1

Hello Web 2.0 World

When you visit a new country, a good way of getting started is to begin with a tour that gives you a first idea of what the country looks like and the key sites that you'll want to visit in more detail. This chapter is the tour that will give you the first idea of what a Web 2.0 application looks like from the inside and help you to get the big picture.

The Web 2.0 world is wide and rich, and the typical "Hello World" application wouldn't be enough to give you a good overview of a Web 2.0 application. BuzzWatch, the sample Web 2.0 application that you'll visit in this chapter, is thus more than the typical "Hello World" programming example. This chapter introduces most of the techniques that you will learn throughout the book, and you might find it difficult to grasp all the details the first time you read it. You can see it as the picture that is on the box of a jigsaw puzzle and use it as a guide to position the different pieces that you'll find in each chapter of the book. You can glance through it rapidly at first without installing the application, and revisit each point after you've seen the details in the corresponding chapter.

Introducing BuzzWatch

The application that you'll explore in this chapter is a program that aggregates information from multiple sources to give a different perspective. This kind of application is called a *mashup* and you'll see how to create mashups in more details in Chapter 15. This program is for executives who want to see, side by side, financial information about a company together with the vision from the web community on this same company. The goal is to compare the images from a company in the financial community (illustrated by information available on Yahoo! Finance) and in the web community (illustrated by del.icio.us). There are a number of sites that let you build pages with information pulled from different sources, but this one is really simple: you just need to enter a stock symbol and a del.icio.us tag and you're done. To share it, you'll also be able to save these pages together with a title and a description so that other people can use them. This shareable aspect is what makes BuzzWatch a read/write application that fully deserves to be called a Web 2.0 application.

What does the name BuzzWatch mean? BuzzWatch is about watching companies, thus the Watch *in its name. With its Yahoo! financial news and del.icio.us panels, the application is good at watching how the buzz emitted by the companies is perceived, and that's the reason for the* Buzz *in its name. If you need another reason for* Buzz*, note that like most Web 2.0 applications BuzzWatch is fully buzzword compliant!*

These concepts of aggregating multiple sources and sharing with others are the foundations of the Web 2.0 social layer described in the Introduction. To make the application conform to the technical layer of Web 2.0, the application uses a number of typical Web 2.0 techniques. The information is presented in panels that you can close and drag along the page to change its presentation. The title and description uses edit-in-place techniques that hide the ugly HTML form inputs when you don't use them, and the information will be periodically refreshed using Ajax to avoid reloading the page and to support having different refresh frequencies for each data source.

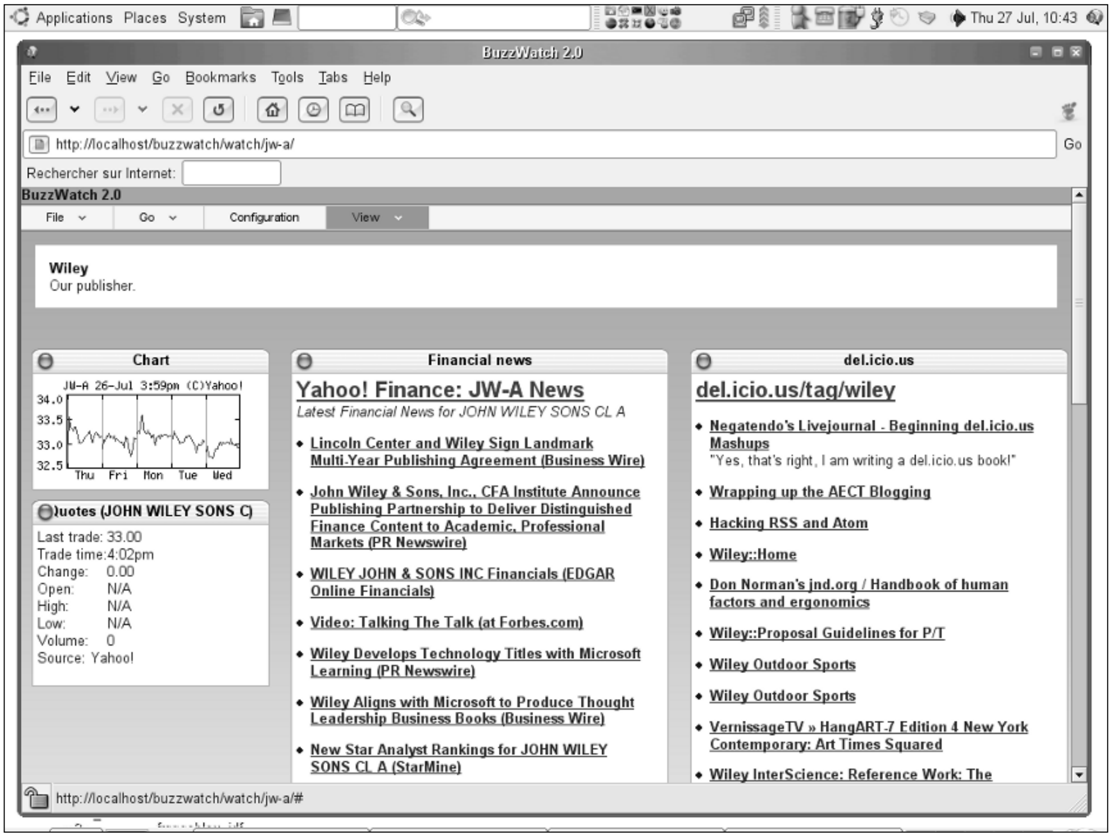
A page is composed of a menu bar, the title and description area, and four panels containing:

- ❑ A quotation chart
- ❑ The quotes
- ❑ The financial news
- ❑ The latest deli.cio.us bookmarks

The menu bar is composed of four menu items:

- ❑ File, with two subitems to save a page on the server and create a new one
- ❑ Go, with a sub item per existing page
- ❑ Configuration, which opens a new panel to edit the stock symbol and the tag
- ❑ View, to define which of the four panels should be displayed

Figure 1-1 shows a sample page with all these elements.



http://localhost/buzzwatch/watch/jw-a/

BuzzWatch 2.0

Wiley
Our publisher.

JW-A 26-Jul 3:59pm (C)Yahoo!

Quotes (JOHN WILEY SONS C)

Last trade: 33.00
Trade time: 4:02pm
Change: 0.00
Open: N/A
High: N/A
Low: N/A
Volume: 0
Source: Yahoo!

Yahoo! Finance: JW-A News
Latest Financial News for JOHN WILEY SONS CL A

- Lincoln Center and Wiley Sign Landmark Multi-Year Publishing Agreement (Business Wire)
- John Wiley & Sons, Inc., CFA Institute Announce Publishing Partnership to Deliver Distinguished Finance Content to Academic, Professional Markets (PR Newswire)
- WILEY JOHN & SONS INC Financials (EDGAR Online Financials)
- Video: Talking The Talk (at Forbes.com)
- Wiley Develops Technology Titles with Microsoft Learning (PR Newswire)
- Wiley Aligns with Microsoft to Produce Thought Leadership Business Books (Business Wire)
- New Star Analyst Rankings for JOHN WILEY SONS CL A (StarMine)

del.icio.us/tag/wiley

- Negatendo's Livejournal - Beginning del.icio.us Mashups
  "Yes, that's right, I am writing a del.icio.us book!"
- Wrapping up the AECT Blogging
- Hacking RSS and Atom
- Wiley::Home
- Don Norman's jnd.org / Handbook of human factors and ergonomics
- Wiley::Proposal Guidelines for P/T
- Wiley Outdoor Sports
- Wiley Outdoor Sports
- VernissageTV » HangART-7 Edition 4 New York Contemporary: Art Times Squared
- Wiley InterScience: Reference Work: The

http://localhost/buzzwatch/watch/jw-a/#

Figure 1-1

When you open BuzzWatch, the page is empty and the Go submenu is open so that you can choose an existing page, as shown in Figure 1-2. Of course, you can also open the File menu and create a new one to get started.

