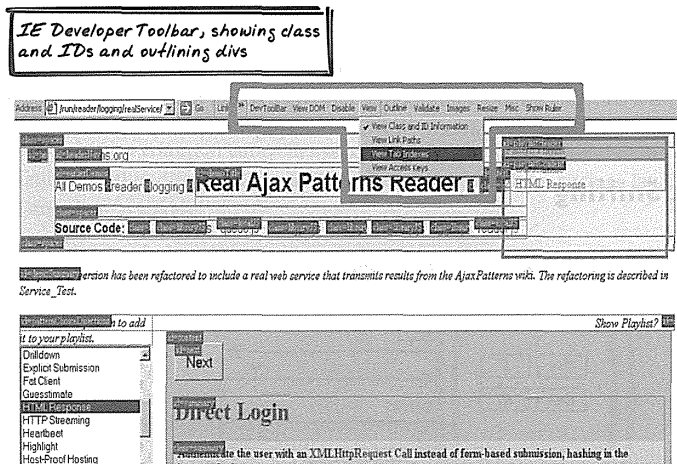


FIGURE 18-10. IE Developer Toolbar

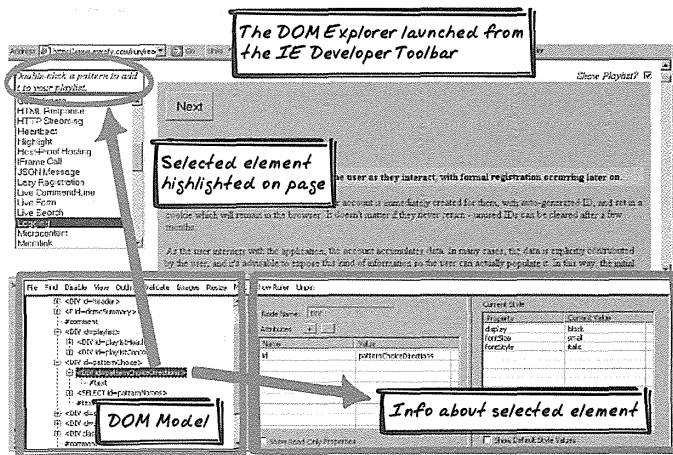


FIGURE 18-11. IE DOM Explorer

IEDocMon

Tim Tabor's IEDocMon (<http://www.cheztabor.com/IEDocMon/index.htm>) is an open source tool that extends IE to include an interactive view of the DOM.

Mouseover DOM Inspector

Steven Chipman's Mouseover DOM Inspector (http://slayeroffice.com/tools/modi/v2.0/modi_help.html) is a convenient bookmarklet that lets you explore the DOM by mousing over page elements. It works on all recent browsers except Safari.

Traffic Sniffing



Capture, Intercept, Log, Monitor, Network, Record, Sniff, Traffic

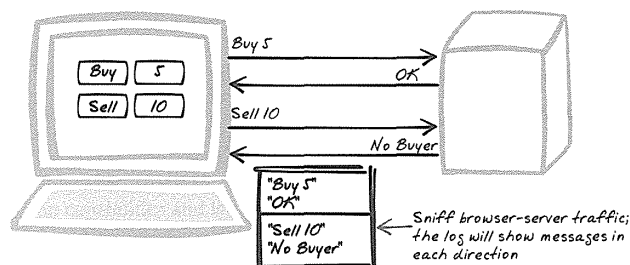


FIGURE 18-12. Traffic Sniffing

Developer Story

Dave hears a report that critical form data isn't being saved on the database. He turns a sniffer on and tries uploading the same data himself. The sniffer shows him an XML message being uploaded via `XMLHttpRequest`. When inspecting the message in an XML editor, he discovers that the JavaScript hasn't formatted it correctly.

Problem

How can you diagnose errors and strange behavior?

Solution

Diagnose problems by sniffing Web Remoting traffic. Many Ajax problems arise because Web Remoting (Chapter 6) messages are either wrong or are not sent at all. *Debugging* (see earlier in this chapter) and general *Logging* on either side of the network might help you infer what's being transmitted, but it's better to grab the traffic directly. There are various ways to do this:

- In the browser, use *XMLHttpRequest Calls* (Chapter 6) in a wrapper that logs the request and response (see *Logging* earlier in this chapter). This may be the most convenient technique, because the programmer is probably already looking at the browser. The downside is that you'll miss out on non-Web Remoting (Chapter 6) traffic, e.g., full page refreshes.
- Employ a generic network traffic monitor, filtering HTTP traffic to and from the server.
- Configure your browser to use a generic HTTP proxy and have the proxy log interesting traffic in either direction.
- In the server, intercept incoming and outgoing traffic for *Logging*. Server-side frameworks often have a common controller object which can be instrumented to capture information in either direction. In addition, interception facilities, such as Java's Servlet Filters, are often available.

Traffic Sniffing is a kind of *Logging* function. As such, you'll probably want similar filtering functionality, e.g., to see messages in only one direction, matching certain text, or to a particular URL.

Tool Support

XMLHttpRequest Tracing and XMLHttpRequest Debugging

Julien Couvreur has written two invaluable Firefox tools: *XMLHttpRequest Tracing* (<http://blog.monstuff.com/archives/000252.html>) unobtrusively logs traffic to the JavaScript console, while *XMLHttpRequest Debugging* (<http://blog.monstuff.com/archives/images/XMLHttpRequest-Debugging.v1.0.user.js>) is a much more powerful, interactive *Popup* tool that not only shows the messages but lets you set filters and configure the display (Figure 18-13). Both are Greasemonkey scripts (<http://greasemonkey.mozdev.org/>), so you'll need to install the Greasemonkey extension first.

Flangelib

Adam Vandenberg has noted that his personal *flangelib.js* library (<http://www.livejournal.com/users/piehead/tag/javascript>) contains an XMLHttpRequest wrapper to log traffic. While you're unlikely to use *flangelib*, it's mentioned here because some XMLHttpRequest wrapper libraries may eventually implement *Logging* functionality. The main barrier right now is the lack of an industry-standard Logging library.

Code Example: Using XMLHttpRequest Tracing

Following is the output from the XMLHttpRequest Tracing script during the initial load sequence for the Ajax Patterns Reader (<http://www.ajaxify.com/run/reader/logging/realService/>). First, the request is logged in the Firefox JavaScript console:

```
http://blog.monstuff.com/XmlHttpRequestTracing: [736] intercepted open (GET ,
patternList.phtml ,
true , undefined , undefined)
http://blog.monstuff.com/XmlHttpRequestTracing: [736] intercepted send (null)
```

Then, the result is displayed, showing response code (200) and content:

```
http://blog.monstuff.com/XmlHttpRequestTracing: [736] intercepted load: 200 Ajax
Stub|||Browser-Side
Cache|||Browser-Side Templating|||Browser-Side XSLT||| ....
```

Related Patterns

Logging

Traffic Sniffing in the browser is a form of *Logging* (see earlier).

CHAPTER NINETEEN

Testing

THANKS TO THE AGILE MOVEMENT, TESTING IS NO LONGER SEEN AS THE BIT AT THE END WHERE QA gives its stamp of approval. It's a core development technique, and if you're creating complex web apps, it's worth testing within and across all tiers.

Because of remoting, Ajax browser scripts usually rely on web services. But it's not much fun working on both sides at the same time, so *Simulation Service* explains how to create a "dummy" or "stub" web service to call from the browser. You can use it for tests or for prototyping. *Browser-Side Test* is simply a test of the JavaScript code sitting in the browser. The hardest thing is to exercise it in an automated fashion; the pattern also looks at that issue. *Service Test* involves no JavaScript, because it actually takes the place of the browser; an HTTP client is used to test the web service. *System Test* effectively simulates a user to exercise the application across all layers.

Simulation Service



Canned, Double, Dummy, Fake, Masquerade, Mock, Service, Shunt, Sim, Simulation, Stub

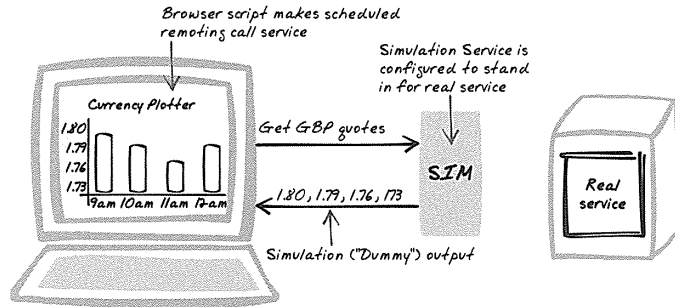


FIGURE 19-1. Simulation Service

Developer Story

Dave is beginning to work on a new email application. Eventually, the server will need to expose a mail API, so the browser can query for recent messages and upload new ones. However, that will require email expertise, and Dave wants to focus on the user interface in the browser for now. Thus, he hardcodes the responses he'd like in a "fake" server-side service. This allows him to proceed with user-interface development while his colleague takes care of the real server implementation.

Problem

How can you develop the browser script in parallel with the services it uses?

Solution

Develop the browser application against "fake" web services that simulate the actual services used in production. A Simulation Service is typically a hardcoded response that ignores any input parameters. So instead of:

```
$taxRate = calculateTaxRate($userId, $year);
print $taxRate;
```

a Simulation Service would just say:

```
print "0.1235";
```

It's also possible to do all this in a browser. If you use a library to perform Web Remoting (Chapter 6), you could make the implementation produce a hardcoded response, and ignore the server altogether.

In some cases, you might make the service more flexible; e.g., it can expose an operation to let the browser tell it what to output later on.

Simulation Service decouples the server and browser components of the application. Once the interface is defined, each can be developed in parallel. Moreover, this pattern paves the way for an agile process for building the application, based loosely on concepts such as Test-Driven Design (TDD) and lean manufacturing.

MOCKS, STUBS, SIMULATIONS

A Simulation Service is a kind of “stub.” In software testing, simple stubs are created to provide isolated tests. Instead of working with real components, the tested component works with simple stubs. Because the stubs are simple and stable, any test failures can then be attributed to the tested component. Furthermore, stubs will always yield the same result, so it’s easier to automate the test.

It’s important to distinguish between “mocks” and stubs, because the terms are often used interchangeably (<http://www.softwareas.com/mocks-stubs-dependency-injection-and-xmlhttprequest>). Like a stub, a “mock” simulates a real component. But for a mock, the simulation is only a by-product of its true purpose: to verify that it’s used correctly. Before a test, the mock is told what calls the component will make on it. As soon as that sequence is violated, the mock will complain and the test will fail. There are several potential applications for mocks in Ajax; one of those involves mock Services. The browser could start a test by telling a mock XMLHttpRequest wrapper what calls will soon be made. Then, the wrapper will throw an exception if that call isn’t actually made. “Mock Service” is too immature a concept to form a pattern at this stage, but it’s something that might emerge as Ajax developers focus on testing techniques.

Conventional web development is usually a bottom-up approach: first comes the database, then the data layer to access it, then the business components on top of that, and finally the user interface. This sequence might seem like the most logical thing to do, given the dependency relationships involved. But the problem is that the user interface is really the whole point of the application, so there’s a good argument for starting there instead. For a given feature, flesh out the user interface and use that work to drive the service interface. So instead of designing the server side first and forcing the browser script to work with whatever interface arises, let the needs of the browser script drive the interface design. Consequently, the service will not only contain the precise functionality required by the browser, but also expose it with the most suitable interface.

Starting development in the browser sounds like a good idea until you realize the browser depends on the service, which doesn’t yet exist! That’s where a Simulation Service comes in: it lets you define *what* the service offers, not *how* the service achieves it. As you develop

the browser script, you'll continue to come across refinements to the service—an extra operation here, a different response format there. In each case, changing the Simulation Service is usually as simple as editing the hardcoded response. Once you're confident with the requirements for the Service, you—or someone else—can go build the real thing.

Another application is in *Browser-Side Tests* (see later in this chapter). When testing a browser component, it's important to test only the component itself, and not the services it depends on. Imagine you're testing a browser component that renders news arriving from an RSS web service. An automated test could then interrogate the DOM and run a bunch of assertions to ensure it was formatted correctly. But if the component uses the real service, the resulting DOM will reflect the news of the hour. A human could manually verify it, but automation wouldn't be possible. A hardcoded response would eliminate that problem, allowing the assertions to be consistent with the simulated RSS. Furthermore, there's no dependence on the real implementation—no matter how complex the actual search algorithm, the rendering component will always be tested on its own merit.

Decisions

How will you switch between different services?

It's easy enough to create a Simulation Service, but how do you switch between the simulation and the real thing? And, how do you switch between different simulation services, since you might prefer one or the other for different tasks. Right now, there's no easy answer to that because there's no real support in the way of tools and libraries. Where the practice of Simulation Service is followed, most developers probably switch over to a real service in the most brute-force way possible: they edit the code. But doing so means there's a manual step involved in order to test or deploy. Here are a few ways to make the process more manageable:

- Keep the JavaScript pointing at the same service URL, but have your server-side build tool (e.g., ant) decide what implementation resides there.
- Build some configuration into the JavaScript to help it point to the correct service. This would be useful for an automated JavaScript-based test. You could even create a simple browser control to let the user choose which service is executed, which is helpful during development.
- Instead of hosting JavaScript in a static file, dynamically generate it from the server, so that variable URLs can be set on the fly.

Code Example: Simulation Service

The Ajax Pattern Reader (<http://ajaxify.com/run/reader/>) runs against a couple of *Cross-Domain Proxy* (Chapter 10) services. They fetch information from the AjaxPatterns.org web site and return them in a format easily parsed by the browser. For development purposes, the Basic version (<http://ajaxify.com/run/reader/>) uses Simulation Service.

The first service produces a *Plain-Text Message* (Chapter 9) containing the patterns in a pipe-separated list. A hand-created simulation (<http://ajaxify.com/run/reader/patternList.phtml>) is used instead of fetching the page from AjaxPatterns.org, parsing the HTML, and reformatting it.

```
Periodic Refresh|||One-Second Spotlight|||Slider
```

For comparison, the real service outputs a list that looks similar, but is much longer, and, of course, not hardcoded (<http://ajaxify.com/run/reader/logging/realService/patternList.phtml>).

```
Ajax Stub|||Browser-Side Cache|||Browser-Side Templating||| ... ||XML
Message|||XMLHttpRequest Call
```

Similarly, there's a simulation of the summary service (<http://ajaxify.com/run/reader/patternSummary.phtml?patternName=Slider>), whose job is to produce an *HTML Message* summarizing a given pattern. The simulation includes some rudimentary logic to make the response vary according to the input:

```
<? $patternName = urlDecode($_GET["patternName"]); ?>
<h1><?= $patternName ?></h1>
<? if ($patternName=="Periodic Refresh") { ?>
    With periodic refresh, it's all about the timer.
<? } else if ($patternName=="Slider") { ?>
    <p>The slider slides from left to right.</p>
    ...
<? } else {
    print ("Text goes here.");
}

?>
```

The output from the real service is in the same format (<http://ajaxify.com/run/reader/logging/realService/patternSummary.phtml?patternName=Slider>), but as with the list service, it's fetched from AjaxPatterns.org in real-time.

Related Patterns

Service Test

A *Simulation Service* is a good starting point for a real service. You might begin developing a *Service Test* (see later) against a *Simulation Service*, then evolve both into a real service and test.

Want to Know More?

- In Lean IT Processes terminology, the *Simulation Service* pattern facilitates a “Pull System,” where design is driven by customer demands rather than technical capability (<http://abc.trueemesh.com/archives/000093.html>).
- Categorization of Test Doubles, distinguishing between mocks, stubs, and so on. Under that categorization, the pattern here describes a “Fake” (<http://tap.testautomationpatterns.com:8080/Using%20Test%20Doubles.html>).

Browser-Side Test



Automated, IntegrationTest, JavaScript, Regression, UnitTest, TDD, Test

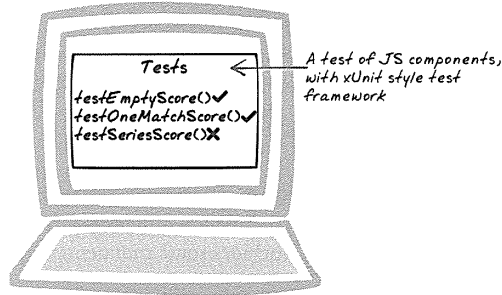


FIGURE 19-2. Browser-Side Test

Developer Story

Devi is about to embark on upgrading a blog reader and realizes she will be tinkering with the JavaScript RSS parser. The parser will need some refactoring, but fortunately there's already a JavaScript unit test for it. She proceeds to refactor the parser, ensuring the test still passes, and continues to enhance the test as she builds in further functionality.

Problem

How can you test JavaScript components?

Solution

Create automated tests of browser-side JavaScript components. This pattern is a mapping from automated testing in other languages, where frameworks like the xUnit series (<http://en.wikipedia.org/wiki/XUnit>—e.g., JUnit; see <http://junit.org>)—are now popular. Several frameworks do the same for JavaScript. As with other languages, you can unit test an individual JavaScript component (e.g., a function) you've created, and also perform integration tests and system tests that work across several objects. And also like other languages, you can accumulate tests into a test suite object, so it's possible to run a full regression test.

The test is usually some JavaScript that performs some task and includes assertion to check the state is as expected. Usually, JavaScript frameworks include a Test Runner that will run a given set of tests, then morph the DOM to show results in a table. You usually build an HTML page, with test code embedded, and perhaps some inputs to control the test; e.g., Start and Stop buttons.

THE XUNIT FRAMEWORKS

This pattern is heavily influenced by the “xUnit” testing frameworks (<http://en.wikipedia.org/wiki/XUnit>), which includes frameworks like JUnit for Java, PHPUnit for PHP, and NUnit for .Net. If you haven’t used any of these before, it’s worth learning the basic principles involved in xUnit-style testing, as these apply universally to all xUnit frameworks including the JavaScript ones. Kent Beck’s *Test-Driven Development By Example* (Addison Wesley) is an excellent overview.

So the test is automated insofar as the programmer’s not manually interacting with the application and verifying results. That’s good enough while developing, but the demands of continuous integration suggest we need to go beyond that. How do you allow tools like Make, Ant, or Rake start and stop a JavaScript test?

One way to coordinate a Browser-Side Test is to simulate browser behavior. Using a JavaScript engine, you can programmatically execute some JavaScript code. For example, Java programmers have the Rhino library (a standard component of Java 6). If there’s some DOM manipulation involved as well, an approach like *Browser Simulation*, discussed in *System Test* can be used. A second approach is to fire up a real browser using a shell command, specifying the test page as an argument. The test framework will then run in the browser, and the page must be configured to upload results to a service somewhere, which will expose them to the test process. The test process, upon detecting new results, shuts down the browser with an interprocess communication protocol or, failing that, kills the process.

Tool Support

Scriptaculous testing framework

The Scriptaculous framework (<http://script.aculo.us>) includes a unit-testing framework. It works in a similar manner as JsUnit frameworks (discussed next) and the nUnit frameworks they are modelled on. A Test Case is an object containing a sequence of methods beginning with “test”; e.g., `testStringMatches()`, and optional `setup()` and `teardown()`. The test is passed to a `testrunner` object. For each test method, the runner executes `setup()`, then the test method, and then `teardown()`. A library of assertions are available to verify state throughout the tests. As the tests proceed, the runner updates the UI to report on the results.

As with many Scriptaculous classes, the `Runner` object accepts an `options` argument. A particularly interesting option is `resultsURL`, which the runner will post results to. This is a facility that makes automated testing possible, as discussed earlier in the “Solution.”

Scriptaculous also contains some functions that let you simulate mouse and keyboard; see *System Test*.

JsUnit framework (Hieatt)

Edward Hieatt's JsUnit (<http://www.edwardh.com/jsunit/>) is a testing framework that supports standard xUnit-like functionality. Each test is actually an HTML page containing the test in a script, and the HTML page is passed as an argument to a test runner page. There's a demo page available (<http://www.edwardh.com/jsunit/runner/testRunner.html?testpage=www.edwardh.com/jsunit/runner/tests/jsUnitTestSuite.html>).

JsUnit Framework (Schaible)

Jorg Schaible's JsUnit (<http://jsunit.berlios.de/index.html>) works in a similar manner as the preceding frameworks.

Code Example: Using Scriptaculous Unit-Testing

The Basic Ajax Pattern Reader (<http://ajaxify.com/run/reader/>) uses a queue abstraction to represent the playlist. The queue itself was developed with a Scriptaculous test case (<http://script.aculo.us>; see Figure 19-3). The setup method establishes a new queue:

```
setup: function() { with(this) {  
    q = new Queue();  
}},
```

Reader Queue Test using Scriptaculous

7 tests, 34 assertions, 0 failures, 0 errors		
Status	Test	Message
passed	testEmptyQueue	2 assertions, 0 failures, 0 errors
passed	testAdmit	2 assertions, 0 failures, 0 errors
passed	testServe	3 assertions, 0 failures, 0 errors
passed	testSeveral	10 assertions, 0 failures, 0 errors
passed	testSequence	12 assertions, 0 failures, 0 errors
passed	testServeEmptyQueueReturnsNull	2 assertions, 0 failures, 0 errors
passed	testListener	3 assertions, 0 failures, 0 errors

The test runner displays results on the web page, using Display Manipulation

FIGURE 19-3. Scriptaculous Unit Test

An initial test method verifies the queue's initial state:

```
testEmptyQueue: function() { with(this) {  
    assertEquals(0, q.size);  
    var items = q.items();  
    assertEquals(0, items.length);  
}},
```

Subsequent methods examine the queue as it's built up:

```

testSequence: function() { with(this) {
    q.admit("first");
    q.admit("second");
    assertEquals(2, q.size);
    assert(util.membersEqual(["second", "first"], q.items()));
    ...
}},

```

All of this is embedded in an HTML file (<http://ajaxify.com/run/reader/queueTest.html>), which, if you visit it, will run the test and report the results. Here's the structure of the HTML file:

```

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Queue Test</title>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <script src="/run/Lib/js/util.js" type="text/javascript"></script>
  <script src="prototype.js" type="text/javascript"></script>
  <script src="unittest.js" type="text/javascript"></script>
  <script src="queue.js" type="text/javascript"></script>
  <link rel="stylesheet" href="test.css" type="text/css" />
</head>
<body>
  ...
  <!-- Log output -->
  <div id="testlog"> </div>

  <!-- Tests follow -->
  <script type="text/javascript" language="javascript" charset="utf-8">
    new Test.Unit.Runner({
      q: null,
      setup: function() { with(this) {
        q = new Queue();
      }},
      testEmptyQueue: function() { with(this) {
        assertEquals(0, q.size);
        var items = q.items();
        assertEquals(0, items.length);
      }},
      ...
    });
  </script>
</body>
</html>

```

Related Patterns

Service Test

A *Service Test* (see the next pattern) is a good way to complement a Browser-Side Test, since it focuses on the server-side interface rather than the browser application. Testing both is necessary to be confident that all aspects of the system are working.

Logging

Logging (Chapter 18) inside a test can help diagnose problems.

Service Test



Automated, JavaScript, Remoting, UnitTest, TDD, Test, Webservice, XMLHttpRequest

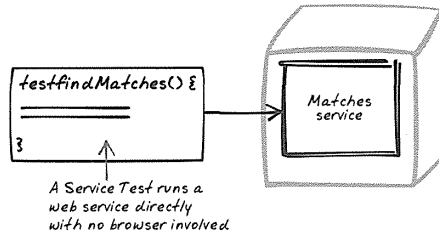


FIGURE 19-4. Service Test

Developer Story

Dave is creating a trading application. On the server, there are services to give quotes and accept orders. To help develop and test the web service, he builds a suite of Service Tests. One test, for example, submits an order and checks that the database registered it. These tests are automated and re-executed every time the project changes.

Problem

How can you test web services?

Solution

Build up automated tests of web services, using HTTP clients to interact with the server as the browser normally would. The browser-server interaction of an Ajax App is usually a sequence of *XMLHttpRequest Calls* (Chapter 6). That's fortunate, because testing the service is usually as easy as simulating the browser—passing a certain request and checking the response. Sometimes, verification might also involve checking what happened on the server, e.g., a database state change or a message to an external system.

The test itself can be built in any language and testing framework you like, because it's completely decoupled from both the browser and the server. The only practical constraint is that you should be able to run it from a script, to allow for continuous integration. Many languages provide an HTTP client library, so it's easy to build a test pass request to the server and make assertions against the response. Indeed, there are client frameworks like HTTPUnit (<http://httpunit.sourceforge.net/>) that are specifically for the purposes of testing web apps. They're typically used for testing conventional applications, where the response contains an entire page, but they work fine for the terse responses usually provided by web services.

A *RESTful Service* (Chapter 9), in particular, should be straightforward to test because it will usually output structured responses which are easily parsed; and there won't be any conversational state to set up and track. A service that outputs an *HTML Message* (Chapter 9) makes tests a bit more fragile; passing in 5+5 will always let you assert 10 as the *Plain-Text Message* (Chapter 9), but not always. The `<span class="strong">answer</class>` is `<span class="surprising">10<span>!!!` if an HTML Message is used.

Tool Support

Ruby Net::HTTP

Net::HTTP (<http://www.ruby-doc.org/stdlib/libdoc/net/http/rdoc/>) is an HTTP client library for Ruby, shipping with the standard distribution. It's used in the refactoring illustration below.

Jakarta HTTPClient

Jakarta HTTPClient (<http://jakarta.apache.org/commons/httpclient/>) is a richly featured HTTP client framework for Java.

PHP HTTPClient

Simon Willison's HTTPClient (<http://scripts.incutio.com/httpclient/>) is a basic HTTP client framework for Perl.

Code Example: Service Test

The Basic Ajax Pattern Reader (<http://ajaxify.com/run/reader/>) uses some *Simulation Services* to get information from the server. A Service Test was first created against the Simulation Service, and once the Simulation Service passed, the test was used to drive the design of the real services. Earlier, the Solution mentioned that the programming language of the HTTP client is completely independent of the server or browser languages. That's not just theoretical, since some languages (e.g., Ruby and Perl) are particularly well-suited to string manipulation. To prove the point, the example here uses Ruby's Net::HTTP library.

The service being tested is `patternSummary.phtml` (<http://ajaxify.com/run/reader/patternSummary.phtml>), a pattern text provider that accepts a pattern name and outputs its solution. The test case is very basic—it just requests the *Slider* pattern and checks that the output matches a simple regular expression:

```
require "test/unit"
require "net/http"

class PatternServiceTest < Test::Unit::TestCase

  def test_slider
    sliderResponse = grabPattern("Slider");
    regex = /^\\s*<h1><a\\ href=\\\".*?\\\">Slider</a><\\h1>\\s*\\
      <span\\ class=\\\"summary\\\">.+?</span>.+
      /mx;
```

```

        assert_match(regex, sliderResponse.body);
    end
    ...
    def grabPattern(patternName)
        session = Net::HTTP.new('ajaxlocal', 80);
        response = session.get("/run/reader/logging/realService/" +
                               "patternSummary.phtml?patternName=" + patternName);
        assert_equal("200", response.code);
        return response;
    end
end
end

```

Related Patterns

Simulation Service

You can sometimes start creating a test against a *Simulation Service* (see earlier), and gradually refactor it into a real service, while evolving the test as well.

Browser-Side Test

A *Browser-Side Test* (see earlier) is a good complement to a Service Test, since the former tests browser behavior while the latter ensures the server will support the browser in the correct way.

System Test



Automated, FunctionalTest, JavaScript, PerformanceTest, Regression, Robot, UnitTest, TDD, Test

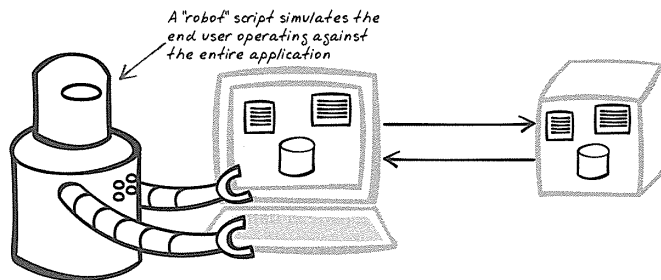


FIGURE 19-5. System Test

Developer Story

Devi has completed the first cut of a new Ajax App and wants to build an automated test of the whole thing. With the help of a testing application, she records a typical usage so it can be played back later. The software takes note of the final state, which will be used for verification during automated playback.

Problem

How can you test an entire Ajax App?

Solution

Build automated tests to simulate user behavior and verify the results. A System Test exercises a complete system in much the same way as a user would. The most common category of system test is a functional test—verifying the functionality—but system testing also encompasses qualities such as performance and robustness. While they are often conducted manually, with users clicking and mousing around, it's highly preferable to automate as much as possible in order to promote continuous integration.

Automated system tests remain somewhat difficult, but tools are improving. They can be broadly categorized as follows:

Browser Controller (Figure 19-6)

A specialized, “robot”-style testing tool will fire up the browser, then use the underlying operating system's API to create the effect of a user interacting with the browser. Several commercial controllers are able to record user actions for later playback.

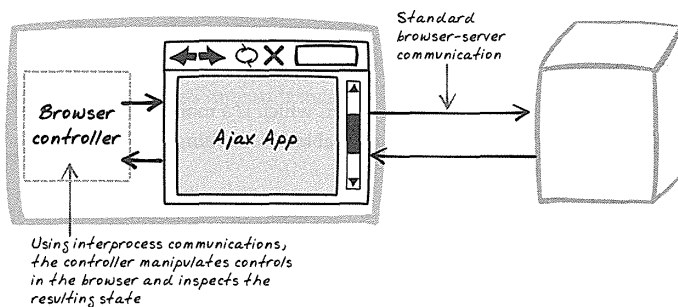


FIGURE 19-6. Browser Controller

Embedded Browser Script (Figure 19-7)

Run the application in a standard browser and embed a *Browser-Side Test* on the page. The test can use some JavaScript trickery to simulate user activity. Don't let the browser aspect put you off, because it's feasible to have a continuous integration tool like Ant coordinate *Browser-Side Tests*, as pointed out earlier in this chapter.

Browser Simulation (Figure 19-8)

The application is executed by a model of a real browser. Typically, the simulation pulls a page from a URL somewhere and builds up a DOM to represent it. The DOM can then be manipulated; e.g. the simulation allows a button click operation to occur or a form to be submitted. While several frameworks like this exist, the important issue for Ajax developers is how well they simulate JavaScript. JavaScript interpreters are implemented in several languages, so it's possible for *Browser Simulations* to incorporate them in order to simulate a realistic Ajax interaction.

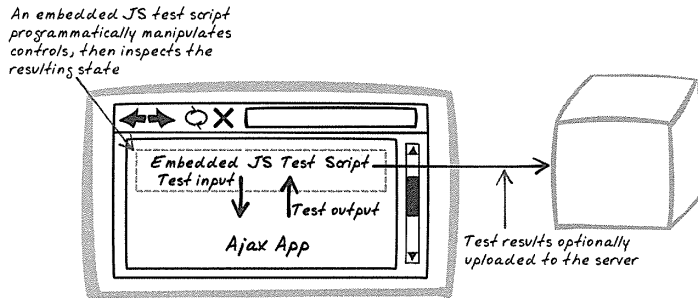


FIGURE 19-7. Embedded Browser Script

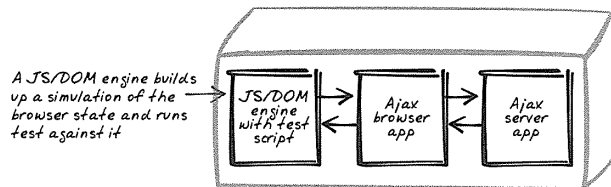


FIGURE 19-8. Browser Simulation

The first two approaches involve a real browser, which is a more realistic test, but it does imply that the test must run on a machine capable of launching the browser, or of somehow controlling it remotely.

System Tests require not just automation of user behavior, but also automation of result verification. As with other tests, results can be verified with a combination of assertions against the DOM and against server-side state.

Tool Support

GhostTrain

Scriptaculous's GhostTrain (<http://wiki.script.aculo.us/scriptaculous/show/GhostTrain>) is an *Embedded Browser Script* with recording capability. When you start the recording mode, a Scriptaculous test case begins to fill up inside the div. Each mouse and key action becomes a new line of the test case. Later on, you can add your own assertions about the application state. GhostTrain is a very promising work in progress.

Watir

Watir (<http://wtr.rubyforge.org>) follows the *Browser Controller* approach, enabling a Ruby programmer to script browser behavior and interrogate the DOM at any stage. You can easily integrate Watir commands into a standard test case—for example:

```
ie = Watir::IE.start("http://ajaxpatterns.org")
ie.button(:name, "Search").click
```

Watir is IE-specific right now.

Selenium

Selenium (<http://selenium.thoughtworks.com/>) is an open source tool by Thoughtworks developers, an *Embedded Browser Script* framework that works in IE, Mozilla, Firefox, and Safari. A JavaScript Selenium component is embedded inside the browser application under *test*, and it communicates with a server-side component. Interesting, there's flexibility on the location of the server-side component. It can be placed in the web server, to provide a quick summary in the web page being developed. Alternatively, it can be run as a separate server, where the server controls the browser (like a *Browser Controller* would), and the embedded browser component merely transmits server commands to the application.

HTTPUnit

HTTPUnit (<http://httpunit.sourceforge.net/index.html>) is an open source Java testing framework, built on JUnit, and specifically designed for testing web apps. It works as a *Browser Simulator*, converting an HTML page into a Java *HTMLPage* object that wraps a Java DOM model of the original HTML. HTTPUnit simulates JavaScript by delegating to the Rhino JavaScript engine, although support is incomplete.

DojoToolkit

Dojo Toolkit (http://dojotoolkit.org/docs/compressor_system.html) is a Java-based *Browser Simulator* that uses Rhino to let you run tests from an Ant task. Given the project's momentum and focus on Ajax, it's likely to evolve into a very solid product.

Code Example: Using Scriptaculous GhostTrain

The Ajax Patterns Reader (<http://ajaxify.com/run/reader/logging/realService/ghost/>) is refactored here to include Scriptaculous's GhostTrain (<http://wiki.script.aculo.us/scriptaculous/show/GhostTrain>), discussed earlier. You can hit Escape anywhere in the document to open it up, and then use the buttons to record a test case, which can then be played back (Figure 19-9).

To hook up GhostTrain, the Scriptaculous library files must be included (quite a lot to download, but keep in mind they don't need to be included in production):

```
<script type="text/javascript" src="/run/Lib/js/sc/prototype.js"></script>
<script type="text/javascript" src="/run/Lib/js/sc/controls.js"></script>
<script type="text/javascript" src="/run/Lib/js/sc/effects.js"></script>
<script type="text/javascript" src="unittest.js"></script>
<script type="text/javascript" src="ghosttrain.js"></script>
```

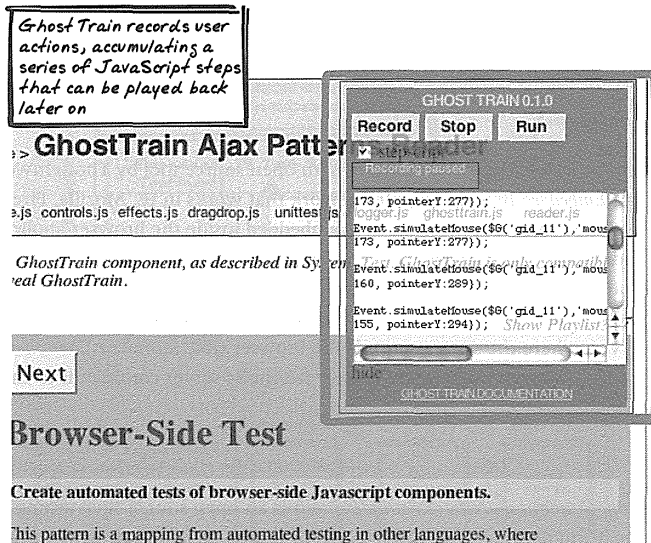


FIGURE 19-9. GhostTrain

The only other requirement is to register the GhostTrain object as a keypress handler. The handler will take care of any other initialization:

```

window.onload = function() {
    Event.observe(document, 'keypress', GhostTrain.intercept, true);
    ...
}

```

Related Patterns

Browser-Side Test

System Test frameworks that work as *Embedded Browser Scripts* are a form of *Browser-Side Tests* (see earlier).

Simulation Service

For integration tests, you can configure the server to run against a *Simulation Service* (see earlier). That way, you'll have tight control over server behavior, ensuring the final state is deterministic.

Want to Know More?

- A Summary of Functional Testing Frameworks (<http://www.softwareqatest.com/qatweb1.html#FUNC>).

Appendixes

T HERE ARE SEVERAL APPENDIXES.

Appendix A, *Ajax Frameworks and Libraries*, is a summary of frameworks and libraries supporting Ajax, ranging across many programming languages and licensing models.

Appendix B, *Setting Up the Code Examples*, offers instructions for setting up the tutorial and demo code on your system. Appendix C, *Patterns and Pattern Languages*, provides a background on patterns, pattern languages, and Ajax Patterns in that context. Finally,

Appendix D, *References*, lists the texts that are referenced throughout the book.

Ajax Frameworks and Libraries

AN EXPLOSION OF LIBRARIES AND FRAMEWORKS HAS BEEN ONE OF THE FORTUNATE CONSEQUENCES OF the Ajax phenomenon. They come in all shapes and sizes and in any licensing model you'd care to name. Throughout the Ajax Patterns, the importance of using frameworks and libraries has been emphasized—it's worth knowing the low-level details, but often not worth implementing them yourself. For this reason, the listing here makes a good companion to the Ajax Patterns.

The initial list was published back in June 2005, when I documented what seemed like an overwhelming 20 products. Maintained in an openly editable wiki, it's blossomed to 160-odd products, and doubtless many still to come—stay tuned to http://ajaxpatterns.org/Ajax_Frameworks for the latest updates.

Some of these were around long before Ajax was “Ajax,” but all are here because they support Ajax development. This section is a lightly edited snapshot of the wiki in January 2006—many of the descriptions come straight from project owners and users. It begins with two sections on pure JavaScript frameworks, where you have to provide your own server-side integration. Following that are frameworks for a large number of server-side environments, many including some JavaScript library API as well as a server-side API.

Note that there's a multilanguage section, since some frameworks address multiple server-side environments. Therefore, consult the multilanguage section in addition to the section for any particular server-side environment you're interested in.

People getting into Ajax often ask me for a "quick-fix" framework to help get started. So at risk of offending many worthy projects, each section points out a few of the frameworks that have generated buzz. This is, of course, a subjective measure, and it is one of popularity more than intrinsic value; do investigate further for any serious development work. In addition, look at the patterns, many of which discuss specific frameworks.

JavaScript Multipurpose Frameworks

These are pure-JavaScript libraries used to achieve a range of functionality in the browser—usually in a manner that protects you from dealing with browser compatibility issues, as described in *Cross-Browser Component* (Chapter 12). The Prototype framework has been very popular for general JavaScript infrastructure, and many frameworks mentioned here build on it. At a higher level, Dojo and Scriptaculous are probably the two best known products here and are good options if you're looking for a multipurpose library. Along those lines, Mochikit and OpenRico have also been popular as well. In the commercial space, there's been a lot of interest in Backbase and Tibco General Interface, though the products here are so varied, it really depends where you look.

AjaxFace (from 2005)

AjaxFace (<http://www.vertexlogic.com>) is an Ajax, JavaScript UI framework.

- High-level JavaScript API for building UI
- Open architecture for updating and retrieving data from the backend
- Automatic synchronization of data and UI
- Easy extensibility
- High performance
- Easy customization of look-and-feel through well-defined stylesheets
- Online demo (<http://www.vertexlogic.com/demo.html>)
- Commercial license by VertexLogic

Backbase (from 2003)

Backbase (<http://www.backbase.com>) is a comprehensive browser-side Ajax framework with cross-browser support. The Backbase Ajax framework is compatible with any server-side platform (e.g., Java, .NET, PHP, or Coldfusion).

- Free community edition for noncommercial/noninstitutional use
- Advanced tooling for .NET developers—Visual Studio 2005 plugin
- Advanced tooling for Java developers—JSF taglib/Eclipse integration

- Over 80 Ajax widgets/components at Backbase Explorer (<http://www.backbase.com/demos/explorer/>)
- Active Ajax developer community at Backbase DevNet (<http://www.backbase.com/#dev/home.xml>)
- Online Ajax demos/starter kits at Check Out (<http://www.backbase.com/#home/pulldown/demos.xml>)
- Commercial license by Backbase (formed in 2003)

Bindows (from 2003)

Bindows (<http://www.bindows.net>) is a software development kit (SDK) that generates highly interactive Internet applications with richness that rivals modern desktop applications using the strong combination of DHTML, JavaScript, CSS, and XML. Bindows applications require no downloads and no installation on the user side—only a browser is required (no Java, Flash, or ActiveX are used). Bindows is probably the leading object-oriented platform for developing Ajax applications.

The Bindows Framework provides you with the following:

- A class-based, object-oriented API
- A complete windowing system with a wide array of supported widgets including menus, forms, grids, sliders, gauges, and more
- The leading toolkit for developing zero-footprint SOA client-side applications
- Native XML, SOAP, and XML-RPC support
- Single-user to enterprise-level support
- Built-in support for Ajax

The Bindows development environment includes:

- Enterprise scale projects support
- Cross-browser, cross-platform support
- Server independent architecture
- Interoperability with new and existing resources
- Uniform development methodology
- Commercial license, by MB Technologies (based in GA, USA, with the main development center in Sweden, formed in 2002)

Dojo (from September, 2004)

Dojo (<http://dojotoolkit.org/>) offers comprehensive widget and browser-server messaging support:

- Extensive deployment support such as dependency-based packaging and compression of required libraries into a single download (see *On-Demand JavaScript* [Chapter 6]).

- Framework for creation of custom JavaScript widgets
- Library of pre-built widgets
- Solid drag-and-drop, effects, and generic animation support
- Browser-server messaging support such as XMLHttpRequest and other mechanisms
- Event management (see *Distributed Events* [Chapter 10])
- Support for bookmarkability and manipulating URLs in the browser
- Open source license (Academic Free License 2.1); led by Alex Russell (<http://alex.dojotoolkit.org/>) of JotSpot (<http://www.jot.com/>)

eBusiness Applications (EBA) Ajax Components (from 2002)

EBA (<http://www.ebusiness-apps.com/>) offers self-contained Ajax components enabling data entry and lookup:

- Library of prebuilt widgets supporting multiple backends (Java, PHP, ASP, and .NET)
- Excel-like data entry and grid tool (see *Data Grid* [Chapter 14])
- Multimode combo box enabling configurable livesearching (Smartsearching, autocomplete, compact, etc.; see *Live Search* and *Suggestion* [Chapter 14])
- Active user community (EBA Developer Forums at <http://forums.ebusiness-apps.com>)
- Online Ajax demos (EBA Grid at <http://developer.ebusiness-apps.com/technologies/webdevelopment/codeandcomponents/ebagrid/productdemos/griddemos.htm>)
- Commercial license by eBusiness Applications

Engine for Web Applications (from 2002)

Engine for Web Applications (<http://www.imnmotion.com/projects/engine/>) is a framework for building modular components for web applications and separating content from functionality:

- Configurable deployment
- Registry for storing, retrieving, and destroying objects that conform to the common interface
- Message service for intraobject publications and subscriptions
- Transaction service for asynchronous multiobject communication
- Task service for bootstrapping client-side modules of a web application and for creating dependency-based task lists; can load an XML-based task.
- Ability to easily create scoped and redistributable components in XML with automatic event hook-up and containment
- Store and retrieve form field values regardless of whether the form field still exists

- Monitor service for easily creating beacon and Ajax monitoring solutions
- API Documentation (http://www.imnmotion.com/projects/engine/api/engine_api.html)
- Demonstrations with source (<http://www.imnmotion.com/projects/engine/demonstrations/>)
- Open source license (custom license) by Stephen W. Cote (<http://www.imnmotion.com/>)

Framework for RESTful JavaScript (Freja) (from 2006)

Freja (<http://www.csscripting.com/freja>) is a JavaScript MVC (Model-View-Controller) framework:

- Geared toward single-screen, zero-latency web applications.
- Excellent scalability (much less server roundtrips).
- True MVC separation.
- True server-side/client-side code separation.
- Very light footprint; only a handful of methods to learn (the framework is 25 KB *uncompressed*).
- Based on Open Standards (XML/XSLT).
- Works with any modern web browser: IE, Firefox, Safari, and Opera.
- Plays well with other JavaScript toolkits and libraries (prototype, Scriptaculous, Dojo, etc.). Freja is not “yet another Ajax library”; it is a Ajax-based, high-level framework. It is not its purpose to provide drag-and-drop functionality or visual effects, but it will let you use your favorite JavaScript library to do so.
- See Tutorials (<http://www.csscripting.com/wiki/index.php?title=Freja#Tutorials>) and Documentation (http://www.csscripting.com/wiki/index.php?title=Freja_Documentation).
- Open source license (CC-GPL).

Mochikit (from 2005)

Mochikit (<http://mochikit.com/>) is a highly documented and well-tested suite of JavaScript libraries that will help you get your work done fast. Authors borrowed good ideas from Python, Objective-C, etc., and adapted them to JavaScript.

- MochiKit.Async: manage asynchronous tasks
- MochiKit.Base: functional programming and useful comparisons
- MochiKit.DOM: painless DOM manipulation API
- MochiKit.Color: color abstraction with CSS3 support
- MochiKit.DateTime: “What time is it anyway?”
- MochiKit.Format: string formatting goes here

- MochiKit.Iter: itertools for JavaScript; iteration made HARD, and then easy
- MochiKit.Logging: we're all tired of alert()
- MochiKit.LoggingPane: interactive MochiKit.Logging pane
- MochiKit.Visual: visual effects
- Open source license (MIT license or Academic Free License, v2.1) by Bob Ippolito

OpenRico (from May, 2005; based on earlier proprietary framework)

OpenRico (<http://openrico.org/demos.page>) is a multipurpose framework with support for Ajax infrastructure and user interaction.

- An XMLHttpRequest response can be routed to one or more callback operations, DOM object, or JavaScript object
- Easy drag-and-drop
- Ajax animation such as scaling and transitions (and presumably the increasingly common idioms such as progress indicators and fading technique)
- "Behaviors," which is essentially a widget library
- External tutorial by Yonah Russ of Mirimar (<http://www.mirimar.net/mailbrowser/>)
- Builds on prototype library
- Open source license; grew from Sabre Airline Solutions internal product, by Bill Scott, Darren James, Richard Cowin, and others (<http://looksgoodworkswell.blogspot.com>)

Plex Toolkit

Plex Toolkit (<http://www.plextk.org>) is an open source feature-complete DHTML GUI toolkit and Ajax framework based on a pure JavaScript/DOM GUI toolkit and Ajax framework. Uses the almost identical markup language to PXML (Flex) embedded in ordinary HTML documents for describing the UI. Binding is done with JavaScript.

Pros:

- Full set of widgets such as datagrid, tree, accordion, pulldown menus, DHTML window manager, viewstack, and more (over 60 libraries)
- Markup driven (makes it easy to build the interface)
- Interface components can be easily themed with CSS
- Client side XSLT for IE and Mozilla
- Well-documented with examples
- Multiple remoting transport options such as XMLHttpRequest and IFrame (an RSLite cookie-based coming soon)

- Back button support (complete history management for all components)
- Support for YAML serialization

Cons:

- No animation framework
- Open source license (GPL, LGPL, or Artistic License; your choice) by Richard Hundt.

Prototype

Prototype (<http://prototype.conio.net/>) makes it easy to use object-oriented concepts like classes and inheritance within JavaScript. It also supports basic Ajax functionality such as web remoting.

- A project run in conjunction with Ruby on Rails, but can be (and certainly is) used independent of Ruby or RoR
- Open source license by Sam Stephenson

qooxdoo (from May, 2005)

qooxdoo (<http://qooxdoo.oss.schlund.de/>) is another ambitious framework with a broad range of UI and infrastructural features being developed.

Infrastructure:

- Portable abstraction of the DOM and event/focus management
- Debugging support
- Timer class for easy scheduling
- Getter/Setter support

UI:

- Widget framework and library of prebuilt widgets
- Layout managers
- Image caching and portable PNG transparency
- Open source license (LGPL) by various contributors

Script.aculo.us (from 2005)

Scriptaculous (<http://script.aculo.us>) builds on the Prototype library to provide a JavaScript with comprehensive Ajax coverage.

- Visual Effects (see *One-Second Spotlight* [Chapter 16]).
- Drag-And-Drop (see *Drag-And-Drop* [Chapter 15]).

- Unit-Testing (see *System Test* [Chapter 19]).
- Open source license by Thomas Fuchs and others. A project run in conjunction with Ruby on Rails, but can be (and certainly is) used independent of Ruby or RoR.

SmartClient (from 2000)

SmartClient (<http://www.smartclient.com>) from Isomorphic Software is a cross-platform Ajax RIA system.

Cross-browser foundation classes and services:

- JSON or XML programming
- Browser abstraction layer
- GUI component services
- Logging and debugging services
- Extensible GUI components
- Navigation and command controls
- Form controls and managers
- Flat, hierarchical, and multidimensional grids
- Containers and layout managers

Data and services stack:

- Client data caches and local operations
- Client data model managers
- Communication and protocol services

Visual development tools:

- Runtime console.
- Log viewer.
- Component inspector and editor.
- Admin console.
- Integrated developer reference.
- SmartClient Ajax applications run on Internet Explorer, Mozilla, Netscape, Firefox, and Safari web browsers, and on Windows, MacOS, Linux, and Solaris operating systems. Unlike other Ajax systems, SmartClient applications are fully functional even when ActiveX is disabled.
- See www.smartclient.com for a product brief, live demos, and downloadable SDK.
- Commercial license by Isomorphic Software.