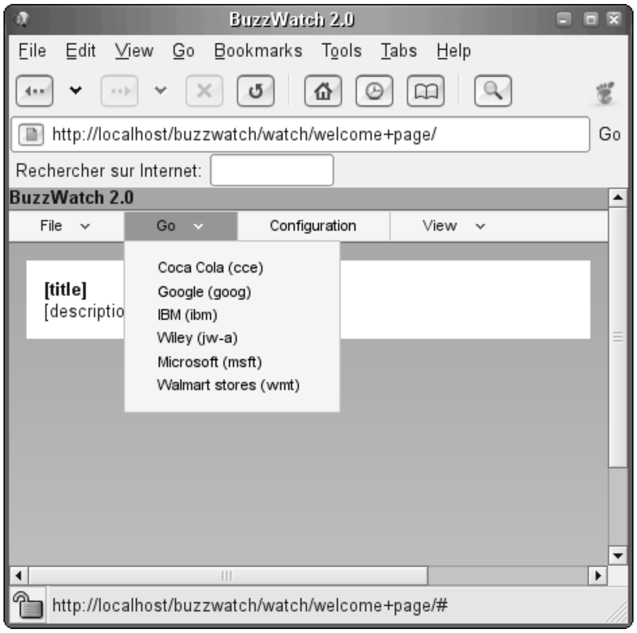


Figure 1-2

Charting the Landscape

With such a user requirement, you have numerous ways to reach your goal. The first big decision to make is the technical architecture. Client side, the obvious choice for this book is Ajax. The term *Ajax* used to be an acronym for Asynchronous JavaScript and XML, but it is now used to designate a whole set of techniques to develop rich web applications using today's browsers, and Ajax no longer always uses XML nor asynchronous exchanges.

You learn more about Ajax in Chapter 3. Other options include using Flash (which isn't covered in this book) and alternative technologies such as XUL, Open Lazlo, and XAML, which are described in Chapter 5 and 6.

After having decided that BuzzWatch relies on JavaScript client side, you need to choose which Ajax libraries you want to rely on. With more than one hundred libraries around, this isn't the easiest part of the job! BuzzWatch has set its choice on the Yahoo! User Interface (YUI) for a couple of reasons: this API is still relatively small but it covers most of what you need when developing Ajax applications. It can be used both to add action to existing (X)HTML elements and to add totally new content and controls to your application. YUI is also well documented, actively maintained, and has a lively mailing list to which you can post your questions. In addition to YUI, BuzzWatch uses JKL.ParseXML, a library that avoids the tedious job of using the DOM API to parse and create XML documents.

A big benefit of libraries such as the YUI is that they hide most of the differences between the JavaScript implementations available in modern browsers. Most Web 1.0 scripts include a huge number of tests that check browser types and versions to behave differently. The YUI probably includes quite a number *of these tests, but it takes the burden off the shoulders of application developers. You will still need to test your scripts very frequently against different browsers to use JavaScript features that work on all of them, but cases where you need to test and write different instructions depending on the browser become exceptional. The BuzzWatch application has only one such test.*

Although a lot of emphasis is put on JavaScript, Ajax applications rely heavily on equally important technologies, which are:

- ❑ XML (Extensible Markup Language), a technology that has become the lingua franca used to exchange data over the Internet. You learn more about XML and its alternatives in Chapter 8.
- ❑ HTML (HyperText Markup Language) or its XML flavor XHTML, the markup language used to publish documents on the Web from its very beginning. You learn more about HTML in Chapter 2.
- ❑ CSS (cascading style sheets), a simple mechanism used to define the presentation of HTML and XML document. CSS is covered with HTML in Chapter 2.
- ❑ HTTP (Hypertext Transfer Protocol), the main application protocol used to communicate between Web clients and servers. HTTP by itself is covered in Chapter 7, and using HTTP to exchange XML documents is detailed in Chapter 11.
- ❑ URIs (and URLs), the identifiers that are the names and addresses of web resources. You learn more about URIs in Chapter 7.

The formats used by sites such as del.icio.us and Yahoo! Finance to publish their headlines are XML formats know as *syndication* formats. You will learn everything about these formats in Chapter 9 and see how you can create your own syndication channels in Chapter 14.

Like any client server application, Web 2.0 applications have also a server side, and the choice of technologies to use server side is even more open than client side. As a developer, you cannot impose a specific environment or browser client side; you must count on what is installed by your users and that's often a severe restriction. Server side, on the contrary, you or your organization decide which platforms, operating system, programming language, frameworks and libraries will be used. The choice that has the most impact on the architecture of your application is usually the programming language. Server side, any programming language can be used to implement Web applications and popular choices include scripting languages such as PHP, Perl, Python, and Ruby and interpreted languages such as Java and C#.

BuzzWatch has decided to use PHP. This doesn't mean that the authors of this book believe that PHP is a better language, and in the course of the book we try to be as agnostic as possible and provide examples using a number of different programming languages. The choice for this first example had to be a language easy to read and install in case you want to try it for yourself and, because of its wide acceptance, this is a domain where PHP really shines.

Being a modern application, BuzzWatch uses the latest version of PHP, PHP version 5, which comes with much improved support for XML. To cache the information gathered from external sources, BuzzWatch relies on the Pear package named Cache_Lite (Pear is a PHP package repository similar to Perl's CPAN). To make things easier to install and administer and avoid relying on an external database, BuzzWatch is also using the PHP SQLite module (SQLite is a zero-admin embedded SQL database) to store its data.

To give you an idea of the complexity of BuzzWatch and of the split between technologies, the first version that you will see in the next section is composed of approximately 700 lines of JavaScript, 200 lines of XHTML, 150 lines of PHP, and 130 lines of CSS. This proportion is dependent of implementation choices and could vary a lot if different choices were taken.

Exploring Behind the Scene

One of the good things with client server applications is that you can easily spy on them and look at what they exchange. A still better thing with HTTP is that this is a text-oriented protocol and that most of what is exchanged is readable (with the exception, of course, of binary documents such as images, PDF files, and Microsoft Office and multimedia documents). To understand what is happening behind the scene, you can use two very interesting tools: the web server log and HTTP traces captured by tools such as HTTPTracer (which is discussed in more detail in Chapter 7), and tcpflow (which is covered in Chapter 11). The web server log is used as a summary of the exchanges, and the TCP traces provide all the details you need to understand what's going on.

If you are just scanning this chapter to get the big picture, following these examples through the printed code snippets and traces will be enough. Otherwise, it is time to install version 1 of BuzzWatch, available on this book's web site at `www.Wrox.com`. Note that you will see four different versions of BuzzWatch in this chapter. The one that you should install at this point is version 1.0.