ThyAPI (from end of 2004)

ThyAPI (<http://www.sf.net/projects/thyapi/>) is a library of *DHTML reusable components*. Ajax is only one of its features. Its main purpose is to allow the creation of full desktop-like applications in a browser. ThyAPI is a library based on DynAPI (<http://www.sf.net/projects/dynapi/>), and open source (LGPL).

- Made entirely in DHTML (JavaScript + CSS).
- Totally object-oriented and extensible.
- Follows XMLRPC protocol, for data exchange with the server. Plans to implement JSON-RPC.
- Interface definition entirely in CSS.
- Works in Mozilla Firefox (version 1.0+)(<http://www.mozilla.org>) and Internet Explorer (version 6+).
- A DataSource component, to simplify linking of JavaScript widgets with server methods.
- Cross-browser, cross-platform support (it inherits this characteristic from DynAPI).
- Aimed to provide RAD development of browser-based applications and reuse of custom-made widgets.
- Open source license (LGPL).

TIBCO General Interface (from 2001)

TIBCO General Interface (http://www.tibco.com/software/business_optimization/rich_internet_applications.jsp) is a mature Ajax RIA framework that's been powering applications since 2001. In fact, the framework is so mature, that TIBCO General Interface's visual development tools themselves run in the browser alongside the Ajax RIAs as you create them.

- Dozens of extensible GUI components.
- Vector-based charting package.
- Support for SOAP communication (in addition to basic HTTP and XML).
- Full visual development environment.
- WYSIWYG GUI layouts.
- Step-through debugging.
- Code completion.
- Demo in Jon Udell's coverage at InfoWorld (<http://weblog.infoworld.com/udell/2005/05/25.html>).
- The next version of the product and many sample applications at the developer community site (<https://power.tibco.com/app/um/gi/newuser.jsp>).
- Commercial license by Tibco; more info at <http://developer.tibco.com>.

Interactive Website Framework (from May 2005)

Interactive Website Framework (<http://sourceforge.net/projects/iwff/>) is a project aiming to support the various aspects of Ajax infrastructure in the browser.

- Describes itself as a “framework for creating highly interactive web sites using JavaScript, css, xml, and html. Includes a custom xml parser for highly readable JavaScript. Essentially, all the plumbing for making Ajax-based web sites, as well as other common scripts.”
- Has a thread-safe XMLHttpRequest implementation.
- Contains a wrapper around the XML document, so you can make more readable code, instead of manual navigation:

```
var node = doc.groceries.frozen[0].pizza[0].size;  
var node = doc.documentElement.firstChild.firstChild.getAttribute("size");
```

- Open source license by Brock Weaver (<http://circaware.com>).

Zimbra AjaxTK

Zimbra (<http://www.zimbra.com/>) is a recently released client/server open source email system. Buried deep within this product is an excellent Ajax Tool Kit component library (AjaxTK) written in JavaScript. A fully featured demo of the product is available on zimbra.com that showcases the extensive capabilities of its email client. A very large and comprehensive widget library that was available only in commercial Ajax toolkits is now available to the open source community. Download the entire source tree to find the Ajax directory, which includes example applications.

Pros:

- Full support of drag-and-drop in all widgets. Widgets include data list, wizard, button, text node, rich text editor, tree, menus, etc.
- Build system uses Ant, and hosting is based on JSP and Tomcat.
- Very strong client-side MVC architecture based; architect is ex-Javasoft lead developer.
- Communications support for client-side SOAP, XMLHttpRequest, and IFrames.
- Support for JSON serialized objects and JavaScript-based XForms.
- Strong multibrowser capabilities: IE 5.5+, Firefox 1.0+, latest S.
- High-quality widgets have commercial quality, since this is a commercial open source product.
- Widget library is available as a separate build target set from the main product.
- Debugging facility is built into the library and displays communications request and response.

Cons:

- Does not currently support keyboard commands in menus and in-place datasheet editing.
- Does not support graceful degradation to IFrames if other transports are unavailable.
- Documentation is lacking, but a PDF whitepaper describing widget set and drag-and-drop is available.
- Open source license (Zimbra Ajax Public License ZAPL, derived from Mozilla Public License MPL) by Zimbra.

JavaScript Remoting Frameworks

As many of the Ajax Patterns establish, Web Remoting (Chapter 6) is error-prone, tedious, and broad in functionality, which makes it ideal to encapsulate in a library or framework like those covered here. The multipurpose frameworks covered earlier will do it too, but if you're looking for a standalone simple option, SACK is a popular choice. XMLHttpRequest is a good option for compatibility as it supports IFrames as well as XMLHttpRequest. A more feature-rich API is offered by Ajax Client Engine (ACE).

AjaxCaller (from May 2005)

AjaxCaller (<http://ajaxify.com/run/Lib/js/ajaxCaller.js>) is a basic, threadsafe wrapper around XMLHttpRequest that is mainly for Ajax newcomers packaged with the Ajax Patterns demos. See TestAjaxCaller (<http://ajaxify.com/run/testAjaxCaller>).

- RESTful calls to the server (GET/POST/PUT/DELETE) with plain-text or XML routed to a callback operation
- Aimed at Ajax newcomers—instead of optimizing performance or footprint, the library aims to be a readable code base and provides debugging support
- Open source license (Creative Commons) by Michael Mahemoff (<http://softwareas.com>), with some ideas from John Wehr and Richard Schwartz

Ajax Client Engine (ACE) (from December 2005)

Ajax Client Engine (<http://www.lishen.name>) is a powerful remoting wrapper.

- Object-oriented API
- Cross-browser support
- Flexible features: request options; request parameter validation; callback argument; callback option; callback timeout; tracing service; caching service; polling service; common callbacks; and exception handling
- Open source license (MIT) by Li Shen

AjaxGear (from November 2005)

AjaxGear (<http://www.ajaxgear.com>) is a simple and cross-platform Ajax toolkit.

- Enables a web browser to make asynchronous call to the web server without the need to refresh the whole page.
- Use the Ajax class to communicate with the web server.
- A complete web site is available for download to see AjaxGear in action.
- Use the *AjaxGear.PageEngine.js* file to learn how a web site can easily use the toolkit.
- Three objects are currently being developed as part of the AjaxGear toolkit—namely, progress bar, autocomplete, and form validator.
- Open source license (MIT) by Allan Spartacus Mangune (<http://www.allanmangune.com>) of ArchCommerce (<http://www.archcommerce.com>).

AJFORM (from June 2005)

AJFORM (<http://redredmusic.com/brendon/ajform/>) is an extremely easy to use Ajax framework. It is designed to be for entry-level Ajax coders. Its single purpose is to send data from any HTML form via XMLHTTP. The AJFORM framework provides you with:

- Three-step setup
- Automatic support for any HTML form
- Ability to implement with little coding knowledge
- Open source license (BSD)

HTMLHttpRequest (from 2005)

HTMLHttpRequest (<http://www.twinhelix.com/javascript/htmlhttprequest/>) also uses XMLHttpRequest and IFrames for improved compatibility.

- Tested and works in IE6, IE5.5, IE5, and IE4 for Windows; Mozilla for Windows; Opera7 for Windows; and in Safari and IE5 for the Mac
- Untested, but probably works in IE4 and Mozilla for the Mac; Opera for other programs; and Konqueror for Linux
- Open source license (LGPL) by Angus Turnbull of Twin Helix Designs (<http://www.twinhelix.com/>)

JSMX (from Aug 2005)

JSMX (<http://www.coldfusion-ajax.com>) is a very simple Ajax implementation for ColdFusion Developers (or any language that can easily build JavaScript Strings). This API does not return XML but String representations of JavaScript expressions.

- Using WDDX and the `toScript()` function within ColdFusion makes converting your ColdFusion objects to JavaScript a snap!

- Smaller packet sizes over the wire (JavaScript versus XML).
- Reduced latency due to less parsing of the responses.
- Parameters can be sent to the server in multiple formats, including strings, objects, and entire forms, without having to build extra logic to handle each type.
- API has no server-side components, which makes it more portable.
- Extremely simple syntax shortens the learning curve and speeds up development:


```
params = document.myForm;
http( "POST" , "remote.cfc?method=dosomething", my_callback , params );
```
- Open source license (Creative Commons Attribution-ShareAlike) by Todd Kingham.

LibXMLHttpRequest (from June 2003)

libXmlRequest (<http://www.whitefrost.com/reference/2005/09/09/libXmlRequest.html>) is a thin wrapper around XMLHttpRequest.

- Synchronous and asynchronous getXML() and postXML() methods
- Pooling of XMLHttpRequest objects
- Response caching
- XSL and XPath utilities
- setInnerXHTML utility for copying XML nodes into an HTML document
- Source protected by standard copyright by Stephen W. Cote (<http://www.imnmotion.com/>)

MAJAX (from August 2005)

MAJAX (<http://unips.sourceforge.net/devblog/?p=6>) is yet another very thin wrapper for XMLHttpRequest. The idea is to have a really simple interface to send and receive ASCII content.

- Provides handy callback interface for doing actions—e.g., upon server response
- Open source license (GPL or LGPL) by “Peter F”

RSLite

RSLite (<http://www.ashleyit.com/rs/main.htm>) is a thin wrapper around a highly cross-browser compatible transport layer comprised of a JavaScript image object request with a querystring parameter and a cookie response.

- A simple component released as part of Brent Ashley’s more comprehensive Remote Scripting work (see “JavaScript Remote Scripting (JSRS) (from 2000)” in the “Multi-language Ajax Frameworks” section)
- Open source license by Brent Ashley

Sack (from May 2005)

Sack (<http://twilightuniverse.com/resources/code/sack/>) is a thin wrapper around XMLHttpRequest.

- Caller can specify callback function or callback DOM object. With a callback DOM, the response text is pushed directly into the DOM.
- Open source license (modified MIT license) by Gregory Wild-Smith.

Subsys_JsonHttpRequest

Subsys_JsonHttpRequest (http://www.dklab.ru/lib/Subsys_JsonHttpRequest/) is an Ajax framework with XMLHttpRequest support and dynamic “script src” generation to emulate Ajax functional for old browsers that are not compatible with XMLHttpRequest.

- Open source license (LGPL 2.1 or later) by Dmitry Koterov

XHConn (from April, 2005)

XHConn (<http://xkr.us/code/javascript/XHConn/>) is a thin wrapper around XMLHttpRequest.

- Example: `new XHConn().connect("mypage.php", "POST", "foo=bar&baz=qux", fnWhenDone);`
- Open source license (Creative Commons Attribution-ShareAlike License) by Brad Fults

JavaScript Effects Frameworks

Fade Anything Technique (FAT)

FAT (<http://www.axentric.com/posts/default/7>) handles the Yellow Fade Technique (see *One-Second Spotlight* [Chapter 16]).

- Open source license (stated as Creative Commons in the page comments) by Adam Michela

Moo.fx (from Oct 2005)

Moo.fx (<http://moofx.mad4milk.net>) builds on a lite version of the Prototype library to provide an extremely compact effects package.

- Visual effects
- Cookie memory for effects
- 36 KB total file size
- Open source license (MIT) from Volerio Proietti

JavaScript Flash Frameworks

The *Richer Plugin* pattern points out how Flash can benefit Ajax development; these frameworks support Ajax-Flash crossovers.

AMFPHPKit

AMFPHP (<http://amfphp.sourceforge.net/>) is an open source implementation of the Flash Remoting framework.

- Fast, reliable, 100 percent free, and open source. With this new version, we strived to make a product as stable and full-featured as ColdFusion-based remoting (the reference implementation). You also get to use the wonderful NetConnection debugger, which shows you exactly what's being sent between the client and server. Remoting uses AMF, a very lightweight binary format that cuts the bulk out of packets, meaning data exchange is a lot faster than with XML.
- Open source license (GPL) created by Wolfgang Hamann and maintained by various developers.

Flash JavaScript Integration Kit

The Flash JavaScript Integration Kit (<http://www.osflash.org/doku.php?id=flashjs>) allows for the integration of JavaScript and Flash content.

- Enables JavaScript to invoke ActionScript functions, and vice versa.
- All major data types can be passed between the two environments.
- Open source license (modified Apache 1.1) by multiple open source Flash contributors.

Stream (from July, 2005)

Stream (<http://www.stormtide.ca/Stream>) is a bi-directional Ajax platform that brings Flash XML socket support to JavaScript. Also includes an extensible open source stateful socket server and client management platform.

- Plugin API (C# and JavaScript client libraries).
- No more polling (true server-pushed events; not emulated events through polling); see *HTTP Streaming* (Chapter 6).
- Ideal for chat, monitoring, and client interaction systems that would require polling.
- Sessions (per-session and per-page state tracking).
- Event monitoring (Interact with mouse events, including maximum fire rate throttling).
- Currently at the Alpha, technology preview, proof of concept stage. Download at <http://beta.stormtide.ca/files/StreamReleases/StreamAlphaGPL20050709.zip?source=ajaxpatterns>.
- Dual license open source (Commercial License and GPL) by StormTide Digital Studios. Full source available.

JavaScript XML Frameworks

These libraries support patterns such as *XML Message*, *Browser-Side XSLT*, and *XML Data Island*.

Google AJAXSLT (from June 2005)

Google AJAXSLT (<http://goog-ajaxslt.sourceforge.net/>) is a JavaScript framework for performing XSLT transformations, as well as XPath queries.

- Builds on Google Map work
- Open source license (BSD) by Google

Sarissa (from February, 2003)

Sarissa (<http://sarissa.sf.net>) is a JavaScript API that encapsulates XML functionality in browser-independent calls.

- Portable XMLHttpRequest creation
- Portable XPath queries
- Portable DOM manipulation
- Portable XSLT
- Portable serialization to XML
- Open source license (GPL 2.0 or LGPL 2.1, your choice) by various contributors

JavaScript Specialized Frameworks

These are a couple of useful libraries that don't fit well in other sections.

Drag-Drop

Walter Zorn's Drag-Drop library (http://www.walterzorn.com/dragdrop/dragdrop_e.htm) is a well-featured drag-and-drop library (see also *Drag-And-Drop* [Chapter 15]).

- Open source license (LGPL 2.1 or later) by Walter Zorn

Giant-Ass Image Viewer (GSV)

GSV (<http://mike.teczno.com/giant/pan/>) lets you place a massive image on the server and allow the user scroll around as if it was all stored locally (like scrolling around Google Maps; see *Virtual Workspace* [Chapter 15]).

- Open source license (custom) by Michael Migurski

Multilanguage Ajax Frameworks

Each library here integrates browser-side JavaScript with several server-side languages. Usually, an individual project will only need to integrate with one server-side language,

but the library includes several to make it applicable to a broader market. JSON has had a big impact, because it is as much a popular data format as it is a collection of open source libraries for remoting. SAJAX has also received considerable attention in this area, in part because it supports so many languages. CPAINT is popular as well.

Cross-Platform Asynchronous Interface Toolkit (CPAINT) (from May 2005)

CPAINT (<http://cpaint.sourceforge.net/>) is a true Ajax implementation and a JSRS (JavaScript Remote Scripting) implementation that supports both PHP and ASP/VBScript. CPAINT provides you the code required to implement Ajax and JSRS on the backend, while the returned data is manipulated, formatted, and displayed on the frontend in JavaScript. This allows you to build web applications that can provide near real-time feedback to the user. See *Ajax Stub* (Chapter 9).

- Supports both PHP and ASP
- Unified JavaScript file for all functions
- Supports both remote scripting and XML
- Supports both local and remote functions
- Single or multiple XMLHttpRequest objects
- Returns backend data as text or as a JavaScript XML/DOM document object
- Can support both POST and GET requests
- Backend proxy functions for accessing remote functions and data
- Tested with all major browsers
- Open source license (GNU, GPL, and LGPL) by various contributors

JavaScript Object Notation (JSON) and JSON-RPC

JSON (<http://www.crockford.com/JSON/index.html>) is a “fat-free XML alternative” and JSON-RPC (<http://www.json-rpc.org/>) is a remote procedure protocol, akin to XML-RPC, with strong support for JavaScript clients. See *JSON Message* (Chapter 9).

- Implementations (<http://www.json-rpc.org/impl.xhtml>) exist for several server-side platforms such as Java, Python, Ruby, or Perl.
- Individual packages and licenses for each platform. Original concept and JavaScript implementation by Douglas Crockford.

JavaScript Remote Scripting (JSRS) (from 2000)

JSRS (<http://www.ashleyit.com/rs/jsrs/test.htm>) routes calls directly from JavaScript into your server-side language and back out again.

- Initially developed back in 2000
- Known browsers: IE4+, NS4.x, NS6.x, Mozilla, Opera7, and Galeon

- Server-side support for ASP, ColdFusion, PerlCGI, PHP, Python, and JSP (servlet)
- Open source license (very flexible custom license) by Brent Ashley (<http://www.ashleyit.com/>)

Rialto: Rich Internet Application Toolkit

Rialto (<http://rialto.application-servers.com/>) is a cross browser JavaScript widgets library.

- Because it is technology agnostic, it can be encapsulated in Java/JSP/JSF, .NET, or PHP-graphic components.
- The widgets library includes forms, drag-and-drop, tree, data list with fix header and resizable columns, pop up, and splitter.
- Enables single-page interface-application development.
- Demo available (<http://rialto.application-servers.com/demoRialto.jsp>).
- Open source license (Apache) by multiple contributors.

SAJAX (from March 2005)

SAJAX (<http://www.modernmethod.com/sajax/>) routes calls directly from JavaScript into your server-side language and back out again. So, for example, calling a JavaScript method “x_calculateBudget()” will go the server and call an ASP/ColdFusion/IO/Lua/Perl/PHP/Python/Ruby calculateBudget() method, and then return the value in JavaScript to x_calculateBudget_cb().

- As described in *Ajax Stub* (Chapter 9), facilitates mapping from the JavaScript stub function to a backend operation.
- See a demo walkthrough in the “Real-World Examples” of Ajax Stub.
- Capable of stubbing calls to numerous server-side platforms: ASP, ColdFusion, IO, Lua, Perl, PHP, Python, or Ruby.
- Open source license (BSD) by various contributors.

ZK—Ajax/XUL Web Framework

ZK (<http://zk1.sourceforge.net/>) is an Ajax-based, event-driven, XUL-based, all Java framework designed to enable Web applications to have both rich user experiences and a simple programming model.

- With ZK, represent and manipulate RIA in XUL/HTML components that are all running at the server, as has been done for years in desktop applications.
- No JavaScript. No replication of business logic at the clients. No proprietary components. No compilation. ZK works with existent frameworks and technologies, such as JSF and portals. The pace of adapting ZK is all under your control.
- XUL-based components.
- Event-driven model.

- Server-centric processing.
- Script in Java and EL expressions.
- Demo available (<http://www.potix.com/zkdemo/userguide>).
- Open source license (GPL) by Potix Corporation.

C++ Ajax Frameworks

Wt

Wt (<http://witty.sourceforge.net/>) is a C++ widget library, which uses Ajax to render changes to the widget tree when available.

- API inspired by existing GUI libraries. It targets C++ application developers.
- The widget library makes complete abstraction of the underlying technology (JavaScript, HTML, Forms, CGI, Ajax).
- Documentation is at <http://jose.med.kuleuven.ac.be/wt/doc/index.html>.
- Open source license (Afferro General Public License) by Koen Deforche.

ColdFusion Ajax Frameworks

AjaxCFC

AjaxCFC (<http://www.robgonda.com/blog/projects/ajaxcfc/>) is a ColdFusion framework meant to speed up Ajax application development and deployment by providing developers seamless integration between JavaScript and ColdFusion. It also provides built-in functions that quickly adapt to any type of environment, security, and help overcoming cross-browser compatibility problems. It contains:

- ColdFusion components following the best practices of object-oriented programming and design patterns. Programming with AjaxCFC involves extending components and creating your own Ajax facades.
- Restricted open source license (free to use in any context but not redistributable) by Rob Gonda.

JSMX (from August 2005)

JSMX (<http://www.coldfusion-ajax.com>) is a very simple Ajax implementation for ColdFusion developers (or for any language that can easily build JavaScript Strings). This API does not return XML but String representations of JavaScript expressions.

- Smaller packet sizes over the wire (JavaScript versus XML).
- Reduced latency due to less parsing of the responses.
- Parameters can be sent to the server in multiple formats including strings, objects, and entire forms without having to build extra logic to handle each type.

- API has no server-side components, which makes it more portable (planned).
- Extremely simple syntax shortens the learning curve and speeds up development:


```
params = document.myForm;
http( "POST" , "remote.cfc?method=dosomething", my_callback, params );
```
- Open source license (Creative Commons Attribution-ShareAlike) by Todd Kingham.

.NET Ajax Frameworks

In the .NET world, all eyes are on Microsoft with their in-progress Atlas framework. It's not yet clear exactly what it covers, how portable it will be, and how it will affect the existing frameworks here. In any event, Michael Schwartz's Ajax.NET is one popular choice, and for a broader-scoped framework, the Monorail project—similar to Ruby on Rails—has a lot of interest too.

Ajax.NET for ASP.NET 1.x/2.0

Ajax.NET (<http://ajax.schwarz-interactive.de/csharpsample/default.aspx>) is an Ajax framework for ASP.NET 1.x/2.0.

- A basic Ajax library for ASP.NET that provides the very basic Ajax capability to make XMLHttpRequest callbacks. Does not have any Ajax-enabled controls or support for ViewState, etc.
- Open source license by Michael Schwartz.

Anthem.NET for ASP.NET 1.x/2.0

Anthem.NET (<http://anthem-dot-net.sourceforge.net/>) is Jason Diamonds' new version of a long-established Ajax library, now a SourceForge open source project.

- Anthem.NET provides a set of Ajax-enabled controls inherited from ASP.NET server controls, with most Ajax behaviors pre-built, avoiding lots of client JavaScript. It supports ViewState so that you can actually modify server controls (either Anthem controls or ASP.NET controls inside an Anthem control—e.g., panel) in your server-side code, in C# or VB without having to mess about in JavaScript. Still in its early days (DEC 2005) but promising!
- Open source license by Jason Diamonds.

AjaxAspects

AjaxAspects (<http://www.mathertel.de/AJAXEngine/>) is an engine that uses JavaScript proxies to call server-side web service methods.

- Use Ajax with web services.
- Built upon JavaScript client proxy methods for standard web services. (Just call a regular JavaScript method to call a method on the server.)
- Standard SOAP and WSDL is reused for the communication between client and server.

- No need for special implementations in web services.
- Multiple types, arrays and XML objects are supported as parameters and return values.
- Caching on the client and server.
- Queuing actions.
- Delaying actions.
- Many Ajax controls available that integrate in standard ASP.NET web forms.
- Supporting ASP.NET 2.0.
- Full source code available.
- Demo web site available (<http://www.mathertel.de/AjaxEngine/>).
- Supported by a blog in English (<http://ajaxaspects.blogspot.com/>) and German (<http://ajaxaspekte.blogspot.com/>).
- Open source license (Creative Commons license) by Matthias Hertel (<http://www.mathertel.de>).

Atlas (from late 2005)

Atlas (<http://www.asp.net/default.aspx?tabindex=7&tabid=47>) is Microsoft's in-progress Ajax project. ASP.NET Atlas is a package of new web development technologies that integrates an extensive set of client script libraries with the rich, server-based development platform of ASP.NET 2.0.

- Commercial (license terms unclear at time of writing) by Microsoft

Bitkraft for ASP.NET

Bitkraft (<http://www.tiggrbitz.com>) is a CLR-based (.NET) web framework that allows distributed web content to be created and served in a unique fashion. It is written in C# and compiles for operation under the Microsoft .NET Framework 1.1+ or the .Mono Framework, making it portable to almost any platform.

At its core, the Bitkraft framework extends the ASP.NET architecture to fully support JavaScript-based server callbacks using the XMLHttpRequest object as a transport layer in Ajax.

- Known browsers: IE5+, Firefox1+, and Netscape 6
- Server-side support for ASP.NET, Mono XSP, Cassini, and Apache (modMono) .NET Framework 1.1+
- Attribute driven
- Asynchronous and synchronous methods mapped directly from JavaScript proxy objects to the server
- Supports all .NET types and custom classes mapped to the JSON object on the client

- Uses JSON instead of XML
- Open source library by Wayne Lee-Archer

ComfortASP.NET for ASP.NET 1.1/2.0

ComfortASP.NET (<http://www.comfortasp.de>) is an approach that lets developers rely on pure ASP.NET programming while offering Ajax features. Internally, ComfortASP.NET uses Ajax (using DHTML, JavaScript, XMLHttpRequest) to implement these features—but the web developer implements only pure server-side ASP.NET!

- Automatically transfer only changes between ASP.NET postbacks, which leads to significantly faster response for low bandwidth connections and less overall HTML traffic (up to 90 percent, depending on new postback data)
- Reduce page reloads and use hidden postbacks instead
- Ajax/DHTML-like client experience (but without programming any Ajax/DHTML directly)
- Keep browser history free of postback entries (the Back button no longer irritates your users)
- Autodisable form and prevent user input while using postback or multiple form postbacks
- Control web client scroll position and focus from the server-side code
- Switch back at all times to original ASP.NET when you need it—even dynamically at runtime
- Open source license by Daniel Zei

MagicAjax.NET (from September 2005)

MagicAjax.NET (<http://www.magicajax.net/>) is an open source framework designed to make it easier and more intuitive for developers to integrate Ajax technology into their web pages, without replacing the ASP.NET controls and/or writing JavaScript code.

- Just make the initial setup, put the controls you want to Ajax-enable inside an Ajax-Panel, and you're done!
- Many other features also available if you really want the advanced features as well.
- Open source license (LGPL) by various contributors.

MonoRail (from May 2005)

MonoRail (<http://www.castleproject.org/index.php/MonoRail>) is a platform for developing C# web applications using templates instead of the webform-based approach.

- Functionally similar to Ruby on Rails
- Generates most or all of the JavaScript for widgets and animation in the browser
- Provides helpers that work with the Prototype JavaScript library (<http://prototype.conio.net/>)

- Functions on the .NET and .Mono platforms on both Windows and Linux
- Open source license by multiple developers

WebORB for .NET (from August 2005)

WebORB for .NET (<http://www.themidnightcoders.com/weborb/aboutWeborb.htm>) is a platform for developing Ajax and Flash-based rich client applications and connecting them with .NET objects and XML web services (see online examples at <http://www.themidnightcoders.com/examples>).

- WebORB includes a client-side library called Rich Client System (<http://www.themidnightcoders.com/rcs/index.htm>). The Rich Client System provides a simple one-line API to bind to and invoke any method on any .NET object or XML web service.
- Supports synchronous and asynchronous method invocations.
- Does not require any modifications on the server-side code, and there is no need for custom method attributes, special signatures, or argument types. Does not require design-time stub generation.
- Synchronous invocations return data from the same invocation (no callback needed). Asynchronous invocations rely on a callback.
- Any server-side method can be invoked synchronously or asynchronously.
- Client application can request special activation modes for the server objects. As a result, objects can easily become stateful without any special programming.
- Provides a special API for handling database query results—server code can return DataSets or DataTables and the client presents it as a special RecordSet JavaScript object. The object provides a way to retrieve column names as well as row data.
- Supports data paging. Client applications can retrieve data in pages, thus streamlining user experience.
- Supports all server-side argument types as well as return values—primitives, strings, complex types, arrays, native .NET collections, and remote references.
- Dual license (standard edition is free, professional is commercial) by Midnight Coders.

zumiPage

With zumiPage (<http://www.zumipage.com>), postbacks to the server are automatically captured on the client side and sent via an XMLHTTP mechanism.

- zumiPage requires no code changes to existing projects, so you can write code for ASP.NET as you always do, using the standard web controls and methodology. zumiPage will take care of the rest. Your web application should be smoother, faster, and more interactive.
- Works with ASP.NET 2.0 and 1.1.
- Fully compatible with Internet Explorer and Mozilla Firefox.

- Save up to 90 percent of the traffic.
- Easy to integrate with existing projects.
- Does not require any use of JavaScript.
- Fast, invisible XMLHttpRequest postbacks.
- Full control over returned controls on each postback.
- Wait behaviors (for server-loading times).
- Commercial license by Amir Leshem.

Java Ajax Frameworks

Java has always had many wide and varied frameworks. With Ajax, we've seen new ones emerge, and we've also seen add-ons for the earlier projects. Two popular libraries have been the Ajax JSP Tag Library and Direct Web Remoting.

AjaxAnywhere (from September 2005)

AjaxAnywhere (<http://ajaxanywhere.sourceforge.net>) returns any set of existing JSP, JSF, Struts, Spring, etc., components into Ajax-aware components without JavaScript coding. (see the Quick Start Demo at <http://ajaxanywhere.sourceforge.net/quickStart.html>).

- Does not break existing server-side MVC architecture.
- Less JavaScript to develop and maintain. Absence of commonly accepted naming conventions, formatting rules, and patterns makes JavaScript code messier than Java/JSP. It is extremely difficult to debug and unit-test in multibrowser environments. AjaxAnywhere eliminates all these complexities.
- Easy to integrate. AjaxAnywhere does not require changing the underlying application code.
- Graceful degradation. Switch whenever you need to between Ajax and the traditional (refresh-all-page) behavior of your web application. Your application can also support both behaviors.
- Open source license (Apache 2).

AJAX JSP Tag Library

The AJAX JSP Tag Library (<http://ajaxtags.sourceforge.net/>) is a set of JSP tags that simplify the use of Asynchronous JavaScript and XML (Ajax) technology in JavaServer Pages. This library eases development by not forcing J2EE developers to write the necessary JavaScript to implement an Ajax-capable web form. The tags are:

Autocomplete

Retrieves a list of values that matches the string entered in a text form field as the user types.

Callout

Displays a callout or pop-up balloon, anchored to an HTML element with an onclick event.

Select

Based on a selection within a drop-down field, a second select field will be populated.

Toggle

Switches a hidden form field between true and false and at the same time switches an image between two sources.

Update Field

Updates one or more form field values based on response to text entered in another field.

Open source license by multiple contributors.

AJAX Java Server Faces Framework

AJAX-JSF (<http://smirnov.org.ru/en/ajax-jsf.html>) will make any existing Java Server Faces applications use Ajax functionality. Most existing components can be used as is or easily converted to support Ajax.

- See a worked example called MyFaces JSF Tree, with table scroller and a tabbed pane as Ajax components (<http://smirnov.org.ru/myfaces-ajax/ajax.jsf>).
- Minimal differences from JSF specifications. This is proposal to the MyFaces project.
- Open source license (Apache Software License 2.0) by Alexander Smirnov.

Direct Web Remoting (DWR) (from 2005)

Direct Web Remoting (DWR) (<http://www.getahead.ltd.uk/dwr/>) is a framework for calling Java methods directly from JavaScript code.

- Like SAJAX, can pass calls from JavaScript into Java methods and back out to JavaScript callbacks
- Can be used with any web framework: Struts, Tapestry, etc.
- Follows Spring-like KISS/POJO/orthogonality philosophy
- Being incorporated into next WebWork release (<http://www.opensymphony.com/webwork/>)
- Open source license (Apache see <http://www.apache.org/LICENSE.txt>) by Joe Walker (<http://www.getahead.ltd.uk/sg/space/joe/>)

Echo 2 (from March 2005)

Echo 2 (<http://www.nextapp.com/products/echo2/>) allows you to code Ajax Apps in pure Java (see the demo at <http://demo.nextapp.com/InteractiveTest/ia>).

- Automatically generates HTML and JavaScript
- Coordinates messages between browser and server. Provides messaging in XML

- Can hand-write custom JavaScript components if desired
- Open source license (Mozilla Public License or GNU LGPL) by Next App, Inc. (<http://www.nextapp.com/>)

Guise

Guise (<http://www.guiseframework.com/>) provides an elegant server-side component architecture that doesn't require developers to write HTML or JavaScript (see the demo at <http://www.guiseframework.com/demonstration>).

- A true application framework, written completely in Java
- Allows developers to think in terms of component and events instead of HTML generation and DOM programming
- Provides controls, modal dialogs, flyovers, and more
- XHTML-compliant
- Flexible commercial license, with free full-featured development version available for immediate download by Global Mentor Inc.

ICEfaces

ICEfaces (<http://www.icesoft.com/products/icefaces.html>) is a standards-compliant extension to JavaServer Faces (JSF) for building and deploying rich Ajax applications.

- Smooth, incremental page updates with in-place editing and no full page refresh
- User context preservation during page update, including scrollbar positioning and user focus
- Asynchronous page updates driven from the application in real-time
- Fine-grained user interaction during form entry that augments the standard submit/response loop
- Commercial license by IceSoft

JSON-RPC-Java (from April 2004)

JSON-RPC-Java (<http://oss.metaparadigm.com/jsonrpc/>) is an Ajax RPC middleware that allows JavaScript DHTML web applications to call remote methods in a J2EE Application server.

- Transparently maps Java objects to and from JavaScript objects using Java reflection
- Handles marshalling/unmarshalling of arbitrarily complex nested data structures
- Lightweight protocol similar to XML-RPC
- Leverages the J2EE security model with session specific exporting of objects
- Open source license (LGPL) by Michael Clark and others

JSP Controls Tag Library (from December 2005)

JSP Controls Tag Library (<http://jspcontrols.sourceforge.net/>) provides the lifecycle for portlet-like JSP components. The library does not require a portal engine or other central controller. The components built with the Library can be used in any JSP-based application.

- Supports dual-mode components (non-Ajax and Ajax). Therefore, it works even with Netscape 4.
- In non-Ajax mode, components utilize synchronous HTTP request/response cycles via the Redirect-After-Post pattern.
- In Ajax mode, components are updated in place.
- Component reload in non-Ajax mode is completely automatic and transparent.
- Transition between Ajax and non-Ajax mode is undetectable when JavaScript is turned on or off.
- See the demo at <http://www.superinterface.com/jspcontrols/index.html>.
- Open source license (Apache 2.0) by Michael Jouravlev.

jWic

jWic (<http://www.jwic.de>) is a Java-based development framework for developing dynamic web applications with the convenience and familiarity of rich client-style programming.

- Component-based, event-driven programming model.
- Controls are rendered using templates (i.e., Velocity).
- Dynamically updates a control using Ajax mechanisms.
- Open source licence (Apache Licence, Version 2.0) by Florian Lippisch and Jens Bornemann.

Struts-Layout

Struts-Layout (<http://struts.application-servers.com>) is a tag library for Apache Struts that provides easy and fast interface creation. This is achieved by powerful tags that display panels, input fields, tables, treeviews, sortable lists, datagrids, popups, calendars, etc. With those tags customized to your own preference, some of the developers in your team don't have to write or even know HTML.

- The Struts-Layout tag library now allows to put Suggest fields on your web pages (<http://struts.application-servers.com/suggest/index.html>).
- Open source license (Apache 2) by Jean-Noel Ribette and others.

SWATO (from 2005)

SWATO (Shift Web Application TO...) (<http://swato.dev.java.net/>) is a set of reusable and well-integrated Java/JavaScript libraries that give you an easier way to shift the interaction of your web apps through Ajax.

- The server-side Java library can be easily deployed in all Servlet 2.3+-compatible containers.
- The client-side JavaScript library is based on the prototype that can help you code your JavaScript the object-oriented way.
- Uses JSON to marshal the data of your POJOs (Plain Old Java Objects) on the server side.
- Provides a simple interface for your JavaScript code to interact with the remote POJOs exposed to the client side such as RPC or cross domain access.
- Easy and flexible configuration using `<servlet>` and `<filter>` in *web.xml*. Comes with Spring integration.
- Comes with several reusable components (Auto-suggest Textbox, JS Template, JS Logger, etc.) that help you develop your web apps easier.
- Online demo available at <http://swato.throughworks.com>.
- Open source license (Apache) by Zhijie Chen.

Tacos Tapestry Components (from December 2005)

Tacos (<http://tacos.sourceforge.net>) is an Ajax engine/component library for Ajax/DHTML-based applications written in Java for the Tapestry web framework (<http://jakarta.apache.org/tapestry>).

- Written to be completely integrated with the Dojo JavaScript library (<http://dojotoolkit.org>), provides an extensive set of widgets and features.
- Requires little or no JavaScript knowledge to use, though any part of the process can be customized to your liking.
- Provides development tools such as a logging/debug console window for Ajax/other JavaScript-initiated requests that make writing Ajax applications much easier.
- Component library hosts extensive set of production quality code, with a built-in functionality that includes dialogs, floating windows, tons of effects, tree, autocompleter, Inline Editor, etc.
- Extensive form-based Ajax support includes client-side validation, field observers, partial form submissions, etc.
- Open source license (Apache) by multiple contributors.

ThinkCAP JX

ThinkCAP JX (<http://www.clearnova.com>) is designed to build industrial strength transactional business applications.

- Contains Visual Workbench.
- Based on over 20 open source projects, including Struts, Hibernate, JFreeChart, and many more.

- Using three-tier client/server development, this minimizes JavaScript by using a server-side generation library of over 150 methods.
- Handles complex Ajax updates using TransactionSafe. This guarantees no connection leaks and automates complex master-detail updates.
- Rich data-aware controls include Scrollable Updateable DataView Data Grid, Client Sorting, Outlook Bar, Accordion, Tab Panes, Trees, and Autocomplete.
- Smart data binding provides declarative binding to HTTP Request or Session Params.
- Client and Server Layout Management automatically directs component/Ajax call output to the appropriate default or named areas of a page.
- Dual license (GPL or commercial) by ClearNova.

WebORB for Java (from August 2005)

WebORB for Java (<http://www.themidnightcoders.com/weborb/aboutWeborb.htm>) is a platform for developing Ajax and Flash-based rich client applications and connecting them with Java objects and XML web services. (See online examples at <http://www.themidnightcoders.com/examples>.)

- See the earlier discussion “WebORB for .NET (from August 2005)” for more details.
- Dual license (standard edition is free, professional is commercial) by Midnight Coders.

WidgetServer (from 2004)

WidgetServer (<https://wiser.dev.java.net>) enables you to code Ajax applications in pure Java.

- Automatically generates HTML and JavaScript
- Coordinates messages between browser and server
- Can hand-write custom JavaScript and HTML/CSS components if desired
- Applications run as web, Swing, and Swing client/server Apps
- Dual license (higher-performance commercial version and lower-performance open source version) by Dirk von der Weiden

Lisp Ajax Frameworks

CL-Ajax

CL-Ajax (<http://cliki.net/cl-ajax>) directs JavaScript calls directly into server-side Lisp functions.

- The export function script is as follows:
(export-function #'my-function)
- Generates a JavaScript stub with arguments.
- Can callback to a JavaScript function or DOM object.

- May be integrated into SAJAX.
- Open source license (custom, very flexible, license) by Richard Newman (<http://www.holygoat.co.uk/>).

Perl Ajax Frameworks

CGI::Ajax—Export Perl Methods to JavaScript for Ajax

CGI::Ajax (<http://pjax.sourceforge.net/>) is a Perl module to make it trivial to write Ajax scripts using Perl. The module is object-based Perl, and creating a link from a JavaScript event to your Perl code is as easy as defining the JavaScript function name to Perl subroutine name mapping.

- Can be used to link any Perl code to an HTML event, including fetching other URLs (using LWP, for example)
- Nestable, so an HTML event can start a chain of Ajax events (e.g., `onClick= "do_this(); now_that(); finally_this()"`)
- Can handle GET or POST
- Not necessary to have your exported subroutines named in a specific way—name them what you want
- Very small overhead—allows you to keep thinking Perl and not get bogged down
- Active development community
- Open source library (Perl and Artistic license) by Brian Thomas

HTML::Prototype—Generate HTML and JavaScript for the Prototype Library

HTML::Prototype (<http://search.cpan.org/dist/HTML-Prototype/>) contains some code generators for Prototype, the famous JavaScript object-oriented library, and the *script.aculous* extensions.

- Basically similar to the Prototype helpers in Ruby on Rails
- Open source license (GPL and Artistic) by multiple contributors

PHP Ajax Frameworks

PHP frameworks tend to focus more on remoting than HTML generation. XOAD is one popular remoting library, as well as CPAINT and SAJAX (covered earlier in “JavaScript Multipurpose Frameworks”).

AJASON

AJASON (<http://ajason.sourceforge.net/>) is a PHP-based framework.

- Uses JSON (JavaScript Object Notation) to encode/decode data between server and browser
- Open source license by multiple contributors

AjaxAC (from April, 2005)

AjaxAC (<http://ajax.zervaas.com.au/>) encapsulates the entire application in a single PHP class.

- All application code is self-contained in a single class (plus any additional JavaScript libraries).
- Calling a PHP file or HTML page is very clean. All that is required is creating the application class, and then referencing the application JavaScript and attaching any required HTML elements to the application.
- Built-in functionality for easily handling JavaScript events.
- Built-in functionality for creating subrequests and handling them.
- Allows for custom configuration values, so certain elements can be set at runtime.
- No messy JavaScript code clogging up the calling HTML code—all events are dynamically attached.
- Easy to integrate with templating engine.
- Easy to hook in to existing PHP classes or MySQL databases for returning data from subrequests.
- Extensible widget structure to be able to easily create further JavaScript objects (this needs a bit of work though).
- Open source license (Apache 2.0) by Zervaas Enterprises (<http://ajax.zervaas.com.au/>).

Cajax

Cajax (<http://sourceforge.net/projects/cajax>) is a PHP object-oriented framework that creates and handles server-side requisitions through JavaScript remote calls. Some features are:

- Simple server-side programming
- Almost no client-side programming (less JavaScript writing is possible)
- Handler for server-side events
- A suggest handler like Google's
- Plugin for multiple selects (like country/state/city)
- Remote server calls directly from JavaScript, using an abstraction layer
- Plugin for submitting forms through XMLHttpRequest

- JavaScript caching
- Open source library (LGPL) by Thiago Pappacena

HTS Web Application Framework

HTSWaf (<http://www.htsdesign.com/index.php?&section=htswaf&page=index>) provides seamless client-to-server event handling.

- The HTS Web Application Framework is a PHP- and JavaScript-based framework designed to make simple web applications easy to design and implement. The framework implements a custom tag engine with an automated JavaScript and/or PHP event model and automates the transfer of data to and from the server.
- Visit the web site for more info and examples (<http://www.htsdesign.com/index.php?&section=htswaf&page=index>).
- Currently in demo stage, by Bert Rosell.

JPSpan

JPSpan (<http://jpspan.sourceforge.net/wiki/doku.php>) passes JavaScript calls directly to PHP functions.

- Heavily unit-tested
- Open source license

PEAR::HTML::Ajax

PEAR::HTML::Ajax (http://pear.php.net/package/HTML_AJAX) is a PHP and JavaScript Ajax library.

- Provides PHP and JavaScript libraries for performing Ajax (communicates from JavaScript to your browser without reloading the page).
- Offers object-oriented (OO) proxies in JavaScript of registered PHP or proxyless operations.
- Serialization of data sent between PHP and JavaScript is provided by a driver model. Currently JSON and Null encodings are provided.
- Open source license (LGPL) by multiple contributors.

Pipeline

The Pipeline framework (<http://livepipe.net>) consists of a simple client-side library and direct integration with the controller. It's main focus is on simplicity, it is not feature rich, and does not use XML. It favors returning HTML or Strings from the controller, or associative arrays that get translated from PHP into JavaScript.

- Open source license with commercial add-ons in development by Picora Pipeworks.

Symfony

Symfony (<http://www.symfony-project.com>) integrates a server-side Ajax layer together with helpers “à la” *script.aculo.us*, to provide an easy way to build Ajax applications in PHP.

- A step-by-step tutorial is available (http://www.symfony-project.com/tutorial/symfony_ajax.html) describing the creation of an interactive shopping cart. This demonstrates the tight integration of prototype, *script.aculo.us*, and the MVC model.
- The Advent Calendar initiative (<http://www.symfony-project.com/askeet>) illustrates how to build a real Ajax application in 24 short tutorials, complete with downloadable code. Interactive form submission, Folksonomy, and visual effects are some of the Ajax features of the askeet web site (<http://www.askeet.com>) they are building live. A must see.
- Open source license (MIT License), sponsored by Sensio.

XAJAX

XAJAX (<http://www.xajaxproject.org>) passes JavaScript calls directly to PHP functions.

- Use JavaScript stubs to call functions, object methods, and class methods in a PHP script.
- The response is created using the XAJAX response object, which provides numerous commands, such as assigning HTML to an element, displaying an alert, and more. Custom script output is also supported.
- All browser-specific code (such as using XMLHttpRequest) is abstracted, and sending various types of data to the server (including form submission via Ajax) is very easy to accomplish.
- It supports a number of configuration options to allow easy integration with existing web apps and frameworks.
- The JavaScript core is easy to understand and can be overridden to support more advanced JavaScript functionality.
- Well-supported via forums and a new wiki with documentation and tutorials.
- Open source license (LGPL) by J. Max Wilson and Jared White.

XOAD (formerly NAJAX)

XOAD (<http://www.xoad.org>) is a PHP-based Ajax/XAP object-oriented framework.

- It uses *JSON Messages* and native PHP serialized objects to communicate.
- Special attention has been paid to security.
- Supports server-side events (*Distributed Events* [Chapter 10]).
- Supports client-side events (XOAD Events).
- Supports server and client extensions.
- Supports HTML manipulation (extension).

- Supports caching (extension).
- Each class, method and variable is documented.
- Provides easy tutorials to get started using XOAD.
- Provides examples that demonstrate various functionality.
- Won the PHP Programming Innovation Award of August of 2005 (<http://www.phpclasses.org/winners.html>).
- Open source license (PHP 3.0 license) by Stanimir Angeloff.

Python Ajax Frameworks

CrackAJAX

CrackAJAX is a Python framework (http://www.aminus.org/blogs/index.php/phunt/2005/10/06/subway_s_new_ajax_framework) that does not require JavaScript skills.

- Open source license by Peter Hunt

Turbo Gears

Turbo Gears (<http://turbogears.org>) is a “mega-framework” combining many libraries and frameworks.

- Open source licence (MIT license) and multiple licenses from incorporated frameworks by multiple developers

Ruby Ajax Frameworks

Ruby On Rails

Ruby On Rails (<http://www.rubyonrails.org/>) is a general web framework with strong Ajax support.

- Rails was still in its early days when Ajax hype began, so Ajax may become increasingly important to the Rails framework.
- Generates most or all of the JavaScript for widgets and animation in the browser.
- Supports calling server side.
- Provides scheduling support.
- Open source license [MIT or similar (<http://wiki.rubyonrails.com/rails/show/License>) by David Heinemeier Hansson and contributors (<http://www.loudthinking.com/>).

Setting Up the Code Examples

A SINGLE PACKAGE CONTAINING ALL CODE FOR CHAPTER 2 AND THE AJAX PATTERNS DEMOS AS WELL IS at <http://ajaxify.com/run>. Inside there is a *INSTALL.txt* file containing installation instructions. These instructions are reprinted below, but if you have any trouble, please do check the electronic version as it may include updates.

INSTALLING PHP =====

* If you don't have it already, install PHP5 and integrate with Apache (or another web server). The demos rely on the PHP5 OO features, and won't work with PHP4 or below.

There are various installers available that will create an "AMP" (Apache, PHP, and MySQL) setup for you (You'll need MySQL for the wiki demo - see below). e.g. XAMPP for all major platforms (<http://www.apachefriends.org/en/xampp.html>). For Apple, consider <http://www.entropy.ch/software/macosx/php/>. For Linux, you may try the appropriate Redhat/Debian packages. Alternatively, try the compilation instructions at <http://dan.drydog.com/apache2php.html>.

SETTING UP AJAX DEMOS

=====

- * Unzip the package to a temporary location and copy run/, tutorial/, and records/ to the apache document root. Assuming the doc root is /apache/document/root:

```
cp run tutorial records /apache/document/root.
```

(Alternatively, if you have sufficient access, set up a new virtual host in apache's httpd.conf and point it to the root of the unzipped directory, ajaxdemos/).

- * Ensure the server can write to (the initially empty) records/ directory. The easiest (though not the most secure) way is:

```
chmod 777 apache/document/root/records
```

- * Open up run/.htaccess and follow instructions there to set the library path.

- * Finished! Visit <http://localhost/run/> and <http://localhost/tutorial>. If you can access <http://localhost/run/index.phtml> but not <http://localhost/run/>, read the following note.

NOTE ON CONFIGURATION: You hopefully won't need any configuration because run/.htaccess and tutorial/.htaccess contain the relevant settings. But some apache installations will ignore those settings (depending on the AllowOverride directive), so be ready to transfer some of the contents of run/.htaccess into httpd.conf. (run/.htaccess is a superset of tutorial/.htaccess.) Specifically, modify the existing DirectoryIndex directive to include index.phtml, and add a new type for phtml as in .htaccess. If the PHP include_path isn't working, you'll probably need to modify it in php.ini according to the setting in .htaccess.

SETTING UP THE WIKI DATABASE

=====

NOTE: The wiki demo and its refactorings are the only demos that require a database. The setup isn't particularly difficult, but if you don't want to mess with MySQL, you can still run all the other demos.