If BuzzWatch is installed on your workstation and you open the location `http://localhost/buzzwatch/` in your favorite web browser, the following first series of exchanges will be logged in your web server's log before the page is displayed, and you can choose a first destination or create a new page:

```
12:35:59 200 GET /buzzwatch/index.html (text/html)
12:35:59 200 GET /buzzwatch/yui/yahoo/yahoo.js (application/x-javascript)
12:35:59 200 GET /buzzwatch/yui/event/event.js (application/x-javascript)
12:35:59 200 GET /buzzwatch/yui/dom/dom.js (application/x-javascript)
12:35:59 200 GET /buzzwatch/yui/dragdrop/dragdrop.js (application/x-javascript)
12:35:59 200 GET /buzzwatch/yui/animation/animation.js (application/x-javascript)
12:35:59 200 GET /buzzwatch/yui/container/container.js (application/x-javascript)
12:35:59 200 GET /buzzwatch/yui/connection/connection.js (application/x-javascript)
12:35:59 200 GET /buzzwatch/yui/menu/menu.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/XML/ObjTree.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/menuBar.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/script.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/panels.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/config.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/editInPlace.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/controller.js (application/x-javascript)
12:36:00 200 GET /buzzwatch/buzzWatch.css (text/css)
12:36:00 200 GET /buzzwatch/yui/reset/reset.css (text/css)
12:36:00 200 GET /buzzwatch/yui/fonts/fonts.css (text/css)
12:36:00 200 GET /buzzwatch/yui/menu/assets/menu.css (text/css)
12:36:00 200 GET /buzzwatch/yui-container-css/example.css (text/css)
12:36:00 200 GET /buzzwatch/yui/container/assets/container.css (text/css)
12:36:00 200 GET /buzzwatch/yui-container-css/panel-aqua.css (text/css)
12:36:00 200 GET /buzzwatch/img/bg.png (image/png)
12:36:00 200 GET /buzzwatch/img/aqua-hd-bg.gif (image/gif)
12:36:00 200 GET /buzzwatch/img/aqua-hd-lt.gif (image/gif)
```

```
12:36:00 200 GET /buzzwatch/img/aqua-hd-rt.gif (image/gif)
12:36:00 200 GET /buzzwatch/img/aqua-hd-close.gif (image/gif)
12:37:00 200 GET /buzzwatch/watch.php (application/xml)
```

This log uses a custom log format defined as `"%{%T}t %>s %m %U%q (%{Content-Type}o)"` on an Apache 2.*x* web server. This format would not be appropriate for a production server where you'll want to see important information such as the date and the client IP addresses, but it has the benefit of being easy to print in this book and contains the minimum information needed to understand what's going on.

The whole exchange is triggered by the first request, which is executed when you open the page in your browser:

```
12:35:59 200 GET /buzzwatch/index.html (text/html)
```

If you look at what is exchanged on the wire, you'll find a request sent by your browser to the web server:

```
GET /buzzwatch/ HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.4) Gecko/20060608
Ubuntu/dapper-security Epiphany/2.14 Firefox/1.5.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
```

Note that the line breaks between `20060608` *and* `Ubuntu` *and between* `text/html;` *and* `q=0.9,text/plain;` *have been added for readability reasons and are not present in the exchange over the wire.*

In this request, the browser is asking to get (`GET` in the first line is the HTTP request) a page at location `/buzzwatch/` using version 1.1 of HTTP and contacting the host `localhost`. The remaining lines are called HTTP headers and contain more information about the browser, the kind of resources it can handle, and the way it would like cached data to be handled. The answer from the server to the web browser is:

```
HTTP/1.1 200 OK
Date: Fri, 21 Jul 2006 10:35:59 GMT
Server: Apache/2.0.55 (Ubuntu) PHP/5.1.2
Last-Modified: Thu, 20 Jul 2006 18:05:26 GMT
ETag: "240449-2985-3970c580"
Accept-Ranges: bytes
Content-Length: 10629
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
          "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
    <head>
```

```
            <title>BuzzWatch 2.0</title>
            <script type="text/javascript" src="yui/yahoo/yahoo.js"> </script>
            <script type="text/javascript" src="yui/event/event.js"> </script>
            <script type="text/javascript" src="yui/dom/dom.js"> </script>
            <script type="text/javascript" src="yui/dragdrop/dragdrop.js"> </script>
            <script type="text/javascript" src="yui/animation/animation.js"> </script>
            <script type="text/javascript" src="yui/container/container.js"> </script>
            <script type="text/javascript" src="yui/connection/connection.js">
            </script>
            <script type="text/javascript" src="yui/menu/menu.js"> </script>
            <script type="text/javascript" src="XML/ObjTree.js"> </script>
            <script type="text/javascript" src="menuBar.js"> </script>
            <script type="text/javascript" src="script.js"> </script>
            <script type="text/javascript" src="panels.js"> </script>
            <script type="text/javascript" src="config.js"> </script>
            <script type="text/javascript" src="editInPlace.js"> </script>
            <script type="text/javascript" src="controller.js"> </script>

            <link rel="stylesheet" type="text/css" href="buzzWatch.css"/>
            <link rel="stylesheet" type="text/css" href="yui/reset/reset.css"/>
            <link rel="stylesheet" type="text/css" href="yui/fonts/fonts.css"/>
            <link rel="stylesheet" type="text/css" href="yui/menu/assets/menu.css"/>
            <link rel="stylesheet" type="text/css"
                  href="yui-container-css/example.css"/>
            <link rel="stylesheet" type="text/css"
                  href="yui/container/assets/container.css"/>
            <link rel="stylesheet" type="text/css"
                  href="yui-container-css/panel-aqua.css"/>

        </head>
        <body>
    .
    .
    .
        </body>
    </html>
```

The answer is composed of HTTP headers followed by the actual content. The first line gives the status of the transaction. Here the server answers that it's okay to exchange using HTTP version 1.1 and returns a code equal to 200 with its textual meaning (`OK`). The following headers are information about the server itself and the document that is returned, including its media type (`text/html`) and encoding (`UTF-8`). The first line of the document following the headers is called a `DOCTYPE` definition. Here, this `DOCTYPE` definition indicates that the document uses XHTML 1.1. (X)HTML documents are composed of a `head` and a `body` section. The `body` section has been cut from this listing to keep it short. The `head` section contains a title and a number of references to cascading style sheets (CSS) and JavaScript scripts.

In theory, the media type of XHTML documents is `application/xhtml+xml`*. Unfortunately, Internet Explorer does not support this media type and refuses to display documents sent with this type. A common practice is thus to serve XHTML documents with a media type of* `text/html`*.*

When receiving such a document, a browser that supports CSS and JavaScript (which is true of modern graphical browsers such as Internet Explorer, Firefox, Opera, Safari, and Konqueror if their users have not disabled JavaScript) downloads all the CSS files and JavaScript scripts referenced in the `head` section and the images and multimedia documents referenced in the `body` section. This behavior explains the burst of exchanges logged by the server from the second line to the line preceding the last line. These exchanges follow the same pattern that was used to download the initial XHTML document.

The scripts are executed as soon as they are loaded by the browser. However, most of the actions that are performed by these scripts require that the page and all its scripts and CSS have been loaded. Executing action before that stage would mean that they cannot be sure that the other scripts on which they rely have already been loaded, and also that they don't know if the HTML document itself is complete. A common pattern is thus to perform declarations in each script and trigger their initialization and the beginning of the real work with a `load` event that is sent by the browser when everything has been loaded. A typical example of this pattern is `script.js`, the BuzzWatch main script:

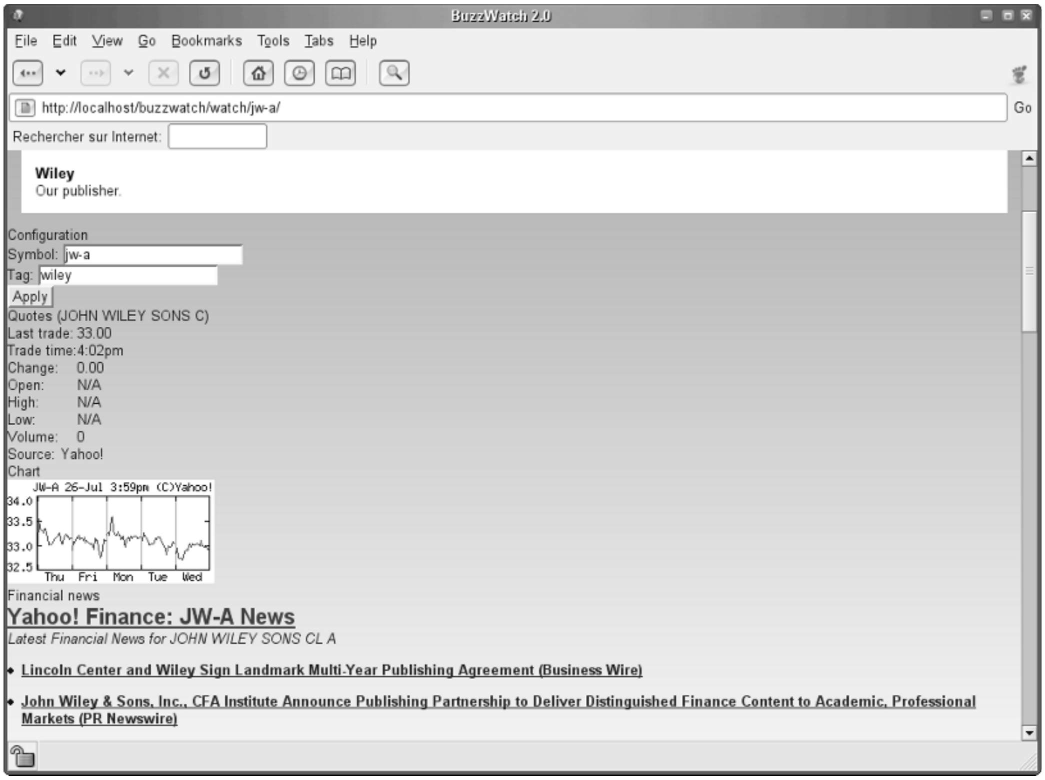
```
YAHOO.namespace("buzzWatch");
YAHOO.namespace("editInPlace");

function init() {

  initMenuBar();
  initConfig();
  initPanels();
  initEditInPlace();
  initController();

}
.
.
.
YAHOO.util.Event.addListener(window, "load", init);
```

The first two lines are YUI-specific initializations. The next lines define an `init` function that performs the initialization of the BuzzWatch application, and the last one uses the YUI event utility to require that the `init` function is called when the page is loaded. If you freeze your browser after the page has been loaded and before the load event has been propagated to the different function that performs the initialization of the application, you'll see a page (Figure 1-3) that looks very different from what you see after the initialization, and the difference between these two views is the domain of Ajax programming.

If you want to reproduce this figure, there are a couple of ways to freeze your browser after loading and before initialization. The first is simply to disable JavaScript before you load the page. The second is to use a JavaScript debugger available for your browser and add a breakpoint at the beginning of the `init` *function.*

You wouldn't expect to be walked through the 700 lines of JavaScript that power BuzzWatch in this first chapter, but you're probably curious to see what type of tasks are performed in Web 2.0 applications. To categorize these tasks, you can consider that they fall into three main categories:

- Changing the document that is displayed
- Reacting to user interaction
- Interacting with web servers


BuzzWatch 2.0
File Edit View Go Bookmarks Tools Tabs Help
Go
Apply
JW-A 26-Jul 3:59pm (C)Yahoo!
34.0
33.5
33.0
32.5
Thu Fri Mon Tue Wed

http://localhost/buzzwatch/watch/jw-a/

Rechercher sur Internet:

Wiley
Our publisher.

Configuration
Symbol: jw-a
Tag: wiley
Quotes (JOHN WILEY SONS C)
Last trade: 33.00
Trade time: 4:02pm
Change: 0.00
Open: N/A
High: N/A
Low: N/A
Volume: 0
Source: Yahoo!
Chart
Financial news

Yahoo! Finance: JW-A News

Latest Financial News for JOHN WILEY SONS CL A

- **Lincoln Center and Wiley Sign Landmark Multi-Year Publishing Agreement (Business Wire)**
- **John Wiley & Sons, Inc., CFA Institute Announce Publishing Partnership to Deliver Distinguished Finance Content to Academic, Professional Markets (PR Newswire)**

Figure 1-3

These categories are tightly coupled and a user interaction often triggers an exchange with a server that leads to a modification in the document. To illustrate the point, you are invited to follow some of the actions that are performed after the document is loaded. One of the functions called in `init` is `initMenuBar()`. Most of the instructions in this function are copied from examples coming with the YUI. (Unlike most of the other modules that are nicely wrapped in classes that you just have to instantiate, the menu bar module requires quite a few instances of copying and pasting JavaScript instructions.) Among these instructions, the ones that operate the magic and bring the menu bar to life in the file `menubar.js` are:

```
    var oMenuBar = new YAHOO.widget.MenuBar(
      "menubar",
      { fixedcenter: false }
    );
    oMenuBar.render();

    YAHOO.buzzWatch.menuBar = oMenuBar;
    YAHOO.buzzWatch.menuGo = oMenuBar.getItem(1);
```

The first instruction creates a menu bar object. The second one, `oMenuBar.render()`, applies the modifications needed to instantiate the menu bar in the HTML document. The first argument of the `YAHOO.widget.MenuBar` constructor call, `"menubar"`, is the identifier of an element of the HTML document that will be used to create the menu bar. In other words, the structure of the menu bar will be derived from the structure of this element, and the content of this element will be replaced when the menu bar is rendered by a totally new content that YUI will build so that it displays like a menu bar. The HTML element that describes the menu bar in `index.html` is:

```
<div id="menubar" class="yuimenubar">
  <div class="bd">
    <ul class="first-of-type">
      <li class="yuimenubaritem" id="menubar.file">
        <span>File</span>
        <div class="yuimenu">
          <div class="bd">
            <ul class="first-of-type">
              <li class="yuimenuitem" id="menubar.file.new">New</li>
              <li class="yuimenuitem" id="menubar.file.save">Save</li>
            </ul>
          </div>
        </div>
      </li>
      <li class="yuimenubaritem" id="menubar.go">
        <span>Go</span>
        <div class="yuimenu">
          <div class="bd">
            <ul class="first-of-type">
              <li class="yuimenuitem">dummy</li>
            </ul>
          </div>
        </div>
      </li>
      <li class="yuimenubaritem" id="menubar.config">Configuration</li>
      <li class="yuimenubaritem" id="menubar.view">
        <span>View</span>
        <div class="yuimenu">
          <div class="bd">
            <ul class="first-of-type">
              <li class="yuimenuitem" id="menubar.view.yahoofinance">
                <span>Yahoo! FINANCE</span>
                <div id="widgets.Yahoo.finance" class="yuimenu">
                  <div class="bd">
                    <ul>
                      <li class="yuimenuitem"
                          id="menubar.view.yahoofinance.quotes">Quotes</li>
                      <li class="yuimenuitem"
                          id="menubar.view.yahoofinance.chart">Chart</li>
                      <li class="yuimenuitem"
                          id="menubar.view.yahoofinance.news">News</li>
                    </ul>
                  </div>
                </div>
              </li>
              <li class="yuimenuitem"
                  id="menubar.view.delicious">del.icio.us</li>
            </ul>
          </div>
```

```
            </div>
          </li>
        </ul>
      </div>
    </div>
```

If you are familiar with HTML, you will have recognized an HTML division (`div`) that embeds a multi-level unordered list (`ul`). This multilevel list defines the structure of the menu bar and follows a set of conventions that are described in the YUI library. You may also notice many `id` attributes, such as `id="menubar.go"`. These are identifiers that will be used by our scripts to attach actions to the menu items. Speaking of the Go menu, identified by `menubar.go`, you may also notice that its content includes only a dummy item. This is because you populate the content of this menu with dynamic information gathered from the web server.