- * Install MySQL if necessary
- * Prepare for database access.
 - * Enter mysql as power user, e.g.:

```
mysql -u root
```
 - * Create "ajaxify_wiki" power user with password "ajaxify_wiki", e.g.:

```
GRANT ALL PRIVILEGES ON *.* TO 'ajaxify_wiki'@'localhost'  
IDENTIFIED BY 'ajaxify_wiki';
```
 - * If you'd prefer to use an existing account instead, set the details in Model/DAO.php for any wiki demo you wish to run.
- * Run run/wiki/Model/freshDB.sh. You only need to do this once, as the tables can be shared by all versions of the wiki. NOTE: Ensure your command-line path is set up so that "php" will resolve to php5.
- * Point your browser at the wiki, e.g. <http://localhost/run/wiki/>, and you should see three fresh messages.

TROUBLESHOOTING

=====

If you run into any problems:

- * Check the section below, as some demos require some extra setup.
- * Try the tutorial demo first. It's less error-prone because there's no library path involved.
- * Verify file permissions are correct. The easiest way is to run "chmod 755 document/root". In addition, ensure the records/ directory is writable.
- * For the wiki, manually check the database is ready and can be accessed with the settings in DAO.php.
- * Verify your PHP setup - create a file with "<? php_info ?>" and see if you can access it from the browser.
- * Ensure you're using PHP 5 (or later).
- * Check that you can run a test file, with both .php and .phtml suffixes.
- * Check that index.phtml is recognised as a valid directory index. (Check this by visiting path/index.phtml instead of just path/.)
- * Check that you set the library in .htaccess.
- * Check remoting is working as expected with a traffic sniffing tool. (See those listed in the Traffic Sniffing pattern).
- * Inspect log files. Your PHP log might be different to your Apache error log, depending on your setup.
- * If problems persist, mail michael@mahemoff.com.

DEMOS REQUIRING EXTRA SETUP

=====

These demos won't work unless you do some extra setup.

- * As mentioned earlier, all the wiki demos (/wiki/..) (but nothing else) require a MySQL database.
- * For all portal (/portal/..) and reader (/reader/..) demos, you need net access. For the portal synclinks demo (/run/portal/drilldown/synclinks/), you also need to run `prepareDrilldown.php` first.
- * For the shop demo, if you want to run the whole lazy registration process, you'll need some form
- * For the progress indicator in the Ajaxagram tutorial (/tutorial/ajaxagram/..), you either need to run it with net access, or change the image source in index.html to point to "/resources/progress.gif".

Patterns and Pattern Languages

ONE SIDE BENEFIT OF THIS PROJECT HAS BEEN TO SEE HOW “PATTERNS” ARE BEING ACCEPTED BY THE development community. Since the Ajax Patterns began to be published, the focus has always been on the “Ajax” and not the “Patterns.” For patterns, this is a good thing, and a welcome change from the late ‘90s, when you not only had to explain the concept but also justify its existence. “Ajax” itself is a pattern—one whose popularity is further evidence for the power of “just a name.” For all these reasons, this discussion of patterns and their relationship to Ajax Patterns is kept brief.

Patterns came about in the late 1970s, after Christopher Alexander and colleagues spent years studying the towns and buildings of many diverse cultures. The findings were summarized as 253 distinct patterns, derived from a global pattern describing the division of nations to town-planning patterns such as “Market of Many Shops,” all the way down to building patterns like “Soft Tile and Brick.” Each pattern is a “thing” you often see in good designs, combined with instructions on making the thing.

Patterns have mostly been confined to academia in their original field, but they have really taken off in software design. Their rise over the past decade has been a great boon to the industry, allowing us to learn about design from real examples and not just vague

comments about cohesion, coupling, and encapsulation. The pool of real examples to draw upon has itself risen exponentially, thanks to the Web and the open source movement.

The best-known patterns are software design patterns like Gamma et al.'s *Design Patterns* (1995). But that's not the only domain of software patterns, which have been applied to development processes, deployment practices, and—most important here—usability. Just as the original architecture patterns were fundamentally about improving the experience of inhabitants, usability patterns focus on the people who use—rather than develop—software systems. They have been especially popular for web design (e.g., *The Design of Sites* (Duyne et al., 2003), *A Pattern Language for Web Usability* (Graham, 2003), Martijn van Welie's "Web Design Patterns" at <http://www.welie.com/patterns/>). The Web is ideal for design patterns because examples are readily available for authors to mine and readers to try out.

The Ajax Patterns continue the tradition of web usability patterns, though they focus as much on technical design as on usability. As Alexander et al.'s work shows, it's perfectly possible (desirable, even) for a single language to range across several levels of abstraction. The first three sections of Alexander's work progress from low-level operations to high-level functionality and usability concepts. The fourth section is somewhat separate, as it addresses practices relevant to all those earlier sections, and in that sense is similar to some of the process-oriented pattern languages.

The question has arisen: are these patterns? According to most definitions, these are indeed; each Ajax Pattern shows how real projects have solved a recurring problem. In some cases, there's some speculation involved, but as long as any lack of evidence is declared and the idea is useful, I believe it's worth documenting in the pattern form. Another question here: is this a pattern language, or just a collection of patterns? I'm inclined to say the former, because the patterns are closely linked, filtered according to a consistent set of principles, and generative in that they build on each other. In any event, the most important question here is not about definitions, but about utility: how useful are the Ajax Patterns? I certainly hope the patterns are useful and practical, but I'll let you be the judge of that.

References

T

THESE ARE REFERENCES CITED THROUGHOUT THE TEXT.

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996). *Pattern-Oriented Software Architecture*. John Wiley & Sons. See <http://c2.com/cgi/wiki?PatternOrientedSoftwareArchitectureOne>.
- Cooper, A. (1999). *The Inmates are Running the Asylum*. Sams. See http://www.uidesign.net/1999/books/oct_books1.html.
- Duyne, D.K., Landay, J.A., Hong, J.I. (2002). *The Design of Sites*. Addison-Wesley Professional. See <http://www.designofsites.com/>.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns*. Addison-Wesley Professional. See <http://c2.com/cgi/wiki?DesignPatternsBook>.
- Graham, I. (2003). *A Pattern Language for Web Usability*. Pearson Education. See <http://www.wupatterns.com/>.
- Norman, D. (2003). *Emotional Design: Why We Love (Or Hate) Everyday Things*. Basic Books. See http://www.jnd.org/dn.mss/emotional_desig.html.

INDEX

Symbols

`$()` alias, 23
`$()` convenience function
 web remoting, 25
.NET Ajax frameworks, 586–590
<script> tag, 114

Numbers

204 response, 107
37Signals
 Backpack, 451
37signals, 447, 450, 455

A

A9, 3, 230
 Highlight, 470
 Maps, 393
accessibility, 44
ACE (see Ajax Client Engine)
ActiveMQ, 237
ActiveX
 Building ActiveX Controls for IE, 157
 Richer Plugin, 149
Adams, Cameron, 404
Adaptor pattern, 244
addEvent(), 136
AES, 496
AJASON, 597
Ajax, 559
 acronym, 6
 giving something back, 5
 portals, 6–9
 portlets, 7
 rise of, 5

trends, 17–20
 browser adoption, 18
 desktop, 18
 development support, 20
 Flash, 18
 hardware and bandwidth
 improvements, 20
 JavaScript, 20
 local storage, 19
 rich graphics, 19
 standards and fragmentation, 17
 two-way web, 19
 user acceptance, 18
 XForms, 19

Ajax applications
 characteristics of, 10–13
 continuous interaction, 11
 live, 11
 new styles of interactions, 13
 not just web sites, 11
 standards-based, 13
 supportive, 11
 visual effects, 11
 widgets, 13

Ajax Apps, 59–67
 Ajax Deluxe, 62
 Ajax Lite, 62
 bandwidth, 63
 browser compatibility, 62
 browser-processed power, 63
 browsers and, 60–61
 desktop apps, advantages over, 66
 Flash and, 64
 Java applets, advantages over, 65
 latency, 63

- Ajax Apps (*continued*)
 - making richer, 150
 - mobile apps, 67
 - networking capacity, 63
 - throughput requirements, 63
 - web apps, advantages over, 64
 - XMLHttpRequest, 62
- Ajax Blog, 183
- Ajax calendar, 62
- Ajax Client Engine (ACE), 211, 212, 213, 577
- Ajax Deluxe, 62
- Ajax frameworks and libraries, 567–600
 - .NET, 586–590
 - Ajax.NET, 586
 - AjaxAspects, 586
 - Anthem.NET, 586
 - Atlas, 587
 - Bitkraft, 587
 - ComfortASP.NET, 588
 - MagicAjax.NET, 588
 - MonoRail, 588
 - WebORB for .NET, 589
 - zumiPage, 589
- Backbase
 - JavaScript multipurpose, 568
- C++
 - Wt, 585
- ColdFusion
 - AjaxCFC, 585
 - JSMX, 585
- Java, 590–595
 - AJAX JSP Tag Library, 590
 - AjaxAnywhere, 590
 - AJAX-JSF, 591
 - Direct Web Remoting (DWR), 591
 - Echo 2, 591
 - Guisse, 592
 - ICEfaces, 592
 - JSON-RPC-Java, 592
 - JSP Controls Tag Library, 593
 - jWic, 593
 - Struts-Layout, 593
 - SWATO, 593
 - Tacos Tapestry Components, 594
 - ThinkCAP JX, 594
 - WebORB, 595
 - WidgetServer, 595
- JavaScript effects
 - FAT, 580
 - Moo.fx, 580
- JavaScript Flash, 581
 - AMFPHP, 581
 - Flash JavaScript Integration Kit, 581
 - Stream, 581
- JavaScript multipurpose, 568–577
 - AjaxFace, 568
 - Bindows, 569
 - Dojo, 569
 - EBA, 570
 - Engine for Web Applications, 570
 - Freja, 571
 - Interactive Website Framework, 576
 - Mochikit, 571
 - OpenRico, 572
 - Plex Toolkit, 572
 - Prototype, 573
 - qooxdoo, 573
 - Scriptaculous, 573
 - SmartClient, 574
 - ThyAPI, 575
 - TIBCO General Interface, 575
 - Zimbra AjaxTK, 576
- JavaScript remoting, 577–580
 - Ajax Client Engine, 577
 - AjaxCaller, 577
 - AjaxGear, 578
 - AJFORM, 578
 - HtmlHttpRequest, 578
 - JSMX, 578
 - libXmlRequest, 579
 - MAJAX, 579
 - RSLite, 579
 - Sack, 580
 - Subsys_JsHttpRequest, 580
 - XHConn, 580
- JavaScript specialized
 - Drag-Drop library, 582
 - GSV, 582
- JavaScript XML, 582
 - Google AJAXSLT, 582
 - Sarissa, 582
- Lisp, 595
 - CL-Ajax, 595
- multilanguage, 582–585
 - CPAINT, 583
 - JSON, 583
 - JSON-RPC, 583
 - JSRS, 583
 - Rialto, 584
 - SAJAX, 584
 - ZK—AJAX/XUL, 584
- Perl, 596
 - CGI::Ajax, 596
 - HTML::Prototype, 596
- PHP, 596–600
 - AJASON, 597
 - AjaxAC, 597
 - Cajax, 597

- HTSWaf, 598
- JSPSPAN, 598
- NAJAX, 599
- PEAR::HTML::Ajax, 598
- Pipeline, 598
- Symfony, 599
- XAJAX, 599
- XOAD, 599
- Python, 600
 - CrackAJAX, 600
 - Turbo Gears, 600
- Ruby On Rails, 600
- AJAX JSP Tag Library, 590
- Ajax Lite, 62
- Ajax Login Demo, 491–493
- Ajax Pages, 269
- Ajax Patterns Demo
 - code, 601–603
- Ajax Patterns Demos, 54–55
- Ajax Patterns overview, 47–48
- Ajax Patterns Reader, 552, 556, 559
 - Logging, 536
 - System Test, 563
- Ajax Shop Demo, 480–486
 - cart management, 482
 - mailing cart contents, 483
 - retrieval of categories and items, 482
 - tracking favorite categories, 483
 - verifying password and email, 484
- Ajax spell checker, 76
- Ajax Stub, 161, 182–186
 - Ajax Blog, 183
 - AjaxPatterns SAJAX Sum, 185
 - alternatives, 185
 - CL-AJAX, 185
 - code refactoring, 185
 - CPAINT, 184
 - decisions, 183
 - DWR, 183, 184
 - examples, 184
 - JSPspan, 183
 - JSON Message, 186, 207
 - related patterns, 186
 - RESTful Service, 184
 - RPC Service, 181, 186
 - SAJAX, 184
 - securing server-side functions, 183
 - SOAP, 186
 - solution, 182
 - XMLHttpRequest Call, 185
 - XML-RPC, 186
- Ajax Stub frameworks, 178
- Ajax technologies, 14–15
 - CGI, 14
 - CSS, 14
 - DOM, 15
 - event model, 15
 - form submission, 14
 - HTML/XHTML, 14
 - HTTP, 14
 - JavaScript, 14
 - server-side scripting, 14
 - Web Remoting, 15
 - XML, 15
- Ajax.NET, 586
- Ajax.Net, 204
- AjaxAC, 597
- Ajaxagram, 29–33
 - web service, 29
- ajaxagram.css, 35
- ajaxagram.js, 29, 31
- AjaxAnywhere, 590
- AjaxAspects, 586
- AjaxCaller, 577
- AjaxCaller library, 213
- AjaxCFC, 585
- AjaxFace, 568
- AjaxGear, 578
- AJAX-JSF, 591
- AjaxPattern On-Demand JavaScript Wiki, 129
- AjaxPatterns Assistive Search, 226
- AjaxPatterns Basic Wiki, 83, 322–324
- AjaxPatterns Basic Wiki Demo, 148
- AjaxPatterns Data Island Sum, 257–258
- AjaxPatterns Distributed Events Wiki
 - Demo, 237–242
- AjaxPatterns Form-Based Sum, 230–232
- AjaxPatterns HeartBeat Wiki, 512–514
 - list of online users, 514
 - periodic loop, 513
- AjaxPatterns Periodic Time, 219
- AjaxPatterns Portal, 314
- AjaxPatterns Predictive Fetch Sum, 301
- AjaxPatterns Restful Shop Demo, 173–175
- AjaxPatterns RPC Shop Demo, 180
- AjaxPatterns SAJAX Sum, 185
- AjaxPatterns Spotlight Time, 452–454
- AjaxPatterns Status Wiki, 429–431
- AjaxPatterns Streaming Wiki, 120
- AjaxPatterns Templating Drilldown
 - Demo, 270–273
- AjaxPatterns TestAjaxCaller, 90, 103–105
- AjaxPatterns Timeout Wiki, 505–508
 - monitoring mouse movements, 507
 - unconditional timeout, 505
 - warning message, 506
- AjaxPatterns Unique URL Sum, 524–529
 - Full IFrame Sum Demo, 527
 - IFrame source, 528
 - IFrame Sum Demo, 526
 - Polling URL Sum Demo, 525
 - synchronization logic, 528
 - Unique URL Sum Demo, 525

- AjaxPatterns Wiki, 413–415
 - Highlight, 470
- AjaxPatterns Wiki Links Page, 349
- AjaxPatterns XML Sum, 200
- AjaxPatterns XSLT Drilldown Demo, 263–264
- AJAX-S
 - Browser-Side XSLT, 262
- Ajax-S, 76
- AjaxTags, 280
- AJFORM, 578
- Alexander, Christopher, 605
- altKey, 140
- Am I Ajax Or Not, 40
- AMASS, 320
- Amazon, 365, 480
 - A9 search engine (see A9)
 - Diamond Search, 333, 339, 372
 - Mini Shop, 153, 155–156
 - public APIs, 245
 - Zuggest, 190, 340
 - Submission Throttling, 225
- AMFPHP, 581
- Amigo, Gustavo Ribeiro, 269
- anagram.php, 29
- anagram.phtml, 29, 30
- AnagramTest.php, 30
- animation loop, 460
- Anthem.NET, 586
- Anyterm, 10, 103
- appendChild(), 72
- architectures, archetypal, 61
- are, 156
- Arthur, Kevin, 122
- Ashley, Brent, 107
- Assistive Search Demo, 379–382
- Assistive Search demo, 374
- asXMLTag(), 349
- asynchronous calls, 94
- asynchrony, 46
- Atlas, 587
- Atom, 172
- atomic processing, 212

B

- Backbase, 279, 424, 530, 568
 - Browser-Side Templating, 270
 - Portal, 345, 392
 - One-Second Motion, 465
 - XMLHttpRequest Call, 103
- Backpack, 147
 - RESTful Service, 169
 - User Action, 141
- Baekdal, Thomas, 3, 413
- bandwidth, 20, 46, 63
- Basic AjaxPatterns Demos, 142

- Basic Sum Demo, 341, 524
- Basic Wiki Demo, 505
- Behavior Message, 125
 - On-Demand JavaScript, 131
- Berners-Lee, Tim, 530
- Betfair, 345, 385
 - Status Area, 429
- Bill Gates' Wealth Clock, 247
- Bindows, 569
- Bitkraft, 587
- Blogger, 172
- Blowfish, 496
- Blummy, 479
- Bone, Jeff, 177
- bookmarklets, 41, 157
 - manager, 479
- Boston.com, 334
- Brattli, Thomas, 131
- browser compatibility, 62
 - issues, 284
- Browser Controller, 561
- Browser Simulation, 561
- Browser Status Bar, 431
- Browser UIs, 110
- browser-processessing power, 63
- browsers
 - Ajax Apps and, 60–61
 - built-in browser cache, 296
 - feature support, 287
- browser-server dialogue, 209–251
 - Call Tracking (see Call Tracking)
 - Cross-Domain Proxy (see Cross-Domain Proxy)
 - Distributed Events (see Distributed Events)
 - Explicit Submission (see Explicit Submission)
 - Periodic Refresh (see Periodic Refresh)
 - Submission Throttling (see Submission Throttling)
- Browser-Side Cache, 39, 157, 289–297, 350, 355, 367, 382, 439
 - alternatives, 296
 - built-in browser cache, 296
 - cache size considerations, 292
 - Cached Sum Demo, 293–296
 - code example, 293–296
 - decisions, 292–293
 - dequeue(), 295
 - encache(), 295
 - enqueue(), 295
 - examples, 293
 - finite cache, 295
 - infinite cache, 294
 - keys and values, 292
 - libXmlRequest, 293
 - onSumResponesnot, 294

- Periodic Refresh, 220
- Predictive Fetch, 298, 302
- queryCache(), 296
- related patterns, 297
- repaintSum(), 294
- sendToTail(), 295
- server-side cache, 297
- solution, 290–292
- stale data, 292
- Submission Throttling, 297
- submitSum() queries, 296
- Virtual Workspace, 443
- XML Data Island, 256, 259
- XMLHttpRequest Call, 290, 293
- Browser-Side Templating, 39, 265–273, 282
 - Ajax Pages, 269
 - AjaxPatterns Templating Drilldown Demo, 270–273
 - alternatives, 273
 - Backbase, 270
 - Browser-Side XSLT, 265, 273
 - code in template, 268
 - code refactoring, 270–273
 - decisions, 268–269
 - examples, 269–270
 - improving template context, 272
 - JavaScript Templates (JST), 269
 - JSON Message, 273
 - obtaining the template, 268
 - preparing template context, 268
 - related patterns, 273
 - rendering from template, 271
 - solution, 266–268
 - XML Data Island, 259
 - XML Message, 198, 273
 - Browser-Side Test, 549, 554–557, 560, 561, 564
 - Browser-Side Test
 - Logging, 557
 - code example, 556
 - JsUnit, 556
 - related patterns, 557
 - Scriptaculous, 555
 - Service Test, 557
 - solution, 554
 - tool support, 555
 - xUnit, 555
 - Browser-Side XSLT, 39, 259–265
 - AjaxPatterns XSLT Drilldown Demo, 263–264
 - AJAX-S, 262
 - alternatives, 265
 - Browser-Side Templating, 265, 273
 - code refactoring, 263–264
 - decisions, 261
 - examples, 262
 - Google Maps, 262
 - Kupu, 262
 - obtaining stylesheet, 261
 - related patterns, 265
 - Sarissa, 263
 - solution, 260–261
 - XML Data Island, 256, 259, 265
 - XML Message, 265
 - built-in browser cache, 296
 - Business Card Creator, 413
 - button, 140
- C**
 - C++ Ajax frameworks
 - Wt, 585
 - Cached Sum Demo, 293–296
 - Caesar, 496
 - Cajax, 597
 - calendar, 82
 - call context, 212
 - Call Tracking, 38, 95, 209–214
 - Ajax Client Engine (ACE), 211, 212, 213
 - AjaxCaller library, 213
 - alternatives, 214
 - atomic processing, 212
 - call context, 212
 - code example, 213
 - examples, 212
 - Fire-and-Forget, 214
 - global XMLHttpRequest, 214
 - IFrame Call, 110
 - libXmlRequest, 213
 - limiting, 212
 - logging, 212
 - pooling, 211
 - Predictive Fetch, 214
 - sequencing, 212
 - solution, 210
 - timeouts, detecting, 212
 - callback, 136
 - Categorization of Test Doubles, 553
 - Category, 349
 - CGI, 14
 - CGI::Ajax, 596
 - chat, two-person, 40
 - chat.app, 451
 - Chen, Zhijie, 594
 - childNodes[], 72
 - Chipman, Steven, 544
 - CL-AJAX, 185, 595
 - clearTimeout, 146
 - click ‘n’ wait, 142
 - clientX, clientY, 140
 - Close Consultants, 493
 - code for Chapter 2 and Ajax Patterns Demo, 601–603

- ColdFusion Ajax frameworks
 - AjaxCFC, 585
 - JSMX, 585
 - Coloir, 451
 - Comfort.ASP, 280
 - ComfortASP.NET, 588
 - compatibility, 44, 46
 - consistency, 61
 - cookies
 - Lazy Registration, 477, 478
 - Costello, Roger, 177
 - Cote, Stephen W., 293, 571
 - Couvreux, Julien, 545
 - Cowin, Richard, 444
 - CPAINT, 184, 583
 - CPaint, 248
 - CrackAJAX, 600
 - Craigslist, 246
 - Creative Commons web site, 385
 - Creative Commons' issueLicense web service, 250
 - Crockford, Douglas, 126
 - Cross-Browser Component, 275, 283–288, 357
 - browser-feature support, 287
 - code example, 287
 - decisions, 285
 - examples, 287
 - HTMLHttpRequest library, 287
 - incompatibility issues, 284
 - more information, 288
 - On-Demand JavaScript, 288
 - related patterns, 288
 - Server-Side Code Generation, 288
 - solution, 284
 - transparency, 284
 - XMLHttpRequest, 284
 - Cross-Domain Portlet, 423
 - Cross-Domain Proxy, 39, 40, 209, 243–251, 349, 423
 - Adaptor pattern, 244
 - alternatives, 250
 - Bill Gates' Wealth Clock, 247
 - code example, 248–250
 - connecting to external server, 245
 - CPaint, 248
 - Craigslist, 246
 - decisions, 244
 - document.domain, 250
 - external content, 244
 - Facade, 244
 - Google Maps, 246, 251
 - Housing Maps, 246
 - images, 251
 - more information, 251
 - On-Demand JavaScript, 250
 - Performance Optimization Patterns, 251
 - Portlet, 427
 - related patterns, 251
 - solution, 244
 - Web Service, 91
 - WPLicense, 248–250
 - handling user answers, 250
 - retrieving license questions, 249
 - retrieving license types, 249
 - WPLicense Wordpress plugin, 246
 - Cross-Domain Scripting, 125
 - On-Demand JavaScript, 125
 - CSS, 14
 - ajaxagram.css, 35
 - CSS stylesheet call, 107
 - ctrlKey, 140
 - Curran, Peter, 493
 - customization, 45
- ## D
- Dam, James, 491, 493
 - Data Grid, 39, 375
 - Data Grid widget, 350–355
 - code example, 353
 - Delta Vacations, 353
 - examples, 352
 - NumSum, 352
 - Oddpost, 353
 - OpenRico, 352, 353
 - read-only or mutable, 352
 - related patterns, 354
 - solution, 351
 - Debugging, 533, 537–539
 - JavaScript HTML Debugger, 539
 - Microsoft Script Debugger, 538
 - solution, 537
 - tool support, 538
 - Venkman, 538
 - debugging
 - decision whether or not to debug, 538
 - Delicious, 365
 - On-Demand JavaScript, 127
 - public APIs, 245
 - Delicious JSON Feed, 203
 - Delta Vacations, 353, 372, 374
 - Demo RSS Reader, 103
 - design, 43–55
 - Ajax Patterns Demos, 54–55
 - Ajax Patterns overview, 47–48
 - desireable attributes of Ajax
 - applications, 43
 - patterns
 - anatomy, 48–54
 - software design principles, 45
 - usability, 44
 - desktop, 18