This is done by another BuzzWatch class that is called the controller. During its initialization, the controller performs the following actions in `controller.js`:

```
    YAHOO.util.Event.addListener(
      "menubar.file.save",
      "click", this.save,
      this,
      true);
    YAHOO.util.Event.addListener(
      "menubar.file.new",
      "click",
      this._new,
      this,
      true);
    YAHOO.buzzWatch.menuGo.cfg.getProperty("submenu").show();
    this.loadList();
```

The first two instructions belong to the user interaction category and they assign actions on the `click` events of the `file.save` and `file.new` menu items for which this class is responsible. The third one tells the application to open and show the Go menu. This is the instruction that opens this menu when you load the page. The last instruction calls the `loadList` method, which is defined as:

```
  BuzzWatchController.prototype.loadList = function() {
    YAHOO.util.Connect.asyncRequest('GET', "watch.php", this.callbackList);
  }
```

With this method, you leave the domain of changing the way the document is presented to enter the domain of web server interaction. The YUI connect module is a wrapper around the mythic `XMLHTTPRequest` object that is the heart of Ajax applications. As already mentioned, the big benefit of using such a wrapper is that it hides the differences between implementations and makes all the browsers pretty much equal. Here, the controller uses an asynchronous request (asynchronous meaning that the browser does not wait until the response from the server comes back but calls a method depending on the result of the request). The first parameter, `'GET'`, is the same HTTP request code that you saw in the HTTP traces. The second parameter, `"watch.php"` is the URL of the resource to fetch. This URL doesn't start with a URI scheme such as `http://` and it is what is called a *relative URI*. This means that its address is evaluated relatively to the current page. If you've accessed this page as `http://localhost/buzzwatch/`, this URL is equivalent to `http://localhost/buzzwatch/watch.php`. The last parameter defines what needs to be done when the answer comes back. It is a reference to the following definition:

```
    this.callbackList = {
        success: this.handleListSuccess,
        failure: this.handleListFailure,
        scope: this
    };
```

Basically, this means that if the response is `OK`, `handleListSuccess` will be called and if not, `handleListFailure` will be called. In both cases, the context with which these methods are called is the context of this object (this being the controller). When `YAHOO.util.Connect.asyncRequest` is called, the YUI sends a request to the server and this request is what was logged as the last line of your web server log snippet earlier:

```
12:37:00 200 GET /buzzwatch/watch.php (application/xml)
```

The request sent by the library is similar to the one that you have already seen, and there is nothing to tell you that it was sent by the YUI:

```
GET /buzzwatch/watch.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.4) Gecko/20060608
Ubuntu/dapper-security Epiphany/2.14 Firefox/1.5.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Serving content coming from SQL databases as XML is a very frequent task for Web 2.0 applications. The PHP script that does that for BuzzWatch has been developed to show that this is simpler than you may think if you want to do it by hand, but there are also a number of generic tools that can do that for you. This subject is covered in Chapter 12.

The HTML document was a static file, and now we are accessing a dynamic PHP script on the server. This PHP script queries the SQLite database, retrieves a list of available watches, and sends this list as XML. This PHP script, `watch.php`, can do more than that: it can also save a new watch and display a single one. Its main part is:

```
header("Cache-Control: max-age=60");
header("Content-type: application/xml");
echo '<?xml version="1.0" encoding="utf-8"?>';
if (strlen($HTTP_RAW_POST_DATA)>0) {
  write();
} else if ($_GET['name']) {
  readOne();
} else {
  listAll();
}
```

The first instructions are to set HTTP headers like those you saw in the previous traces. The first one sets the time the document should be considered as fresh. Here, BuzzWatch considers that users can frequently update the database and that the document shouldn't be considered fresh after more than a minute (60 seconds). The second header says that the media type is `application/xml`. The third instruction outputs the XML declaration. The tests that follow are to check in which case we are. The first one checks if data has been posted, in which case you would be handling a request to save a document. The second one checks if you have received a parameter with a `GET` request. This isn't the case here so the `listAll` function will be executed:

```
function listAll() {
  $db=openDb();
  echo "<watches>";
  $query = $db->query("SELECT * from watches order by symbol", SQLITE_ASSOC);
  while ($row = $query->fetch(SQLITE_ASSOC)) {
    displayOne($row);
  }
  echo "</watches>";
}
```

This code starts by calling a function that opens the database. This function has been written so that if the database doesn't exist, it is created and if the table that contains the data doesn't exist, it is also created. This insures an auto-install feature for BuzzWatch! The next instruction is to send the `<watches>` start tag that embeds the different records. The SQL query selects all the rows in the table named watches and orders them by symbols and loops over the rows that are returned by the request and call the following function for each row:

```
function displayOne($row) {
  $xml = simplexml_load_string(
         "<watch><symbol/><tag/><title/><description/></watch>");
  $xml->symbol=$row['symbol'];
  $xml->tag=$row['tag'];
  $xml->title=$row['title'];
  $xml->description=$row['description'];
  $asXML = $xml->asXML();
  print substr($asXML, strpos($asXML, '<', 2));
}
```

This function uses the handy SimpleXML PHP5 module to populate an XML document. The first instruction creates an empty document with the structure that needs to be sent, and the next ones assign values into this document as if the document was a PHP object. The result is then serialized as XML and, because you want an XML snippet instead of a full XML document, you remove the XML declaration before sending the snippet.

This is a hack, but a safe one. This hack is needed because BuzzWatch has been developed with PHP 5.1.2. In PHP 5.1.3, new features have been added to SimpleXML that enable you to add nodes to a document, so the full document could easily be built entirely with SimpleXML and serialized in a single step.

The result of this script is to send a response, which is:

```
HTTP/1.1 200 OK
Date: Fri, 21 Jul 2006 10:37:00 GMT
Server: Apache/2.0.55 (Ubuntu) PHP/5.1.2
X-Powered-By: PHP/5.1.2
Cache-Control: max-age=60
```

```
Content-Length: 831
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: application/xml
X-Pad: avoid browser bug

<?xml version="1.0" encoding="utf-8"?>
<watches><watch><symbol>cce</symbol><tag>coke</tag><title>Coca
Cola</title><description>Find their secret...</description></watch>
<watch><symbol>goog</symbol><tag>google</tag><title>Google</title><description>If
you don't know them, google them!</description></watch>
<watch><symbol>ibm</symbol><tag>ibm</tag><title>IBM</title><description>Big
blue...</description></watch>
<watch><symbol>jw-a</symbol><tag>wiley</tag><title>Wiley</title><description>Our
publisher.</description></watch>
<watch><symbol>msft</symbol><tag>microsoft</tag><title>Microsoft</title>
<description>The company we love to hate.</description></watch>
<watch><symbol>wmt</symbol><tag>walmart</tag><title>Walmart
stores</title><description>You can't be number one and have only
friends...</description></watch>
</watches>
```

After this small incursion into PHP land, you need to switch back to JavaScript. When this response reaches the browser, the YUI calls back the `handleListSuccess` method in `controller.js` as instructed:

```
BuzzWatchController.prototype.handleListSuccess = function(o) {
  if(o.responseText !== undefined){
    this.setTimeout(getMaxAge(o));
    var xotree = new XML.ObjTree();
    xotree.force_array = ["watch"];
    var tree = xotree.parseDOM( o.responseXML.documentElement );
    var menu = YAHOO.buzzWatch.menuGo.cfg.getProperty("submenu");
    while (menu.getItemGroups().length > 0) {
      var menuItem = menu.removeItem(0);
      menuItem.destroy();
    }
    var watches = tree.watches.watch;
    for (var i=0; i< watches.length; i++) {
      var watch = watches[i];
      var menuItem = new YAHOO.widget.MenuItem(
        watch.title + ' (' + watch.symbol+')',
        {}
      );
      menu.addItem(menuItem);
      menuItem.clickEvent.subscribe(
        this.loadSymbol,
        watch.symbol,
        false
      );
    }
    menu.render();
  }
}
```

The first instruction is to test whether there is a response. Obviously, BuzzWatch deserves better error handling when this isn't the case. The next instruction is to set a timer with the value gathered in the `Cache-Control: max-age` HTTP header that has been set by the PHP script. This is done by a simple BuzzWatch function, `getMaxAge()` in `script.js`, that parses this header using JavaScript regular expressions:

```
function getMaxAge(oResponse) {
  var cacheControl = oResponse.getResponseHeader['Cache-Control'];
  if (!cacheControl)
    return undefined;
  var result;
  if (result=cacheControl.match(/^.*max-age=(\d+)(;.*)?$/))
    return result[1];
  return undefined;
}
```

There is an unfortunate tendency among developers to reinvent the wheel. Applied to web development, this tendency often leads people to reinvent HTTP features. Many developers would have included an XML attribute in the XML document to define when the document should be refreshed. The cache control header would still have been needed because it is used by cache managers in the browser and in caching proxies that might sit between the server and the browser. Defining the same information twice (once in the cache control header and once in the XML document) is always error prone: chances are that when you'll update the value in one of the locations, you forget to update it in the other one, and it's preferable to avoid this duplication when possible!

Back to the `handleListSuccess` method. The next three instructions are for initializing the JKL.ParseXML library with the XML that you've received from the server. This library is similar to SimpleXML in PHP and it will release you from the burden of using a low-level API such as the DOM to parse the XML document. After the last of these lines, `var tree = xotree.parseDOM( o.responseXML.documentElement );`, you have a JavaScript object in your variable `tree` that has the same structure than the XML document. The `handleListSuccess` method is used each time the list of watches is reloaded from the server and it needs to remove the previous content of the Go menu before inserting the content just received from the server. This is done by a loop that removes and destroys the menu items.

The next step is to feed the menu with data read in the XML response. The structure from this document is a `watches` root element with a number of `watch` sub-elements, and the script loops over these `watch` sub-elements. The JKL.ParseXML library, like most similar libraries, automatically creates an array when it finds repeated elements and it has been told by the instruction `xotree.force_array = ["watch"]` to create such an array even if there is only a single `watch` element. The loop creates a menu item for each `watch` element. The menu items that were created so far were created from existing (X)HTML elements, but the menu items that are created in this loop are created entirely in JavaScript. Each menu item is added to the menu and an event subscription is added to call the method `loadSymbol` of the current object (which is the controller) with the `watch symbol` as a parameter when the user clicks on this menu item.

At this stage, you have reached the point where the application is waiting for users to click one of these menu items, unless they decide to create a new watch. This is also the end of the exchanges between the browser and the server that correspond to the server log snippet shown earlier. If you click one of these items, the following exchanges are added to the server's log:

```
12:18:32 200 GET /buzzwatch/watch.php?name=jw-a (application/xml)
12:18:32 200 GET /buzzwatch/yahoo_quotes.php?tag=jw-a (application/xml)
12:18:32 200 GET /buzzwatch/yahoo_chart.php?tag=jw-a&
date=Sat%20Jul%2022%202006%2012:18:32%20GMT+0200%20(CEST) (image/png)
12:18:33 200 GET /buzzwatch/yahoo_finance_news.php?tag=jw-a (application/xml)
12:18:34 200 GET /buzzwatch/yui/menu/assets/menuchk8_dim_1.gif (image/gif)
12:18:33 200 GET /buzzwatch/delicious.php?tag=wiley (application/xml)
```

Note that the line break between jw-a& and date= has been added for readability reasons and is not present in the server's log.

This new burst of exchanges is triggered by clicking one of the menu items that were added dynamically. The action registered to this click is a call to the `loadSymbol` method in `controller.js`:

```
BuzzWatchController.prototype.loadSymbol = function(e, o, symbol) {
  YAHOO.buzzWatch.menuGo.cfg.getProperty("submenu").hide();
  YAHOO.buzzWatch.controller.load(symbol);
}
```

This method is what you would call in other programming languages a `static` or a `class` method: it isn't called in the context of an object or, in other words, the `this` object isn't available. It hides the Go submenu and calls the instance method `load` of the controller. This method is hardly more complicated:

```
BuzzWatchController.prototype.load = function(symbol) {
  if (symbol != undefined)
    this.symbol = symbol;
  if (this.symbol != undefined) {
    YAHOO.util.Connect.asyncRequest('GET', "watch.php?name="+this.symbol,
 this.callback);
  }
}
```

Its main action is the call to the `YAHOO.util.Connect.asyncRequest` that you already know. This call is what explains the first line in the server's log and fetches the definition of the watch that has been requested by the user. The request that is transmitted over the wire by this instruction is:

```
GET /buzzwatch/watch.php?name=jw-a HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; U; Linux i686; fr; rv:1.8.0.4) Gecko/20060608
Ubuntu/0.9.3 (Ubuntu) StumbleUpon/1.909 Firefox/1.5.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

It's time to replace your JavaScript hat with your PHP one. Server side, this uses the same `watch.php` script that you already know. The difference is that this time you fall in the second branch of the multiple `if...then...else` statement:

```
 } else if ($_GET['name']) {
         readOne();
```

This is because the URL is now /buzzwatch/watch.php?name=jw-a. What has been added after the question mark is called a *query string*. It contains parameters that are available in PHP scripts in the $_GET global variable. The function readOne is similar to what you've already seen with a single highly critical point to note:

```
function readOne() {
  $db=openDb();
  $query = $db->query(
    "SELECT * from watches where symbol='".
      sqlite_escape_string(trim($_GET['name']))."'"
    , SQLITE_ASSOC);
  if ($row = $query->fetch(SQLITE_ASSOC)) {
    displayOne($row);
  } else {
    $xml = simplexml_load_string("<watch/>");
    $asXML = $xml->asXML();
    print substr($asXML, strpos($asXML, '<', 2));
  }
}
```

Basically, the function selects a single row from the database and returns it as XML using the same displayOne function that you've already seen.

Have you found what is really critical in this function? The small detail that makes a difference between a function that hackers can easily exploit to delete your complete database and a function which is secure? As any web application powered by a SQL database, BuzzWatch is potentially vulnerable to the kind of attacks known as SQL injection. Instead of name=jw-a, a hacker could send the request:

```
name=jw-a';%20delete%20from%20watches;select%20*%20from%20watch%20where%20
symbol='jw-a
```

That's a very easy attack; the hacker would just have to type the URL in a browser. For this request, the value of $_GET['name'] is

```
jw-a'; delete from watches;select * from watch where symbol='jw-a
```

and if you use this value to create your SQL select without calling the sqlite_escape_string() function, you get the following request:

```
select * from watches where symbol=' jw-a'; delete from watches;select *
from watch where symbol='jw-a'
```

SQLite, like most SQL databases, uses the semicolon (;) as a separator between queries and executes on one but three queries and one of them, delete from watches, deletes all the data in your table.

You learn more about security in Chapter 18, but you should remember that the Internet is a jungle and that security should be on your mind at all times when building a web application. SQL injection is a good example of simple attacks that are easy to counter (escaping parameter values as you've seen here is a simple way to make sure that these values will be interpreted as single strings by the SQL database and won't leak out of the string into other SQL statements). Unfortunately, new web applications are rolled out every day that are vulnerable to SQL injection because their developers were not aware of this attack.