- desktop apps
 - advantage of Ajax Apps over, 66
- desktop client, 156, 325
- developer productivity, 43
- Development Environment (SPADE), 156
- Development patterns, ix, 40
- DHTML Lemmings, 396, 397–400
 - Fat Client, 322
 - One-Second Motion, 465
 - One-Second Mutation, 458
- diagnosis, 533–547
 - Debugging (see Debugging)
 - DOM Inspection (see DOM Inspection)
 - Logging (see Logging)
 - Traffic Sniffing (see Traffic Sniffing)
- Diamonds, Jason, 586
- Digg Spy, 76, 189, 190, 433, 434
 - One-Second Spotlight, 451
- Direct Login, 473, 487–493
 - Ajax Login Demo, 491–493
 - code example, 491–493
 - decisions, 490
 - hashing algorithm, 490
 - Host-Proof Hosting, 493, 498
 - Lazy Registration, 486, 493
 - NetVibes, 491
 - one-time seed, 490
 - Protopage, 491
 - related patterns, 493
 - seed, 492
 - solution, 488–490
 - Timeout, 493, 508
 - Treehouse Magazine, 491
- Direct Web Remoting (DWR), 591
- display demo, 23
- Display Morphing, 69–78
 - Ajax spell checker, 76
 - AjaxPatterns Countdown Demo, 77
 - Ajax-S, 76
 - classname or style, 74
 - code example, 77
 - decisions, 74–76
 - Digg Spy, 76
 - DOM, 70–74
 - appendChild(), 72
 - childNodes[], 72
 - getElementsByName(), 72
 - id, 72
 - insertBefore(), 72
 - nodeType, 72
 - parentNode, 71
 - removeChild(), 72
 - tagName, 72
 - examples, 76
 - more information, 78
 - One-Second Mutation, 456
 - Page Rearrangement, 78, 83
 - properties, 75
 - related patterns, 78
 - src, 76
 - style.backgroundColor, 75
 - style.backgroundImage, 75
 - style.borderColor, 75
 - style.borderWidth, 75
 - style.fontColor, 75
 - style.fontStyle, 75
 - style.fontWeight, 75
 - style.innerHTML, 76
 - style.outerHTML, 76
 - style.fontSize, 75
 - User Action, 143
- Distributed Events, 19, 209, 232–242, 342
 - ActiveMQ, 237
 - addListener(), 234
 - AjaxPatterns Distributed Events Wiki Demo, 237–242
 - Browser-to-Browser, 234
 - Browser-to-Server, 234
 - code refactoring, 237–242
 - decisions, 235–236
 - Dojo Events library, 237
 - event mechanism, 237–240
 - event notification, 235
 - examples, 237
 - history versus current state, 235
 - HTTP Streaming, 234, 242
 - LivePage, 237
 - MapBuilder, 237
 - Observer-style events, 235
 - onUpdate(), 234
 - Periodic Refresh, 234, 242
 - Portlet, 426
 - processing events synchronously or asynchronously, 236
 - related patterns, 242
 - removeListener(), 234
 - RESTful Service, 242
 - Server-to-Browser, 234
 - Server-to-Server, 234
 - solution, 233–235
 - watchlist, 240
 - XML Data Island, 242
- Dobrado, 424
- Document Object Model (DOM), 61, 70–74
 - appendChild(), 72
 - childNodes[], 72
 - getElementsByName(), 72
 - id, 72
 - insertBefore(), 72
 - nodeType, 72
 - parentNode, 71
 - removeChild(), 72
 - tagName, 72
- document.domain, 250
- document.getElementById(), 23

- Dojo, 569
 - Events library, 237
 - On-Demand JavaScript, 128
 - Dojo Rich Text Editor, 357
 - Dojo Toolkit, 563
 - Unique URLs, 523
 - DOM, 15
 - DOM Inspection, 71, 533, 540–544
 - alternatives, 544
 - DOM Inspector tool (Firefox), 541
 - IE Developer Toolbar, 542
 - IEDocMon, 544
 - Mouseover DOM Inspector, 544
 - solution, 540
 - tool support, 541–544
 - View Rendered Source, 542
 - Web Developer extension, 541
 - DOM Inspector tool (Firefox), 541
 - DOM population, 253–273
 - Browser-Side Templating (see Browser-Side Templating)
 - Browser-Side XSLT (see Browser-Side XSLT)
 - XML Data Island (see XML Data Island)
 - drag function, 335
 - Drag-and-Drop, 336, 382, 389–394
 - A9 Maps, 393
 - alternatives, 393
 - Backbase Portal, 392
 - code example, 393
 - decisions, 392
 - examples, 392
 - Fat Client, 326
 - libraries, 391
 - Magnetic Poetry, 392, 393
 - Popup, 406
 - Portlet, 394
 - related patterns, 393
 - Scriptaculous, 391
 - Slider, 394
 - solution, 390
 - Sprite, 393, 400
 - Virtual Workspace, 443
 - drag-and-drop cart, 40
 - Drag-Drop library, 582
 - Drilldown, 385, 388
 - Microlink, 421
 - Portlet, 426
 - Drilldown widget, 329, 343–350
 - alternatives, 349
 - Backbase portal, 345
 - Betfair, 345
 - code refactoring, 346–349
 - browser-side implementation, 347–349
 - further, 349
 - server-side implementation, 349
 - decisions, 345
 - dynamic content, 349
 - examples, 345
 - Live Search, 349
 - OpenRico framework, 346
 - related patterns, 350
 - solution, 343–345
 - Tree widget, 350
 - DWR, 183, 184
 - dynamic behavior, 133–148
- ## E
- eBusiness Applications (EBA) Ajax
 - Components, 570
 - Echo2, 278, 591
 - Echo2 Number Guess, 280–282
 - GamePane class, 281
 - Eclipse, 378
 - efficiency, 44
 - Embedded Browser Scripts, 561, 564
 - Endecott, Phil, 103
 - Engine for Web Applications, 570
 - EvDB public APIs, 245
 - eVelocity, 499
 - event handlers, 141
 - event handling, 135
 - event model, 15
 - Event Model (DHTML), 61
 - event object attributes, 139
 - event type, 140
 - Excite.com, 7
 - Explicit Submission, 209, 227–232, 384
 - A9, 230
 - AjaxPatterns Form-Based Sum, 230–232
 - alternatives, 232
 - code refactoring, 230–232
 - decisions, 228
 - examples, 229
 - The Fonz, 230
 - frequent submissions, 229
 - Lace Chat, 229
 - Live Form, 232
 - Progress Indicator, 232
 - related patterns, 232
 - requesting submission, 228
 - solution, 228
 - Submission Throttling, 232
 - extended technologies, 149–157
 - Building ActiveX Controls for IE, 157
 - Firefox Extension Tutorial, 157
 - Richer Plugin (see Richer Plugin)
 - Tutorial: Creating a Mozilla Extension, 157
 - Extension
 - Richer Plugin, 149

F

- Facade, 180, 244
- Fade Anything Technique (FAT), 452, 580
- fade effect, 36
- Fat Client, 39, 156, 289, 317–326, 340, 382
 - AjaxPatterns Basic Wiki, 322–324
 - alternatives, 324
 - browsers, 319
 - business and application logic, 320
 - code example, 322–324
 - decisions, 320
 - desktop client, 325
 - DHTML Lemmings, 322
 - Drag-and-Drop, 326
 - examples, 321
 - Gmail, 321
 - Guesstimate, 310
 - Host-Proof Hosting, 326
 - JavaScript, 319
 - JS/UNIX shell, 322
 - more information, 326
 - NumSum, 321
 - On-Demand JavaScript, 326
 - Periodic Refresh, 220, 325
 - persistence, 320
 - Predictive Fetch, 302
 - related patterns, 325
 - security, 319
 - solution, 317–320
 - Submission Throttling, 325
 - thin client, 324
 - Web Service, 320
- FKCEditor, 357, 359–360
- Fettig, Abe, 119
- Fiddler, 546
- Fielding, Roy, 176, 530
- filesystem navigation, 40
- findItems(\$name), 349
- Fire-and-Forget, 214
- Firefox
 - Amazon Mini Shop, 153
 - Google Suggests, 153
 - Google Toolbar, 153
- Firefox "Prefetch", 300
- Firefox Extension Tutorial, 157
- Firefox extensions
 - View Rendered Source, 542
 - Web Developer, 541
- flangelib.js library, 547
- Flash, 18, 61
 - Odeo and, 154
 - Richer Plugin, 149
- Flash app
 - Ajax Apps and, 64
- Flash JavaScript Integration Kit, 581
- Flickr, 3, 4, 9, 61, 180, 404, 410
 - public APIs, 245
- The Fonz, 230
- forces driving Ajax, 60
- form submission, 14
- Foundational Technology patterns, ix, 38
- fragmentation, 17
- frame call, 112
- Freja, 571
- Fuecks, Harry, 183
- Full IFrame Sum Demo, 527
- Full IFrame URL Sum Demo, 525
- functionality, 33–36
- Functionality and Usability patterns, ix, 39
- fvLogger, 536

G

- GamePane class, 281
- Garrett, Jesse-James, 5
- GET, 96
- getElementsByTagName(), 72
- GetIFrameHtml(), 359
- GhostTrain, 562
- Giant-Ass Image Viewer (GSV), 582
- Giant-Ass Image Viewer (GSV) Library, 441
- Gmail, 9
 - Fat Client, 321
 - Highlight, 470
 - Microlink, 419
 - storage space, 306
 - Submission Throttling, 225
- Goodman, Danny, 520
- Google AJAXSLT, 582
- Google Homepage, 424
- Google Maps, 3, 5, 9, 64, 334, 397, 529
 - Browser-Side XSLT, 262
 - Cross-Domain Proxy, 246
 - IFrame Call, 111
 - Predictive Fetch, 300
 - public APIs, 245
 - tiling, 400
 - User Action, 141
 - XML Message, 200
- Google Maps API
 - Cross-Domain Proxy, 251
- Google Reader
 - User Action, 141
- Google Search
 - public APIs, 245
- Google Suggest, 5, 146, 226, 362, 364, 368
 - Submission Throttling, 225
- Google Suggests for Firefox, 153
- Google Toolbar for Firefox, 153
- Grand, Mark, 515

- Greasemonkey
 - Richer Plugin, 149
- Greenspun, Philip, 247
- Griffes, Michael, 499
- Guesstimate, 289, 303–310, 339, 342, 367
 - code example, 308–309
 - consolidating with real data, 305
 - curCount, 308
 - decisions, 305
 - doCountdown(), 308
 - examples, 306
 - Fat Client, 310
 - Gmail
 - storage space, 306
 - historical data, 304
 - how often executed, 305
 - HTTP Streaming, 304
 - iTunes Counter, 306, 308–309
 - Lazy Registration, 486
 - Multi-Stage Download, 316
 - One-Second Motion, 468
 - Periodic Refresh, 220, 304, 310
 - Predictive Fetch, 302, 310
 - rate, 308
 - related patterns, 310
 - runCountdown(), 308, 309
 - server support, 306
 - setCounters(), 308
 - solution, 303–305
 - user knowledge, 305
 - Virtual Workspace, 443
- Guise, 592
- H**
- Hack
 - Richer Plugin, 149
- Hansson, David Heinemeier, 600
- hardware, 20
- Hartley, Jeremy, 530
- Heartbeat, 220, 473, 509–515
 - AjaxPatterns HeartBeat Wiki, 512–514
 - list of online users, 514
 - periodic loop, 513
 - code refactoring, 512–514
 - list of online users, 514
 - periodic loop, 513
 - decisions, 511
 - Last Seen or Last Request field, 511
 - maintaining user records, 511
 - more information, 515
 - period between Heartbeats, 512
 - Periodic Refresh, 515
 - related patterns, 515
 - solution, 510–511
 - stopping Heartbeats, 511
 - Submission Throttling, 515
 - Timeout, 508, 515
- Hello World!, 22
- Hertel, Matthias, 587
- Hieatt, Edward, 556
- Highlight, 352, 367, 446, 468–472
 - A9, 470
 - AjaxPatterns Wiki, 470
 - code example, 470
 - decisions, 469
 - examples, 470
 - Gmail, 470
 - Malleable Content, 415
 - One-Second Spotlight, 472
 - related patterns, 471
 - solution, 469
 - Status Area, 471
 - Teacher!, 470
 - Whitespace, 470
- Hill, Tony, 177, 427
- Hoffman, Thad, 256
- Holst, Laurens, 530
- Host-Proof Hosting, 473, 493–499
 - alternatives, 498
 - code example, 496
 - decisions, 496
 - Direct Login, 493, 498
 - encryption, 496
 - Fat Client, 326
 - more information, 498
 - pass-phrase request, 496
 - related patterns, 498
 - Richer Plugin, 498
 - solution, 494
 - Timeout, 498
- HousingMaps, 194
 - Cross-Domain Proxy, 246
- HoverSearch, 404
- href property, 24
- HTML, 14
- HTML Message, 161, 186–191, 387
 - alternatives, 191
 - Amazon Zuggest, 190
 - code example, 190
 - decisions, 188
 - Digg Spy, 189, 190
 - examples, 189–190
 - JSON Message, 191
 - Multi-Stage Download, 190
 - On-Demand JavaScript, 131, 191
 - Plain-Text Message, 191
 - Rapha, 189
 - related patterns, 191
 - Server-Side Code Generation, 283
 - solution, 187
 - TalkDigger, 190
 - when to use, 187
 - XML Message, 191
- HTML:Prototype, 596

- HTMLHttpRequest
 - IFrame Call, 112
- HtmlHttpRequest, 578
- HTMLHttpRequest library, 287
- HTSWaf, 598
- HTTP Meta Refresh, 148
- HTTP Streaming, 19, 85, 113–122
 - <script> tag, 114
 - AjaxPatterns Streaming Wiki, 120
 - alternatives, 121
 - closing connection, 118
 - code refactoring, 120
 - connection time, 118
 - decisions, 118
 - Distributed Events, 234, 242
 - flush(), 115
 - Guesstimate, 304
 - IFrame Call, 115
 - Jotspot Live, 119
 - Lightstreamer, 120
 - LivePage, 119
 - messages, 118
 - more information, 122
 - onreadystatechange, 121
 - page streaming, 114, 115, 118
 - Periodic Refresh, 118, 121, 220
 - pollLatestResponse(), 121
 - portable page updates, 117
 - Pushlets, 120
 - Realtime on Rails, 119
 - responseText property of
 - XMLHttpRequest, 117
 - service streaming, 116, 117, 118
 - solution, 113–118
 - Stream, 122
 - TCP connection, 122
 - Update Control, 435
 - Web Service, 120
 - XMLHttpRequest Call, 106, 115
- HTTPUnit, 563
- Hussenet, Claude, 146, 218, 424, 425

I

- ICEfaces, 592
- id (node), 72
- IE
 - Building ActiveX Controls for IE, 157
- IE Developer Toolbar, 542
- IEDocMon, 544
- IFrame, 359
- IFrame Call, 85, 108–112
 - Browser UIs, 110
 - Call Tracking, 110
 - examples, 111
 - frame call, 112
 - Google Maps, 111

- HTMLHttpRequest, 112
- HTTP Streaming, 115
- Kuro5hin, 111
- PXL8 Demo, 111
- Scoop, 111
- solution, 108–111
- XMLHttpRequest Call, 105, 110, 112

- IFrame source, 528
- IFrame Sum Demo, 526
- IFrame URL Sum Demo, 525
- IFrame-URL technique, 520
- images, 251
- index.html, 29, 31
- innerHTML, 24
- insertBefore(), 72
- instant messaging clients, 10
- IntelliJ, 378
- Interactive Website Framework, 576
- International Herald Tribune, 301
- Ippolito, Bob, 536
- Isaacs, Scott, 146
- iTunes Counter, 146, 306, 308–309

J

- Jakarta HTTPClient, 559
- Java, 149
 - Richer Plugin, 149
- Java Ajax frameworks, 590–595
 - AJAX JSP Tag Library, 590
 - AjaxAnywhere, 590
 - AJAX-JSF, 591
 - Direct Web Remoting (DWR), 591
 - Echo 2, 591
 - Guise, 592
 - ICEfaces, 592
 - JSON-RPC-Java, 592
 - JSP Controls Tag Library, 593
 - jWic, 593
 - Struts-Layout, 593
 - SWATO, 593
 - Tacos Tapestry Components, 594
 - ThinkCAP JX, 594
 - WebORB, 595
 - WidgetServer, 595
- Java applets
 - Ajax Apps advantages over, 65
- JavaScript, 11, 14, 20, 45, 61, 63, 157
 - Ajaxagram tutorial, 31
 - event model, 15
 - Fat Client, 319
 - handling events in, 26
 - registering events, 136
- JavaScript effects Ajax frameworks
 - FAT, 580
 - Moo.fx, 580

- JavaScript Flash frameworks, 581
 - AMFPHP, 581
 - Flash JavaScript Integration Kit, 581
 - Stream, 581
- JavaScript HTML Debugger, 539
- JavaScript JSON library, 202
- JavaScript multipurpose frameworks, 568–577
 - AjaxFace, 568
 - Bindows, 569
 - Dojo, 569
 - EBA, 570
 - Engine for Web Applications, 570
 - Freja, 571
 - Interactive Website Framework, 576
 - Mochikit, 571
 - OpenRico, 572
 - Plex Toolkit, 572
 - Prototype, 573
 - qooxdoo, 573
 - Scriptaculous, 573
 - SmartClient, 574
 - ThyAPI, 575
 - TIBCO General Interface, 575
 - Zimbra AjaxTK, 576
- Javascript Remote Scripting (JSRS), 583
- JavaScript remoting frameworks, 577–580
 - Ajax Client Engine, 577
 - AjaxCaller, 577
 - AjaxGear, 578
 - AJFORM, 578
 - HtmlHttpRequest, 578
 - JSMX, 578
 - libXmlRequest, 579
 - MAJAX, 579
 - RSLite, 579
 - Sack, 580
 - Subsys_JsHttpRequest, 580
 - XHConn, 580
- JavaScript specialized frameworks
 - Drag-Drop library, 582
 - GSV, 582
- JavaScript Templates (JST), 269
- JavaScript XML frameworks, 582
 - Google AJAXSLT, 582
 - Sarissa, 582
- Johnson, Corey, 536
- Jotspot, 357
- Jotspot Live, 119
- JPSpan, 598
- JPSpan, 183
- JS/UIX shell, 322
- JSCalc, 404, 405
- JSMX, 578, 585
- JSON, 126, 583
 - advantages over XML, 206
 - XML advantages over, 206
- JSON Message, 124, 161, 201–207, 352
 - Ajax Stub, 186, 207
 - Ajax.Net, 204
 - alternatives, 205–206
 - Browser-Side Templating, 273
 - code example, 204
 - Delicious JSON Feed, 203
 - examples, 203
 - HTML Message, 191
 - Kiko, 203, 204
 - On-Demand JavaScript, 207
 - related patterns, 207
 - Route Planning, 204
 - solution, 202–203
 - stringify(), 203
 - XML Message, 201, 205
- JSON Message (see JSON Message)
- JSONP, 124
- JSON-PHP library, 203
- JSON-RPC, 186, 583
- JSON-RPC-Java, 592
- JSP Controls Tag Library, 593
- JsUnit, 556
- JsUnit frameworks, 555
- Justus, Chris, 368
- jWic, 593

K

- Katrina-Boston overlay map, 334
- Kayak, 341, 364, 479
 - Multi-Stage Download, 313
 - Suggestion widget
 - code example, 365
- key modifiers, 140
- key pressing, 137
- keyboard focus, 138
- keyCode (IE), 140
- keywords, 49
- Kiko, 82
 - JSON Message, 203, 204
 - RPC Service, 179
- Kirk, Alexander, 90, 479
- Knabben, Frederico Caldeira, 357
- Koch, Peter-Paul, 107, 288
- Kruslicky, Christopher, 436
- Kupu, 262
- KuroShin, 111

L

- Lace Chat, 504
 - Explicit Submission, 229
 - Periodic Refresh, 217, 218, 504
 - Plain-Text Message, 193, 195
 - Status Area, 429, 504
 - XMLHttpRequest Call, 102
- Lacey, Steve, 479

- latency, 46, 63
- Lazy Loading, 125
 - On-Demand JavaScript, 131
- Lazy Registration, 368, 473–487
 - Ajax Shop Demo, 480–486
 - cart management, 482
 - mailing cart contents, 483
 - retrieval of categories and items, 482
 - tracking favorite categories, 483
 - verifying password and email, 484
 - Amazon.com, 480
 - Blummy, 479
 - code example, 480–486
 - cart management, 482
 - mailing cart contents, 483
 - retrieval of categories and items, 482
 - tracking favorite categories, 483
 - verifying password and email, 484
 - cookies, 478
 - decisions, 477
 - Direct Login, 486, 493
 - examples, 479–480
 - Guesstimate, 486
 - Kayak, 479
 - Live Form, 486
 - MemeFlow, 479
 - more information, 487
 - Palmsphere, 479
 - Portlet, 479
 - profile, 477, 478
 - related patterns, 486
 - solution, 475–477
 - technologies
 - cookie manipulation and session tracking, 477
 - database, 477
 - XMLHttpRequest Call, 477
 - Timeout, 486
- Lean IT Processes, 553
- lean manufacturing, 551
- Least Frequently Used (LFU), 292
- Least Recently Used (LRU), 292
- Leshem, Amir, 590
- Ley, Jim, 204
- libXmlRequest, 213, 293, 579
- licenseClass selector, 387
- Lightstreamer, 120
- limiting, 212
- Link, 349
- Lisp Ajax frameworks, 595
 - CL-Ajax, 595
- ListSomething.com, 372
- Live Command-Line widget, 330, 376–383
 - alternatives, 382
 - Assistive Search Demo, 379–382
 - benefits, 377
 - Browser-Side Cache, 382
 - code example, 379–382
 - decisions, 378
 - Drag-and-Drop, 382
 - examples, 379
 - Fat Client, 382
 - Live Search, 382
 - monitoring loop, 380
 - more information, 383
 - partial commands, 378
 - point-and-click, 382
 - Progress Indicator, 382
 - related patterns, 382
 - server provided information, 378
 - solution, 376–378
 - Status Area, 382
 - Submission Throttling, 382
 - Suggestion, 383
 - YubNub, 379
- Live Form
 - Explicit Submission, 232
 - Lazy Registration, 486
 - Malleable Content, 415
 - Microlink, 421
 - Portlet, 426
- Live Form widget, 330, 383–388
 - Betfair, 385
 - code example, 385–387
 - Drilldown, 388
 - examples, 384
 - Explicit Submission, 384
 - Live Search, 388
 - Microlink, 388
 - One-Second Spotlight, 388
 - Progress Indicator, 388
 - related patterns, 388
 - solution, 384
 - Suggestion, 388
 - typical modifications, 384
 - WPLicense, 384, 385–387
- Live Search, 34, 37, 340, 349, 364, 368, 382, 388
- Live Search widget, 330, 368–375
 - Amazon Diamond Search, 372
 - Assistive Search demo, 374
 - code examples, 374
 - Data Grid, 375
 - decisions, 371
 - Delta Vacations, 372, 374
 - examples, 372
 - ListSomething.com, 372
 - Multi-Stage Download, 371
 - One-Second Spotlight, 375
 - Periodic Refresh, 372
 - Predictive Fetch, 375
 - Progress Indicator, 374
 - related patterns, 374

- Live Search widget (*continued*)
 - results, 371
 - Skype, 373
 - solution, 369
 - Submission Throttling, 369, 370, 375
 - Suggestion widget, 375
 - too many results, 372
 - Unique URLs, 375
 - Virtual Workspace, 372, 375
- LivePage, 119
 - Distributed Events, 237
- log4j, 534, 536
- Logging, 533–536, 557
 - Ajax Patterns Reader, 536
 - decisions, 535
 - fvLogger, 536
 - log4j, 534
 - log4js, 536
 - Lumberjack, 536
 - Mochikit, 536
 - solution, 534
 - tool support, 536
 - Traffic Sniffing, 547
- logging, 212
- Lumberjack, 536

M

- Madden, Jennifer, 542
- MagicAjax.NET, 588
- Magnetic Poetry, 194, 392, 393
 - Periodic Refresh, 217
- Magurski, Michael, 441
- MAJAX, 579
- Malleable Content, 389, 407–415
 - AjaxPatterns Wiki, 413–415
 - alternatives, 415
 - Business Card Creator, 413
 - code example, 413–415
 - compound edit, 415
 - decisions, 408–410
 - ending editing, 410
 - examples, 410–413
 - Flickr, 410
 - Highlight, 415
 - labels, 409
 - Live Form, 415
 - locating and editing content, 408
 - Microlink, 415, 421
 - Monket Calendar, 411
 - multiple open blocks, 410
 - One-Second Mutation, 415, 458, 460
 - One-Second Spotlight, 415
 - related patterns, 415
 - Rich Text Editor, 415
 - size, 410
 - solution, 408
 - TiddlyWiki, 411