The HTTP answer to this request is:

```
HTTP/1.1 200 OK
Date: Sat, 22 Jul 2006 10:18:32 GMT
Server: Apache/2.0.55 (Ubuntu) PHP/5.1.2
X-Powered-By: PHP/5.1.2
Cache-Control: max-age=60
Content-Length: 152
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: application/xml
X-Pad: avoid browser bug

<?xml version="1.0" encoding="utf-8"?><watch>
<symbol>jw-a</symbol><tag>wiley</tag><title>Wiley</title>
<description>Our publisher.</description></watch>
```

Back to JavaScript. When the browser receives this answer, the YUI fires the callback routine in `controller.js` that has been set up in case of success:

```
BuzzWatchController.prototype.handleSuccess = function(o) {
  if(o.responseText !== undefined){
    var xotree = new XML.ObjTree();
    var tree = xotree.parseDOM( o.responseXML.documentElement );
    YAHOO.buzzWatch.config.set(tree.watch.symbol, tree.watch.tag);
    refreshPanels();
    YAHOO.buzzWatch.editInPlace.set('textTitle', tree.watch.title);
    YAHOO.buzzWatch.editInPlace.set('textDescription', tree.watch.description);
  }
```

This method relies again on the JKL.ParseXML library to manipulate the XML document that has JavaScript objects. It sets the symbol and tag attributes that are handled by another object, the `YAHOO.buzzWatch.config` object, refreshes the panels, and sets the title and description. The rest of the application is quite similar to what you've already seen. The `refreshPanels` method, for example, uses the `YAHOO.util.Connect.asyncRequest` method to fetch the XML information that is displayed in the panels. You now have a good idea of what's going on behind the scene to skip the rest of the detailed technical description of BuzzWatch, but there are a couple of questions that are still worth answering: Why does BuzzWatch have to cache the content that is aggregated? And how do you save watches into the database?

The server's log shows that, instead of retrieving aggregated data directly from the browser and download, for example, the RSS channel `http://del.icio.us/rss/tag/wiley` directly from del.icio.us, the browser fetches a cached copy on the BuzzWatch server (that's the line with `GET /buzzwatch/delicious.php?tag=wiley`). There are several reasons for that; the most important reason is that the browser would refuse to retrieve the RSS channel directly from del.icio.us and would raise an exception saying that your script isn't authorized to do so. Although you could find this restriction painful and pointless in that specific case (what harm is there in accessing public data from JavaScript?), this restriction is much needed to avoid allowing a script to access private information available to the browser. Without this restriction, a script executed from the public web could access and steal private information available behind a firewall or through the user's credentials.

This restriction is known as the same origin policy. In a nutshell, it means that a script served from a domain can only access resources from the same domain. In Chapter 15, you will see that this is a general issue for mashups and that data providers such as Google Maps and Yahoo! Maps have worked around the limitation by serving the scripts that decorate the maps from their own domain. If the script that downloads the del.icio.us RSS feed was served by del.icio.us, the script and the feed would belong to the same domain, and the same origin policy would not be violated. Unfortunately, this workaround requires that the data provider has anticipated the need, which is not often the case.

Because a script executed served from a domain can access only resources from this domain, you have to use a proxy to access the sources that you want to aggregate. A generic proxy like Apache's mod-proxy can be used, but implementing your own caching proxy as done by BuzzWatch is also an opportunity to change the format of the documents that are being served. For example, BuzzWatch is using stock quotes delivered by Yahoo! as CSV (comma-separated values) documents, and its caching proxy does the conversion from CSV to XML. Converting non-XML data into XML is a very common pattern for Web 2.0 (covered in Chapter 13). You've already seen an example of such a conversion with SQL accesses, and most of the time these conversions are quite straightforward. The conversion from CSV to XML uses SimpleXML and a regular expression; it is located in `yahoo_quotes.php` and is as simple as this:

```
function get_xml($url) {
  $csv = file_get_contents($url, "r");
  $a = '"([^"]*)"';
  $n = '([^,]*)';
  $pattern = "/^$a,$a,$n,$a,$a,$n,$n,$n,$n,$n$/";
  $match = preg_match($pattern, $csv, $matches);
  $xml = new SimpleXMLElement(
         "<quote><symbol/><name/><lastTrade><price/><date/><time/>".
         "</lastTrade><change/><open/><high/><low/><volume/></quote>");
  $xml->symbol = $matches[1];
  $xml->name = $matches[2];
  $xml->lastTrade->price = $matches[3];
  $xml->lastTrade->date = $matches[4];
  $xml->lastTrade->time = $matches[5];
  $xml->change = $matches[6];
  $xml->open = $matches[7];
  $xml->high = $matches[8];
  $xml->low = $matches[9];
  $xml->volume = $matches[10];
  return $xml->asXML();
}
```

The second question was to see how documents can be saved on the server. The exchange is logged as:

```
10:18:43 200 POST /buzzwatch/watch.php (application/xml)
```

It is initiated by the JavaScript method attached to the Save menu item in `controller.js`:

```
BuzzWatchController.prototype.save = function() {
  var xotree = new XML.ObjTree();
  var tree = {
    watch: {
      symbol: YAHOO.buzzWatch.config.symbol,
      tag: YAHOO.buzzWatch.config.tag,
      title: YAHOO.buzzWatch.editInPlace.get('textTitle'),
        description: YAHOO.buzzWatch.editInPlace.get('textDescription')
      }
    };
    var o = YAHOO.util.Connect.getConnectionObject();
    o.conn.open("POST", "watch.php", true);
    YAHOO.util.Connect.initHeader('Content-Type','application/xml');
    YAHOO.util.Connect.setHeader(o);
    YAHOO.util.Connect.handleReadyState(o, this.saveCallback);
    o.conn.send(xotree.writeXML(tree));
  }
```

This method uses the JKL.ParseXML library to create a XML document from a JavaScript object: the `tree` object is a standard JavaScript object containing the values that will constitute the XML document and the conversion itself is done by the instruction `xotree.writeXML(tree)`. Chapter 12 covers in detail the different ways to exchange XML over HTTP. All you need to know for the moment is that, like programming objects, HTTP resources have a number of methods attached to them and that each method corresponds to a different action that is specified by the HTTP specification. Up to now, BuzzWatch had used only the `GET` method, but to save watches, you will use a `POST` method (that was clearly visible in the server's log). Posting XML documents with this version of the YUI is slightly more verbose than getting one and requires six statements. The first one (`var o = YAHOO.util.Connect.getConnectionObject();`) gets a new connection object. The second one opens the connection to the server. The third and fourth ones initialize and set the header that describes the media type. The fifth one sets up the callback handler, and the last one sends the request together with the XML document. The request sent by the browser is:

```
POST /buzzwatch/watch.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; U; Linux i686; fr; rv:1.8.0.4) Gecko/20060608
Ubuntu/0.9.3 (Ubuntu) StumbleUpon/1.909 Firefox/1.5.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Content-Type: application/xml
Content-Length: 181
Pragma: no-cache
Cache-Control: no-cache

<?xml version="1.0" encoding="UTF-8" ?>
<watch>
<symbol>msft</symbol>
<tag>microsoft</tag>
<title>Microsoft</title>
<description>The company we love to hate.</description>
</watch>
```

Server side, it fires the same `watch.php` script that you've already seen but reaches a branch of the main if/then/else test that you've not explored yet:

```
    if (strlen($HTTP_RAW_POST_DATA)>0) {
      write();
    }
```

The `$HTTP_RAW_POST_DATA` variable has been initialized to contain the data that is received by HTTP `POST` methods. In that case, it contains the XML document that was sent by the browser, and its size is obviously greater than zero, which causes the `write()` function to be called:

watch.php – v1.0

```
    function write() {
      global $HTTP_RAW_POST_DATA;
      $db=openDb();
      $dom = new DOMDocument();
      $dom->loadXML($HTTP_RAW_POST_DATA);
      if (!$dom->relaxNGValidate ( 'watch.rng')) {
          die("unvalid document");
      }
      $xml = simplexml_import_dom($dom);
      foreach ($xml->children() as $element) {
        $element['escaped'] = sqlite_escape_string(trim($element));
      }
      //echo $xml->asXML();
      $query = $db->query(
        "SELECT symbol from watches where symbol='".
          $xml->symbol['escaped'].
          "'",
        SQLITE_NUM);
      $req = "";
      if ($query->fetch()) {
        $req="update watches set ";
        $req .= "tag='".$xml->tag['escaped']."', ";
        $req .= "title='".$xml->title['escaped']."', ";
        $req .= "description='".$xml->description['escaped']."' ";
        $req .= "where symbol='".$xml->symbol['escaped']."'";
      } else {
        $req="insert into watches (symbol, tag, title, description) values (";
        $req .= "'".$xml->symbol['escaped']."', ";
        $req .= "'".$xml->tag['escaped']."', ";
        $req .= "'".$xml->title['escaped']."', ";
        $req .= "'".$xml->description['escaped']."')";
      }
      //echo $req;
      $db->queryExec($req);
      echo "<ok/>";
    }
```

BuzzWatch, the sample application for this chapter comes in four different versions packaged in four different archives. The file references include the file name and the version.