- map.search.ch, 300, 440
- MapBuilder, 127
 - Distributed Events, 237
- Mashup
 - Richer Plugin, 149
- Masinter, Larry, 530
- Meebo, 10
- MemeFlow, 479
- Microlink, 350, 384, 388, 389, 416–421
 - benefits, 417
 - code example, 419
 - context, 417
 - decisions, 417
 - Drilldown, 421
 - examples, 418
 - Gmail, 419
 - Live Form, 421
 - Malleable Content, 415, 421
 - One-Second Mutation, 421
 - One-Second Spotlight, 421
 - other content, 417
 - Popup, 406, 421
 - related patterns, 421
 - Rpad, 419
 - solution, 416
 - Tabtastic, 418
 - TiddlyWiki, 418, 419
 - visual presentation, 418
- Microsoft Script Debugger, 538
- Millar, Paul, 155
- Miller, David, 536
- MiniXML, 248
- Mint, 103
- mobile apps
 - Ajax Apps, 67
- Mochikit, 536, 571
- mocks, 551
- monitoring loop, 380
- Monket Calendar, 411
- MonoRail, 588
- Moo.fx, 580
- mouse buttons, 140
- mouse movement, 139
- mouse position, 140
- Mouseover DOM Inspector, 544
- MoveableType comment hack, 385
- MoveBy effect, 466
- moveSlider function, 335
- Mozilla
 - Tutorial: Creating a Mozilla Extension, 157
- Mozilla demo, 256
- MSN Earth, 529
- MS-Windows
 - One-Second Motion, 465

- multilanguage Ajax frameworks, 582–585
 - CPAINT, 583
 - JSON, 583
 - JSON-RPC, 583
 - JSRS, 583
 - Rialto, 584
 - SAJAX, 584
 - ZK, 584
 - ZK—AJAX/XUL, 584
- multiple tiers, 46
- Multi-Stage Download, 289, 310–316, 340, 346, 371
 - AjaxPatterns Portal, 314
 - alternatives, 316
 - calls, 313
 - code example, 314
 - decisions, 312
 - dividing page into blocks, 312
 - downloading content, 313
 - examples, 313
 - Guessimate, 316
 - HTML Message, 190
 - Kayak, 313
 - NetVibes, 313
 - On-Demand JavaScript, 316
 - page structure, 312
 - Portlet, 316
 - Progress Indicator, 316
 - related patterns, 316
 - solution, 311
 - Virtual Workspace, 443

N

- NAJAX, 599
- navigator.appVersion, 286
- navigator.UserAgent, 286
- .NET Ajax frameworks
 - Ajax.NET, 586
 - AjaxAspects, 586
 - Anthem.NET, 586
 - Atlas, 587
 - Bitkraft, 587
 - ComfortASP.NET, 588
 - MagicAjax.NET, 588
 - MonoRail, 588
 - WebORB for .NET, 589
 - zumiPage, 589
- Netflix, 404
- Netflix Top 100, 198
- NetVibes, 8, 89, 491
 - Multi-Stage Download, 313
- networking capacity, 63
- Neuberg, Brad, 320, 520, 530
- Newman, Richard, 185
- Nielsen, Jakob, 338
- nodeType, 72
- nonstandard technologies, 61

- NumSum, 321, 352
- nUnit frameworks, 555
- Nyman, Robert, 76, 262

O

- Occasional Refresh, 529
- Oddpost, 353
- Odeo, 154, 451
- OECD privacy guidelines, 487
- office applications, 9–10
- onblur, onfocus, 138
- onchange, 139
- On-Demand JavaScript, 38, 39, 40, 86, 122–131
 - AjaxPattern On-Demand JavaScript Wiki, 129
 - Behavior Message, 125, 131
 - code example, 129
 - Cross-Browser Component, 288
 - Cross-Domain Proxy, 250
 - Cross-Domain Scripting, 125
 - decisions, 126
 - Delicious, 127
 - Dojo, 128
 - examples, 127
 - Fat Client, 326
 - HTML Message, 131, 191
 - JSON Message, 207
 - Lazy Loading, 125, 131
 - MapBuilder, 127
 - more information, 131
 - Multi-Stage Download, 316
 - Predictive Fetch, 131
 - related patterns, 131
 - Script Tag Creation, 123, 125, 126, 129
 - code, 130
 - technique, 130
 - Service Eval, 124, 126
 - solution, 123–126
 - upload.js, 129
 - uploadMessage function, 129
 - Web Service, 124
 - XMLHttpRequest Call, 106, 124
- One-Second Motion, 415, 445, 462–468
 - Aesthetic effect, 464
 - alternatives, 467
 - Attention-grabbing, 464
 - Backbase Portal, 465
 - code example, 465–467
 - decisions, 463–465
 - destination behavior, 465
 - DHTML Lemmings, 465
 - examples, 465
 - Gesture, 464
 - Guessimate, 468
 - Malleable Content, 415
 - motion trigger, 463

- One-Second Motion (*continued*)
 - MoveBy effect, 466
 - MS-Windows, 465
 - One-Second Mutation, 461, 467
 - One-Second Spotlight, 454, 467
 - Page Rearrangement, 463
 - related patterns, 467
 - render(), 466
 - Scheduling, 463
 - Scriptaculous, 465, 465–467
 - solution, 463
 - Sprite, 467
 - TiddlyWiki, 465
 - One-Second Mutation, 445, 455–462
 - alternatives, 461
 - animation loop, 460
 - Blow Up, 457
 - code example, 458–461
 - Cover Up, 457
 - CSS properties
 - backgroundImage, 456
 - height, 456
 - overflow, 456
 - position, 456
 - width, 456
 - decisions, 457
 - DHTML Lemmings, 458
 - Disappear, 457
 - Display Morphing, 456
 - examples, 458
 - Form, 456
 - Grow, 457
 - Malleable Content, 415, 458, 460
 - Metamorphise, 457
 - Microlink, 421
 - One-Second Motion, 461, 467
 - One-Second Spotlight, 454, 461
 - related patterns, 461
 - Reveal, 457
 - Scheduling, 456
 - Scriptaculous, 458
 - Shrink, 457
 - Slide In, 457
 - Slide Out, 456
 - solution, 456–457
 - Sprite, 461
 - startAnimating(), 460
 - TiddlyWiki, 458, 458–461
 - One-Second Spotlight, 35, 342, 375, 388, 445–455
 - 37Signals Backpack, 451
 - AjaxPatterns Spotlight Time, 452–454
 - alternatives, 454
 - chat.app, 451
 - code refactoring, 452–454
 - Coloir, 451
 - colors, 447, 450
 - decisions, 449
 - Digg Spy, 451
 - examples, 451–452
 - Fade Anything Technique (FAT)
 - Library, 452
 - Fade Away, 448
 - Fade In, 448
 - Fade Out, 448
 - Highlight, 472
 - length of spotlight, 450
 - Malleable Content, 415
 - Materialize, 448
 - Microlink, 421
 - Odeo, 451
 - One-Second Motion, 454, 467
 - One-Second Mutation, 454, 461
 - opacity, 447
 - Periodic Refresh, 454
 - related patterns, 454
 - Scriptaculous, 452, 454
 - Shifting Flash, 448
 - solution, 446–449
 - spotlight triggers, 449
 - Switch, 448
 - Timeout, 454
 - Total Flash, 448
 - transitioning to background, 449
 - visibility, 447
 - onkeypress, onkeydown, onkeyup, 137
 - onload, onunload, 135
 - onmouseover, onmouseout, 139
 - onmouseup, 138
 - onselect, 139
 - opacity, 284
 - OpenRico, 346, 572
 - Data Grid example, 352, 353
 - library, 5
 - Search, 441–442
 - Demo, 440
 - Yahoo! Search Demo, 372
 - operating system timeouts, 505
 - Orchard, Dunstan, 441
- P**
- page architecture patterns, 389–444
 - Drag-and-Drop (see Drag-and-Drop)
 - Malleable Content (see Malleable Content)
 - Microlink (see Microlink)
 - Popup (see Popup)
 - Portlet (see Portlet)
 - Sprite (see Sprite)
 - Status Area (see Status Area)
 - Update Control (see Update Control)
 - Virtual Workspace (see Virtual Workspace)

- Page Rearrangement, 69, 78–83, 384, 385
 - AjaxPatterns Basic Wiki, 83
 - code example, 83
 - decisions, 81
 - Display Morphing, 78, 83
 - garbage collection, 81
 - Kiko, 82
 - One-Second Motion, 463
 - positioning style, 81
 - related patterns, 83
 - solution, 79
 - Super Maryo World, 82
 - TadaList, 81
 - User Action, 143
- page streaming, 114, 115
- PairStairs, 523
- Palmsphere, 479
- Pandora, 504
- parentNode, 71
- Parker, Andrew, 307
- Pascarello, Eric, 504, 535
- PasteText command, 360
- pattern languages, 605
- pattern-led tutorial, 21–41
- patterns, 605
 - Ajax App (see Ajax Apps)
 - Ajax Stub (see Ajax Stub)
 - anatomy, 48–54
 - best-known, 606
 - bookmarklets, 157
 - Browser-Side Cache (see Browser-Side Cache)
 - Browser-Side Templating (see Browser-Side Templating)
 - Browser-Side Test (see Browser-Side Test)
 - Browser-Side XSLT (see Browser-Side XSLT)
 - Call Tracking (see Call Tracking)
 - Cross-Browser Component (see Cross-Browser Component)
 - Cross-Domain Proxy (see Cross-Domain Proxy)
 - Data Grid (see Data Grid)
 - Debugging (see Debugging)
 - Development (see Development patterns)
 - Direct Login (see Direct Login)
 - Display Morphing (see Display Morphing)
 - Distributed Events (see Distributed Events)
 - DOM Inspection (see DOM Inspection)
 - Drag-And-Drop (see Drag-And-Drop)
 - Drilldown (see Drilldown)
 - Explicit Submission (see Explicit Submission)
 - Fat Client (see Fat Client)
 - Foundational Technology (see Foundational Technology patterns)
 - Functionality and Usability (see Functionality and Usability patterns)
 - Guesstimate (see Guesstimate)
 - Heartbeat (see Heartbeat)
 - Highlight (see Highlight)
 - Host-Proof Hosting (see Host-Proof Hosting)
 - HTML Message (see HTML Message)
 - HTTP Streaming (see HTTP Streaming)
 - IFrame Call (see IFrame Call)
 - JSON Message (see JSON Message)
 - Lazy Registration (see Lazy Registration)
 - Live Command (see Live Command-Line widget)
 - Live Form (see Live Form)
 - Live Search (see Live Search)
 - Logging (see Logging)
 - Malleable Content (see Malleable Content)
 - Microlink (see Microlink)
 - Multi-Stage Download (see Multi-Stage Download)
 - On-Demand JavaScript (see On-Demand JavaScript)
 - One-Second Motion (see One-Second Motion)
 - One-Second Mutation (see One-Second Mutation)
 - One-Second Spotlight (see One-Second Spotlight)
 - overview, 47–48
 - Page Rearrangement (see Page Rearrangement)
 - Periodic Refresh (see Periodic Refresh)
 - Plain-Text Message (see Plain-Text Message)
 - Popup (see Popup)
 - Portlet (see Portlet)
 - Predictive Fetch (see Predictive Fetch)
 - Programming (see Programming patterns)
 - Progress Indicator (see Progress Indicator)
 - RESTful Service (see RESTful Service)
 - Rich Text Editor (see Rich Text Editor)
 - Richer Plugin (see Richer Plugin)
 - RPC Service (see RPC Service)
 - Scheduling (see Scheduling)
 - Server-Side Code Generation (see Server-Side Code Generation)
 - Service Test (see Service Test)
 - Simulation Service (see Simulation Service)
 - Single Page Application, 156
 - Slider (see Slider)
 - Sprite (see Sprite)
 - Status Area (see Status Area)

- patterns (*continued*)
 - Submission Throttling (see Submission Throttling)
 - Suggestion (see Suggestion)
 - System Test (see System Test)
 - Timeout (see Timeout)
 - Traffic Sniffing (see Traffic Sniffing)
 - Unique URLs (see Unique URLs)
 - Update Control (see Update Control)
 - User Action (see User Action)
 - Virtual Workspace (see Virtual Workspace)
 - Web Service (see Web Service)
 - XML Data Island (see XML Data Island)
 - XML Message (see XML Message)
 - XMLHttpRequest Call (see XMLHttpRequest Call)
- PEAR::HTML::Ajax, 598
- Pederick, Chris, 541
- PerfectXML Demo, 256
- performance optimization, 289–326
 - Browser-Side Cache (see Browser-Side Cache)
 - Fat Client (Fat Client)
 - Guesstimate (performance optimization)
 - Multi-Stage Download (see Multi-Stage Download)
 - Predictive Fetch (see Predictive Fetch)
- Performance Optimization Patterns, 251
- Periodic Refresh, 19, 63, 209, 215–221, 372, 505
 - AjaxPatterns Periodic Time, 219
 - alternatives, 220
 - Browser-Side Cache, 220
 - Claude Hussenet's portal, 218
 - code examples, 218
 - code refactoring, 219
 - decisions, 216
 - Distributed Events, 234, 242
 - examples, 217
 - Fat Client, 220, 325
 - Guesstimate, 220, 304, 310
 - Heartbeat, 220, 515
 - HTTP Streaming, 118, 121, 220
 - Lace Chat, 217, 218, 504
 - Magnetic Poetry, 217
 - One-Second Spotlight, 454
 - Portlet, 426
 - refresh period, 216
 - Scheduling, 215
 - solution, 215
 - Status Area, 431
 - Submission Throttling, 220, 227
 - Timeout, 501, 508
 - Update Control, 435
- Perkin, Philip, 257
- Perl Ajax frameworks, 596
 - CGI::Ajax, 596
 - HTML::Prototype, 596
- Personalization vs. Customization, 487
- pessimistic locking, 502
- PHP Ajax frameworks, 596–600
 - AJASON, 597
 - AjaxAC, 597
 - Cajax, 597
 - HTSWaf, 598
 - JSPAN, 598
 - NAJAX, 599
 - PEAR::HTML::Ajax, 598
 - Pipeline, 598
 - Symfony, 599
 - XAJAX, 599
 - XOAD, 599
- PHP HTTPClient, 559
- PHPUnit2, 30
- Pick'em Game, 341
- Pipeline, 598
- plain-text formats, 192
- Plain-Text Message, 161, 191–195
 - alternatives, 195
 - code example, 195
 - examples, 193
 - Housing Maps, 194
 - HTML Message, 191
 - Lace Chat, 193, 195
 - Magnetic Poetry, 194
 - RPC Service, 181
 - solution, 192
 - XML Message, 192, 195, 201
 - XMLHttpRequest Call, 192
- Plex Toolkit, 572
- Plugin
 - Richer Plugin, 149
- plugins
 - advantages of writing your own, 153
 - third-party, 152
 - when not installed, 153
- PNG File Format, 400
- podcast manager, 154
- point-and-click, 382
- Polling URL Sum Demo, 524, 525
- pooling, 211
- Popup, 39, 139, 337, 342, 389, 401–407
 - alternatives, 406
 - appearance, 403
 - code example, 405
 - decisions, 403
 - Drag-and-Drop, 406
 - examples, 404
 - Flickr, 404

- HoverSearch, 404
- JSCalc, 404, 405
- Microlink, 406, 421
- Netflix, 404
- opacity, 402
- opening and closing, 403
- Portlet, 406
- related patterns, 406
- Slider, 406
- solution, 401
- Sprite, 400, 406
- Status Area, 431
- Submission Throttling, 227
- Timeout, 508
- visibility, 402
- zIndex, 402
- portability, 46
- Portal Demo, 346
- portals, 6–9
- Portlet, 39, 340, 345, 350, 389, 421–427
 - alternatives, 426
 - Backbase, 424
 - Claude Hussenet’s portal, 424, 425
 - code example, 425
 - Cross-Domain Portlet, 423
 - Cross-Domain Proxy, 427
 - decisions, 424
 - Distributed Events, 426
 - Dobrado, 424
 - Drag-and-Drop, 394
 - Drilldown, 426
 - Google Homepage, 424
 - Lazy Registration, 479
 - Live Form, 426
 - moving around, 424
 - Multi-Stage Download, 316
 - Periodic Refresh, 426
 - Popup, 406
 - related patterns, 426
 - solution, 422–423
 - tabbed browsing, 426
- portletlets, 7, 40
- POST, 96
- Predictive Fetch, 39, 289, 297–303, 352, 355, 367, 375
 - AjaxPatterns Predictive Fetch Sum, 301
 - alternatives, 302
 - Browser-Side Cache, 298, 302
 - Call Tracking, 214
 - code refactoring, 301
 - decisions, 299
 - examples, 300
 - Fat Client, 302
 - Firefox “Prefetch”, 300
 - Google Maps, 300
 - Guesstimate, 302, 310
 - International Herald Tribune, 301
 - map.search.ch, 300
 - On-Demand JavaScript, 131
 - prefetching, 299
 - related patterns, 302
 - server priming, 302
 - solution, 298
 - user actions, 299
 - Virtual Workspace, 298, 443
 - when used, 298
 - XMLHttpRequest Call, 298
- pre-fetching, 300
 - problems with, 298
- Preston, Donovan, 119, 122
- privacy, 44
- Programming patterns, ix, 38
- Progress Indicator, 34, 35, 352, 355, 360, 374, 382, 385, 388
 - Explicit Submission, 232
 - Multi-Stage Download, 316
 - Submission Throttling, 227
 - Timeout, 508
- Progress Indicator widget, 329, 336–342
 - additional information, 342
 - Amazon Diamond Search, 339
 - Amazon Zuggest, 340
 - code refactoring, 341
 - for short delays, 337
 - Kayak, 341
 - Pick’em Game, 341
 - Protopage, 340
 - providing feedback, 338
 - related patterns, 342
 - solution, 337
 - TalkDigger, 340
 - type to use, 338
- ProgressiveEnhancement
 - Richer Plugin, 149
- Proietti, Volerio, 580
- Protopage, 8, 340, 491
 - XML Message, 199
- Prototype, 573
- Prototype framework, 226
- Pryce, Nat, 523
- public APIs, 245
- Pushlets, 120
- PUT versus POST discussion, 177
- PXL8 Demo, 111
- Python Ajax frameworks, 600
 - CrackAJAX, 600
 - Turbo Gears, 600

Q

- qooxdoo, 573
- Quek, 397

R

- radiobuttons, 336, 370
- Rapha, 189
- RC4, 496
- Really Simple History, 523
- Realtime on Rails, 119
- references, 607
- reliability, 44
- Remix
 - Richer Plugin, 149
- removeChild(), 72
- removeEvent(), 136
- render(), 466
- repetition, 146
- ReplaceTextarea, 359
- REST, 176
 - Freja, 571
- REST (Representational State Transfer), 163–177
- REST vs. RPC, 177
- REST Wiki, 176
- RESTful Service, 161–177, 559
 - Ajax Stub, 184
 - AjaxPatterns Restful Shop Demo, 173–175
 - alternatives, 171
 - arbitrary interactions, 171
 - Atom, 172
 - Backpack, 169
 - Blogger, 172
 - cart contents
 - changing, 174
 - reading, 174
 - categories list, 173
 - code example, 173–175
 - data formats, 171
 - Distributed Events, 242
 - documenting services, 170
 - examples, 172
 - GET for queries, 168
 - HTTP methods, 167
 - hyperlinks, 170
 - idempotent, 170
 - individual category, 173
 - more information, 176
 - portability, 171
 - PUT and DELETE, 175
 - related patterns, 176
 - resources and operations, 165
 - RPC Service, 176, 179, 181
 - solution, 163–172
 - stateless services, 168
 - Timeout, 500
 - Unique URLs, 176
 - URLs, 167
 - URLs or URIs, 166
 - weighing, 171
 - XML Data Island, 176
 - XML Message, 176
- retrieveQuestions function, 387
- Rialto, 584
- Rich Text Editor, 359
 - Malleable Content, 415
- Rich Text Editor widget, 329, 355–361
 - Dojo Rich Text Editor, 357
 - examples, 357–359
 - FCCKEditor, 357, 359–360
 - Jotspot, 357
 - related patterns, 360
 - Rich Text Editor, 359
 - solution, 356
 - typical features, 356
 - Wikipedia, 358
 - Writely, 358
- richer form, 143
- Richer Plugin, 61, 122, 149, 150–152, 154, 155, 157
 - Host-Proof Hosting, 498
 - XMLHttpRequest Call, 106
- Romney, Christian, 355
- Rosenberg, Doron, 288
- Roth, Kevin, 359
- Route Planning, 204
- Rpad, 419
- RPC Service, 161, 177–181
 - Ajax Stub, 181, 186
 - AjaxPatterns RPC Shop Demo, 180
 - alternatives, 181
 - cart contents
 - changing, 181
 - reading, 181
 - code example, 180
 - examples, 179
 - Facade, 180
 - Flickr, 180
 - individual category, 181
 - Kiko, 179
 - Plain-Text Message, 181
 - reading categories list, 181
 - related patterns, 181
 - RESTful Service, 176, 179, 181
 - solution, 178
 - XML Message, 181
- RPC, concrete forms of, 178
- RSA, 496
- RSLite, 579
- RSLite library, 107
- Ruby Net::HTTP, 559
- Ruby On Rails, 278, 600

runSearch(), 366
Russell, Alex, 78, 122, 499
Ruston, Jeremy, 155, 458, 462

S

Sack, 580
Sack library, 387
SAJAX, 184, 584
sandbox DOM node, 23
Sarissa, 263, 582
Scarfe, Tim, 78
Scheduling, 37, 133, 135, 143–148
 AjaxPatterns Basic Wiki Demo, 148
 alternatives, 148
 Backpack, 147
 canceling the timer, 144
 Claude Hussenet's Portal, 146
 clearTimeout, 146
 code example, 148
 examples, 146
 Google Suggest, 146
 HTTP Meta Refresh, 148
 iTunes Counter, 146
 One-Second Motion, 463
 One-Second Mutation, 456
 Periodic Refresh, 215
 repetition, 146
 setInterval, 146
 solution, 144–146
Scheffler, Martin, 119
Schwartz, Michael, 586
Schwartz, Richard, 214, 496, 499
Scoop, 111
Scott, Bill, 444
Script Tag Creation, 123, 125, 126
 code, 130
 technique, 130
Scriptaculous, 391, 555, 573
 One-Second Motion, 465, 465–467
 One-Second Mutation, 458
 One-Second Spotlight, 452, 454
search portlet, 40
seed, 490, 492
selection input, 367
selectors, 336, 370
Selenium, 563
sequencing, 212
Serpent, 496
server calls, 15–17
server priming, 302
server push, 19
server-side cache, 297
Server-Side Code Generation, 275–283
 AjaxTags, 280
 alternatives, 282
 Backbase, 279
 benefits, 277
 code example, 280–282
 GamePane class, 281
 Comfort.ASP, 280
 Cross-Browser Component, 288
 Echo2, 278
 Echo2 Number Guess, 280–282
 GamePane class, 281
 examples, 278–280
 HTML Message, 283
 related patterns, 283
 Ruby On Rails, 278
 solution, 277–278
 usability, 277
server-side scripting, 14
Service Eval, 124, 126
service streaming, 116, 117
Service Test, 40, 553, 557, 558–560
 Ajax Patterns Reader, 559
 Browser-Side Test, 560
 code example, 559
 Jakarta HTTPClient, 559
 PHP HTTPClient, 559
 RESTful Service, 559
 Ruby Net::HTTP, 559
 Simulation Service, 560
 solution, 558
 tool support, 559
 Web Service, 91
Session Warning Demo, 504
setInterval, 146
Shanahan, Francis, 190, 340
shiftKey, 140
Shoemaker, Craig, 5
shortcut keys, 138
Simian Design, 385
Simulation Service, 549–553, 560, 564
 Ajax Patterns Reader, 552
 Categorization of Test Doubles, 553
 code example, 552
 decisions, 552
 Lean IT Processes, 553
 more information, 553
 related patterns, 553
 Service Test, 553
 solution, 550
 Web Service, 91
Singh, Darshan, 256
Single Page Application pattern, 156
Skype, 373
Slide, 434
Slider, 370
 Drag-and-Drop, 394
 Popup, 406
 Virtual Workspace, 443

- Slider widget, 329, 330–336
 - alternatives, 336
 - Amazon Diamond Search, 333
 - drag function, 335
 - examples, 333
 - Google Maps, 334
 - handle, 335
 - Katrina-Boston overlay map, 334
 - moveSlider function, 335
 - orientation, 332
 - related patterns, 336
 - scale, 332
 - solution, 331
 - typical control mechanisms, 331
 - WebFX Slider Demo, 335
 - Yahoo! Mindset, 333
 - code example, 335–336
- smartBox, 366
- SmartClient, 574
- SOAP, 178, 186
- software design principles, 45
- spell checker, 76
- Sprite, 389, 394–400
 - alternatives, 400
 - animation, 396
 - code example, 397–400
 - decisions, 396
 - DHTML Lemmings, 396, 397–400
 - Drag-and-Drop, 393, 400
 - examples, 396
 - file formats, 396
 - Google Maps, 397
 - more information, 400
 - One-Second Motion, 467
 - One-Second Mutation, 461
 - Popup, 400, 406
 - Quek, 397
 - related patterns, 400
 - solution, 395
 - tiling, 400
- srcElement (IE), 139
- standard technologies, 61
- standards, 17
- startAnimating(), 460
- Status Area, 139, 342, 361, 382, 389, 427–431
 - AjaxPatterns Status Wiki, 429–431
 - alternatives, 431
 - Betfair, 429
 - Browser Status Bar, 431
 - code refactoring, 429–431
 - decisions, 428
 - examples, 429
 - Highlight, 471
 - Lace Chat, 429, 504
 - Periodic Refresh, 431
 - Popup, 431
 - related patterns, 431
 - size and overflow, 428
 - solution, 428
 - structuring content, 429
 - Timeout, 508
- Stenhouse, Mike, 517, 530
- Stimmerman, Brett, 102, 193, 429, 504
- Strategy pattern, 466
- Stream, 122, 581
- streaming, 19
- Struts-Layout, 593
- stubs, 551
- Submission Throttling, 37, 39, 209, 221–227, 367, 369, 370, 375, 382
 - AjaxPatterns Assistive Search, 226
 - Amazon Zuggest, 225
 - Browser-Side Cache, 297
 - buffers per browser application, 224
 - code example, 226
 - decisions, 223–225
 - Explicit Submission, 232
 - Fat Client, 325
 - Gmail, 225
 - Google Suggest, 225, 226
 - Heartbeat, 515
 - incoming commands to server, 223
 - Periodic Refresh, 220, 227
 - Popup, 227
 - Progress Indicator, 227
 - Prototype framework, 226
 - requestValidCategoriesLoop, 226
 - solution, 222
 - throttle period, 225
 - triggering buffer uploads, 223
- submitWord(), 32, 35
- Subsys_JsHttpRequest, 580
- Suggestion, 39, 383, 388
- Suggestion widget, 330, 361–368
 - alternatives, 367
 - Amazon, 365
 - auxiliary information, 364
 - Browser-Side Cache, 367
 - code example, 365
 - decisions, 363–364
 - Delicious, 365
 - examples, 364
 - Google Suggest, 362, 364
 - Guesstimate, 367
 - Highlight, 367
 - Kayak, 364
 - Lazy Registration, 368
 - Live Search, 368
 - Live Search widget, 375
 - mechanics of, 362
 - more information, 368
 - number of suggestions, 364
 - order of suggestions, 364
 - Predictive Fetch, 367

- related patterns, 367
- rules of thumb, 363
- selection input, 367
- solution, 362
- Submission Throttling, 367
- Super Maryo World, 82
- SVG, 19
- SWATO (Shift Web Application TO...), 593
- Symfony, 599
- System Test, 40, 549, 560–564
 - Ajax Patterns Reader, 563
 - Browser Controller, 561
 - Browser Simulation, 561
 - Browser-Side Test, 561, 564
 - code example, 563
 - Dojo Toolkit, 563
 - Embedded Browser Scripts, 561, 564
 - GhostTrain, 562
 - HttpUnit, 563
 - more information, 564
 - related patterns, 564
 - Selenium, 563
 - Simulation Service, 564
 - solution, 561
 - tool support, 562
 - Watir, 562

T

- tabbed browsing, 426
- Tabor, Tim, 544
- Tabtastic, 418
- Tacos Tapestry Components, 594
- TadaList, 81
- tagName, 72
- tags, 49
- TalkDigger, 340
 - HTML Message, 190
 - Multi-Stage Download
 - TalkDigger, 313
- target (Firefox), 139
- TCP connection, 122
- TEA (Tiny Encryption Algorithm), 496
- Teacher!, 470
- Technorati, 89
 - public APIs, 245
- TechRepublic demo, 257
- terminal emulators, 10
- Test-Driven Design (TDD), 551
- testing, 549–564
 - A Summary of Functional Testing Frameworks, 564
 - Browser-Side Test (see Browser-Side Test)
 - Service Test (see Service Test)
 - Simulation Service (see Simulation Service)

- thin client, 324
- ThinkCAP JX, 594
- third-party plugins, 152
- throughput requirements, 63
- ThyAPI, 575
- TIBCO General Interface, 575
- TiddlyWiki, 155, 411, 418, 419
 - One-Second Motion, 465
 - One-Second Mutation, 458, 458–461
- tiling, 400
- Time Periodic Refresh demo, 452
- Time Spotlight Demo, 452
- Timeout, 473, 499–508
 - applications of server-side session tracking, 502
 - code refactoring, 505–508
 - monitoring mouse movements, 507
 - unconditional timeout, 505
 - warning message, 506
 - decisions, 503
 - Direct Login, 493, 508
 - examples, 504
 - Heartbeat, 508, 515
 - historical records, 502
 - Host-Proof Hosting, 498
 - inactive users, 501
 - invalidating server-side session, 502
 - Lace Chat, 504
 - Lazy Registration, 486
 - length of timeout, 503
 - multi-user awareness, 502
 - One-Second Spotlight, 454
 - operating system timeouts, 505
 - Pandora, 504
 - Periodic Refresh, 501, 508
 - pessimistic locking, 502
 - Popup, 508
 - Progress Indicator, 508
 - related patterns, 508
 - RESTful Service, 500
 - Session Warning Demo, 504
 - solution, 500–503
 - Status Area, 508
 - sudden timeouts, 501
 - user-interface, 503
 - work scheduling, 503
- timeouts, detecting, 212
- Tognazzini, Bruce, 338
- tracking support, 339
- Traffic Sniffing, 533, 544–547
 - code example, 547
 - Fiddler, 546
 - flangelib.js library, 547
 - Logging, 547
 - related patterns, 547
 - solution, 545

- Traffic Sniffing (*continued*)
 - tool support, 545
 - XMLHttpRequest Tracing script, 547
- Tranquilli, Michele, 111
- transparency, 284
- Tree widget, 350
- Treehouse Magazine, 491
- Try Ruby, 10
- Turbo Gears, 600
- Turnbull, Angus, 112, 287
- Tutorial, 131
- tutorial, 21–41
 - Ajaxagram, 29–33
 - ajaxagram.js, 29, 31
 - anagram.php, 29
 - anagram.phtml, 29
 - anagrams.phtml, 30
 - application logic, 31
 - CSS, 35
 - index.html, 29, 31
 - presentation, 31
 - submitWord(), 32, 35
 - updateAnagramsDisplay(), 33
 - web service, 29
 - ajaxCaller, 37
 - ajaxifying web app, 27–40
 - refactoring exercises, 38–40
 - convenience function, 23
 - Development patterns, 40
 - display morphing, 22
 - dynamically creating link, 23
 - enhancing functionality and usability, 33–36
 - foundational technology patterns, 38
 - Functionality and Usability patterns, 39
 - href property, 24
 - innerHTML, 24
 - Live Search, 34, 37
 - One-Second Spotlight, 35
 - preparing demos, 22
 - Programming patterns, 38
 - Progress Indicator, 34
 - refactoring for performance, 37
 - refactoring to external library, 37
 - sandbox DOM node, 23
 - Submission Throttling, 37
 - suggested projects, 40–41
 - Am I Ajax Or Not, 40
 - bookmarklet, 41
 - drag-and-drop cart, 40
 - filesystem navigation, 40
 - image slideshow, 40
 - search portlet, 40
 - two-person chat, 40
 - Whack-A-Mole game, 40

- web remoting, 24–26
 - \$() convenience function, 25
 - handling events in JavaScript, 26
 - XMLHttpRequest usage, 25
- tutorial.js, 24
- Twofish, 496
- type, 140

U

- Unique URL Sum Demo, 524, 525
- Unique URLs, 39, 375, 474, 515–530
 - 'Here' link, 529
 - AjaxPatterns Unique URL Sum, 524–529
 - IFrame source, 528
 - IFrame Sum Demo, 526, 527
 - Polling URL Sum Demo, 525
 - synchronization logic, 528
 - Unique URL Sum Demo, 525
 - alternatives, 529
 - appearance, 523
 - code refactoring, 524–529
 - Full IFrame Sum Demo, 527
 - IFrame source, 528
 - IFrame Sum Demo, 526
 - Polling URL Sum Demo, 525
 - synchronization logic, 528
 - Unique URL Sum Demo, 525
 - decisions, 521
 - Dojo Toolkit, 523
 - examples, 523
 - fragment identifiers, 517
 - IE history, 519
 - IFrame-URL technique, 520
 - more information, 530
 - Occasional Refresh, 529
 - PairStairs, 523
 - polling, 518
 - interval, 522
 - Really Simple History, 523
 - RESTful Service, 176
 - search engine indexing support, 521
 - solution, 516–521
 - state changes, 522
 - Virtual Workspace, 443
- Update Control, 389, 431–436
 - code example, 434
 - Digg Spy, 433, 434
 - examples, 433
 - HTTP Streaming, 435
 - Periodic Refresh, 435
 - related patterns, 435
 - Slide, 434
 - solution, 432
 - WBIR, 434

- updateAnagramsDisplay(), 33
- upload.js, 129
- URLs or URIs, 166
- usability, 33–36, 43, 44
 - patterns, 39
- User Action, 133–143
 - alternatives, 142
 - altKey, 140
 - Backpack, 141
 - Basic AjaxPatterns Demos, 142
 - button, 140
 - clientX, clientY, 140
 - code example, 142
 - ctrlKey, 140
 - decisions, 137–141
 - Display Morphing, 143
 - event handlers, 141
 - event handling, 135
 - event object attributes, 139
 - event type, 140
 - examples, 141
 - Google Maps, 141
 - Google Reader, 141
 - key modifiers, 140
 - key pressing, 137
 - keyboard focus, 138
 - keyCode (IE), 140
 - mouse buttons, 140
 - mouse movement, 139
 - mouse position, 140
 - onblur, onfocus, 138
 - onchange, 139
 - onkeypress, onkeydown, onkeyup, 137
 - onmouseover, onmouseout, 139
 - onmouseup, 138
 - onselect, 139
 - Page Rearrangement, 143
 - related patterns, 143
 - shiftKey, 140
 - shortcut keys, 138
 - solution, 134–137
 - srcElement (IE), 139
 - target (Firefox), 139
 - type, 140
 - which (Firefox), 140
 - XMLHttpRequest Call, 143

V

- Van Everdigen, Michiel, 496
- Vandenberg, Adam, 547
- Vector Markup Language (VML), 19
- Venkman, 538
- View Rendered Source, 542
- Virtual Fish-Eye Lens, 443
- Virtual Magnifying Glass, 443

- Virtual Workspace, 39, 354, 360, 372, 375, 390, 436–444
 - alternatives, 443
 - Browser-Side Cache, 439, 443
 - code refactoring, 441–442
 - decisions, 439
 - Drag-and-Drop, 443
 - Dunstan Orchard's blog, 441
 - examples, 440
 - Giant-Ass Image Viewer (GSV)
 - Library, 441
 - Guesstimate, 443
 - initial view, 439
 - map.search.ch, 440
 - more information, 444
 - Multi-Stage Download, 443
 - navigable workspaces, 437
 - OpenRico Search, 441–442
 - Demo, 440
 - panning, 439
 - Predictive Fetch, 298, 443
 - related patterns, 443
 - repoulating regions, 439
 - Slider, 443
 - solution, 437–438
 - Unique URLs, 443
 - view changes, 439
 - Virtual Fish-Eye Lens, 443
 - Virtual Magnifying Glass, 443
- visual effects, 445–472
 - Highlight (see Highlight)
 - One-Second Motion (see One-Second Motion)
 - One-Second Mutation (see One-Second Mutation)
 - One-Second Spotlight (see One-Second Spotlight)

W

- Watchlist Wiki Demo, 240
- Watir, 562
- WBIR, 434
- wealth clock, 247
- web apps, 12
 - Ajax advantages over, 64
 - ajaxifying, 27–40
- Web Developer extension, 541
- Web Remoting, 15, 24–26, 61, 85–131
 - \$() convenience function, 25
 - handling events in JavaScript, 26
 - HTTP Streaming (see HTTP Streaming)
 - IFrame Call (see IFrame Call)
 - On-Demand JavaScript
 - Web Service (see Web Service)
 - XMLHttpRequest Call (see XMLHttpRequest Call)
 - XMLHttpRequest usage, 25

- Web Service, 85–91
 - AjaxPatterns TestAjaxCaller, 90
 - code example, 90
 - Cross-Domain Proxy, 91
 - decisions, 88
 - examples, 89
 - Fat Client, 320
 - HTTP Streaming, 120
 - Netvibes, 89
 - On-Demand JavaScript, 124
 - related patterns, 91
 - Service Test, 91
 - Simulation Service, 91
 - solution, 86–88
 - Technorati, 89
 - third-party usage, preventing, 88
 - Wish-O-Matic, 90
 - XMLHttpRequest Call, 98
- web services, 161–207
 - Ajax Stub (see Ajax Stub)
 - HTML Message (see HTML Message)
 - RESTful Service (see RESTful Service)
 - RPC Service (see RPC Service)
 - XML Message (see XML Message)
- WebFX Slider Demo, 335
- WebORB, 595
- WebORB for .NET, 589
- Were, Chris, 487
- Whack-A-Mole game, 40
- which (Firefox), 140
- Whitespace
 - Highlight, 470
- widgets, 13, 329–388
- WidgetServer, 595
- Wikipedia, 358
- Willison, Simon, 559
- Wish-O-Matic, 90
- WPLicense, 248–250, 384, 385–387
 - handling user answers, 250
 - retrieving license questions, 249
 - retrieving license types, 249
 - Wordpress plugin, 246
- Writely, 358
- Wt, 585
- X**
 - XAJAX, 599
 - XForms, 19
 - XHConn, 580
 - XHTML, 14
 - XML, 15
 - advantages over JSON, 206
 - JSON advantages over, 206
 - XML Data Island, 253–259
 - AjaxPatterns Data Island Sum, 257–258
 - alternatives, 259
 - Browser-Side Cache, 256, 259
 - Browser-Side Templating, 259
 - Browser-Side XSLT, 256, 259, 265
 - code refactoring, 257–258
 - Distributed Events, 242
 - examples, 256
 - including XML document on initial page load, 256
 - Mozilla demo, 256
 - PerfectXML Demo, 256
 - RESTful Service, 176
 - retaining XML for later use, 256
 - solution, 254–256
 - TechRepublic demo, 257
 - XML Message, 256
 - XML Message, 161, 195–201, 352
 - AJAX and, 197
 - AjaxPatterns XML Sum, 200
 - alternatives, 201
 - Browser-Side Templating, 198, 273
 - Browser-Side XSLT, 265
 - code refactoring, 200
 - decisions, 198
 - DTD or Schema, 198
 - examples, 198
 - Google Maps, 200
 - HTML Message, 191
 - JSON Message, 201, 205
 - Netflix Top 100, 198
 - parsing XML, 197
 - Plain-Text Message, 192, 195, 201
 - Protopage, 199
 - responseXML, 197
 - RESTful Service, 176
 - RPC Service, 181
 - server generating XML, 198
 - solution, 196–198
 - transforming XML, 198
 - XML Data Island, 256
 - XSLT, 198
 - XMLHttpRequest, 62, 124
 - browser compatibility issues, 284
 - Direct Login, 489
 - global, 214
 - responseText component, 339
 - responseText property, 117
 - Tracing script, 547
 - tracking, 210
 - Unique URLs, 516
 - usage, 25
 - XMLHttpRequest Call, 85, 86, 91–107, 337
 - 204 response, 107
 - abort(), 100
 - Ajax Stub, 185
 - AjaxPatterns TestAjaxCaller, 103–105
 - alternatives, 105–107
 - Anyterm, 103
 - API summary, 99

- asynchronous calls, 94
- Backbase, 103
- Browser-Side Cache, 290, 293
- caching, 101
- callback, 135
- code example, 103–105
- constraints on external domain, 97
- content, 101
- creating objects, 93
- CSS stylesheet call, 107
- Data Grid widget, 354
- decisions, 101–102
- Demo RSS Reader, 103
- Drilldown widget, 345
- error detection, 95
- error handling, 102
- examples, 102–103
- GET and POST, 96
- getAllResponseHeaders(), 100
- getResponseHeader(), 100
- HTTP Streaming, 106, 115
- IFrame Call, 105, 110, 112
- images, 251
- import XML documents, 107
- Lace Chat, 102
- Lazy Registration, 477, 478
- Live Form widget, 384
- Live Search widget, 370
- methods, 100
- Mint, 103
- On-Demand JavaScript, 106, 124
- onreadystatechange, 100
- open(), 100
- page refreshes, 105
- Plain-Text Message, 192
- Predictive Fetch, 298
- Progress Indicator widget, 338
- properties, 100
- readyState, 100
- responseText, 100

- responseXML, 100
- Richer Plugin, 106
- RSLite library, 107
- send(), 100
- setRequestHeader(), 100
- Slider widget, 333
- solution, 92–100
- status, 100
- statusText, 100
- Timeout, 501
- Traffic Sniffing, 545
- User Action, 143
- Web Service, 98
- XML responses, 98
- XMLHttpRequest Calls, 157
- XMLHttpRequest Tutorial, 204
- XML-RPC, 178, 186
- XOAD, 599
- XSLT, 198
 - quick overview, 260
- xUnit, 555

Y

- Yahoo Maps public APIs, 245
- Yahoo! Mindset, 333
- Yahoo! Mindset Slider code example, 335–336
- YAML, 206
- Yellow Fade Technique, 35, 447, 455
- YubNub, 379

Z

- Zijde, Tino, 400
- Zijdel, Tino, 396
- Zimbra AjaxTK, 576
- ZK, 584
- ZK—AJAX/XUL web framework, 584
- Zorn, Walter, 582
- zumiPage, 589

ABOUT THE AUTHOR

Michael Mahemoff has been working with software for 22 years, including 10 years of commercial experience. At the University of Melbourne, he earned degrees in software engineering and psychology, and earned a Ph.D with a thesis on “Design Reuse in Software Engineering and Human-Computer Interaction.” He lives in London and consults on software development issues in banking and healthcare. Michael’s blog and online projects are linked from his homepage at <http://mahemoff.com>.

COLOPHON

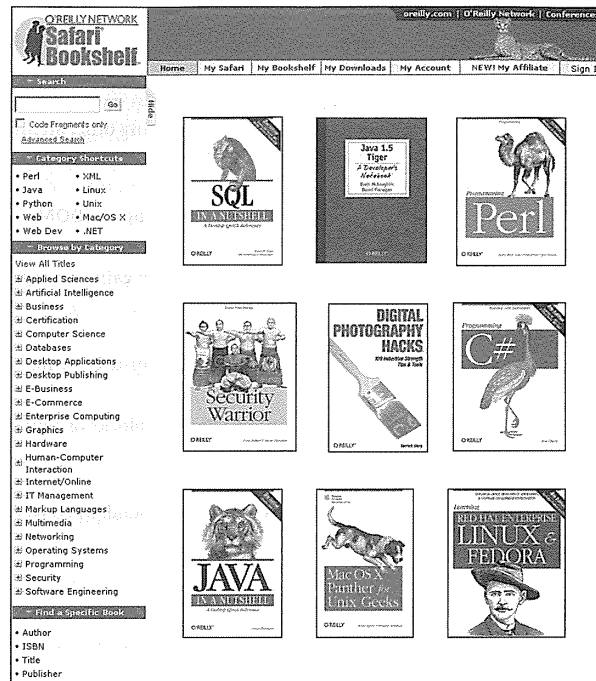
The cover image is from *Cassell's Natural History*. The cover fonts are Akzidenz Grotesk and Orator. The text font is Adobe's Meridien; the heading font is ITC Bailey.

Better than e-books

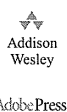
Buy *Ajax Design Patterns* and access the digital edition FREE on Safari for 45 days.

Go to www.oreilly.com/go/safarienabled and type in coupon code 5FJG-AVMK-KJ6L-EQLR-CY8M

- Search
thousands of
top tech books
- Download
whole chapters
- Cut and Paste
code examples
- Find
answers fast



Search Safari! The premier electronic reference library for programmers and IT professionals.



Ajax Design Patterns

Multi-Stage Download Page 310

Break content download into multiple stages, so that faster and more important content will arrive first.

On-Demand JavaScript Page 122

Download and run JavaScript snippets.

One-Second Motion Page 462

Incrementally move an element from one location to another, or temporarily move it, to communicate an event has occurred.

One-Second Mutation Page 455

When a page element undergoes a value change or some other significant event, dynamically mutate its shape for a second or so.

One-Second Spotlight Page 446

When a display element undergoes a value change—or any other significant change—dynamically increase its brightness for a second or so.

Page Rearrangement Page 78

Add, remove, move, and overlay elements by manipulating the DOM.

Periodic Refresh Page 215

The browser periodically issues an *XMLHttpRequest Call* to gain new information; e.g., one call every five seconds.

Plain-Text Message Page 191

Pass simple messages between server and browser in plain-text format.

Popup Page 401

Support quick tasks and lookups with transient Popups, blocks of content that appear “in front of” the standard content.

Portlet Page 421

Introduce Portlets—isolated blocks of content with independent conversational state.

Predictive Fetch Page 297

Pre-fetch content in anticipation of likely user actions.

Progress Indicator Page 336

Indicate the progress of server calls.

RESTful Service Page 162

Expose web services according to RESTful principles.

RPC Service Page 177

Expose web services as Remote Procedural Calls (RPCs).

Rich Text Editor Page 355

Incorporate a Rich Text Editor widget with easy formatting and WYSIWYG display.

Richer Plugin Page 149

Make your application “more Ajax than Ajax.”

Scheduling Page 143

Use JavaScript timers to schedule actions.

Server-Side Code Generation Page 275

Automatically generate HTML and JavaScript from server-side code.

Ajax Design Patterns

Service Test Page 558

Build up automated tests of web services, using HTTP clients to interact with the server as the browser normally would.

Simulation Service Page 550

Develop the browser application against “fake” web services that simulate the actual services used in production.

Slider Page 330

Provide a Slider to let the user choose a value within a range.

Sprite Page 394

Augment the display with Sprites: small, flexible, icon-like blocks of content.

Status Area Page 427

Include a read-only Status Area to report on current and past activity.

Submission Throttling Page 221

Instead of submitting upon each JavaScript event, retain data in a browser-based buffer and automatically upload it at fixed intervals.

Suggestion Page 361

Suggest words or phrases that are likely to complete what the user’s typing.

System Test Page 560

Build automated tests to simulate user behavior and verify the results.

Timeout Page 499

Have the browser Timeout the user after a period of inactivity, and optionally inform the server.

Traffic Sniffing Page 544

Diagnose problems by sniffing HTTP traffic.

Unique URLs Page 515

Provide Unique URLs for significant application states.

Update Control Page 431

Let the user control the rate of content updates.

User Action Page 133

Handle most User Actions within JavaScript, using event handlers.

Virtual Workspace Page 436

Provide a browser-side view into a server-side workspace, allowing users to navigate the entire workspace as if it were held locally.

Web Service Page 86

Expose server-side functionality as fine-grained, easy-to-use Web Services.

XML Data Island Page 254

Retain XML responses as XML Data Islands—nodes within the HTML DOM.

XML Message Page 195

Pass messages between server and browser in Extensible Markup Language (XML) format.

XMLHttpRequest Call Page 91

Use XMLHttpRequest objects for browser-server communication.

Ajax, or Asynchronous JavaScript and XML, is one of the hottest programming techniques among web developers. With its rich combination of technologies, Ajax provides a strong foundation for creating interactive web applications with XML or JSON-based web services by using JavaScript in the browser to process the web server response.

Ajax Design Patterns shows you best practices that can dramatically improve your web development projects. It investigates how others have successfully dealt with conflicting design principles in the past, and then relays that information directly to you.



The patterns outlined in the book fall into four categories:

Foundational technology: Examines the raw technologies required for Ajax development

Programming: Exposes techniques that developers have discovered to ensure their Ajax applications are maintainable

Functionality and usability: Describes the types of user interfaces you'll come across in Ajax applications, as well as the new types of functionality that Ajax makes possible

Development: Explains the processes used to monitor, debug, and test Ajax applications

Ajax Design Patterns will also get you up-to-speed with core Ajax technologies, such as XMLHttpRequest, the DOM, and JSON. Technical discussions are followed by examples so you can see for yourself just what is—and isn't—possible with Ajax. This handy reference will help you to produce high-quality Ajax architectures, streamline web application performance, and improve the user experience.

Michael Mahemoff holds a Ph.D in Computer Science and Software Engineering from the University of Melbourne, where his thesis was "Design Reuse in Software Engineering and Human-Computer Interaction." He lives in London and consults on software development issues in banking, healthcare, and logistics.

O'REILLY®

www.oreilly.com

US \$44.99

CAN \$58.99

ISBN-10: 0-596-10180-5

ISBN-13: 978-0-596-10180-0



Safari
BOOKS ONLINE
ENABLED

Includes
FREE 45-Day
Online Edition