The first instruction is as usual to open and create the database if necessary. The next ones are to handle the XML document that has been received. BuzzWatch could have read the XML document directly with SimpleXML. Instead, it has been decided that some tests need to be done to check that the document is conformant to what is expected. BuzzWatch could have done these tests in PHP. However, because the document is in XML, a more concise option is to rely on an XML schema language.

There are a number of XML schema languages. The dominant one is W3C XML Schema language, a language that is undeniably very complex. The support of W3C XML Schema by the libxml2 library on which PHP5 relies for its XML parsing is so limited that BuzzWatch prefers to use RELAX NG, a XML schema language outsider that is both simpler and more powerful than W3C XML Schema. RELAX NG is an ISO standard and you can find more information on RELAX NG at `http://relaxng.org`.

The RELAX NG schema that is used to validate the document is:

watch.rng – v1.0

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <ref name="watch"/>
  </start>
  <define name="watch">
    <element name="watch">
      <interleave>
        <ref name="symbol"/>
        <ref name="tag"/>
        <ref name="description"/>
        <ref name="title"/>
      </interleave>
    </element>
  </define>
  <define name="symbol">
    <element name="symbol">
      <data type="NCName">
        <param name="maxLength">5</param>
      </data>
    </element>
  </define>
  <define name="tag">
    <element name="tag">
      <data type="NCName">
        <param name="maxLength">16</param>
      </data>
    </element>
  </define>
  <define name="title">
    <element name="title">
      <data type="token">
        <param name="maxLength">128</param>
      </data>
    </element>
  </define>
  <define name="description">
    <element name="description">
      <data type="token"/>
    </element>
  </define>
</grammar>
```

Writing schemas in XML is verbose and RELAX NG has an equivalent compact syntax that is plain text. James Clark (one of the authors of RELAX NG) has published trang, a tool to convert from one syntax into the other that can also generate RELAX NG schemas from an example of a document and translate a RELAX NG schema into W3C XML Schema. The same schema written with the compact syntax is:

watch.rnc – v1.0

```
start = watch
watch = element watch { symbol & tag & description & title }
symbol = element symbol { xsd:NCName {maxLength = "5"}}
tag = element tag { xsd:NCName {maxLength = "16"}}
title = element title { xsd:token {maxLength = "128"}}
description = element description { xsd:token }
```

Both flavors are strictly equivalent. They say that the root element must be `watch` and that the `watch` element is composed of `symbol`, `tag`, `description`, and `title` sub-elements in any order (the fact that these sub-elements can be in any order is expressed by the interleave in the XML syntax and the symbol & in the compact syntax). Some constraints are imposed to these sub-elements: `symbol` and `tag` have a type `xsd:NCName`, meaning that they would be valid XML element or attribute names without colons. The maximum length of `symbol`, `tag` and `title` are also set to be respectively 5, 16, and 128.

You can find more information about RELAX NG at `http://relaxng.org` *and in the online book* RELAX NG *by Eric van der Vlist, available at* `http://books.xmlschemata.org/relaxng/`.

To check that a document is conformant to this schema in PHP5, you load the document into a DOM (this is the instruction `$dom->loadXML($HTTP_RAW_POST_DATA);`) and use the `$dom->relaxNGValidate()` method to perform the validity checks. If the document is valid for this schema, you can load it into a `SimpleXML` object using its `simplexml_import_dom()` method to do the update. The schema validation is enough to be sure that no SQL injection attack can be done using the `symbol` and `tag` elements: they wouldn't meet the requirements for their `xsd:NCName` datatypes if they included the apostrophe needed for a SQL injection. However, the schema doesn't check that there are no apostrophes in the `title` or `description` elements. Furthermore, it wouldn't make sense to forbid apostrophes in these elements; valid titles and descriptions may include this character. To defend BuzzWatch against SQL injection, you thus have to use the `sqlite_escape_string()` method again. BuzzWatch does so in a loop that adds an attribute with the escaped value to each child element in the document. Further treatments can use this attribute instead of the raw value to be immune to SQL injection.

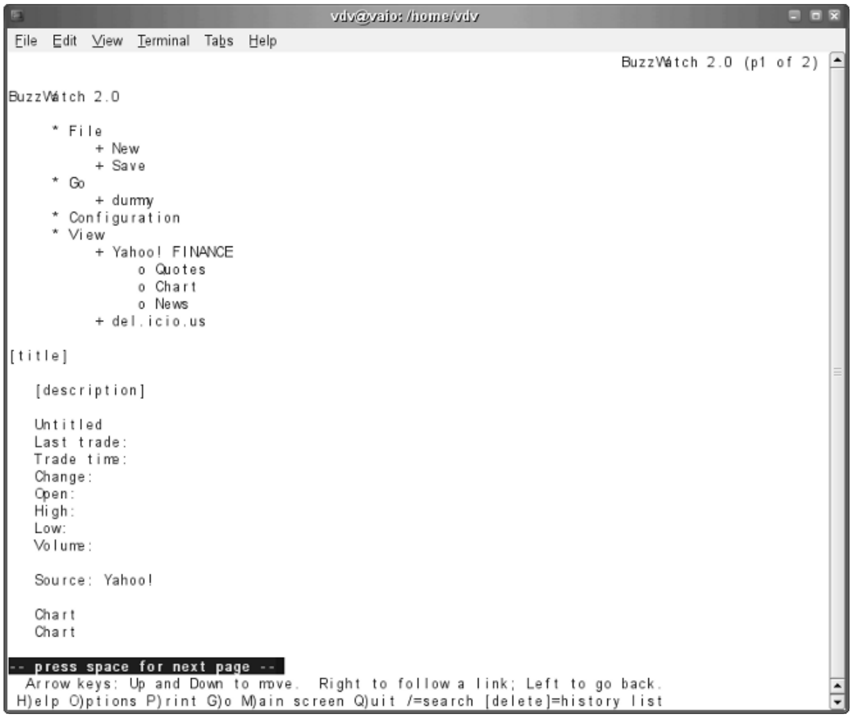
The last difficulty is that BuzzWatch is using the same HTTP `POST` method to create a new watch and to update an existing one. In Chapter 11, you learn that purists consider that these two operations should rather use different methods (`POST` for creating new resources and `PUT` for updating existing ones). BuzzWatch doesn't follow this rule, and the PHP script needs to check whether you're doing an update or an insertion. To do so, the script performs a select query to check if the watch already exists in the database. If that's the case, a SQL update statement is performed. If not, a SQL insert statement is chosen.

Making BuzzWatch a Better Web Citizen

You now have a pretty good idea of what a Web 2.0 application looks like. You learned how scripting interacts with both the user of a page and the web server hosting that page to change the page that is being displayed, and how it can aggregate information from various sources and let users contribute to the system. The version of BuzzWatch presented in the last section does all that, and, even though it has been kept as simple as possible for this first chapter, it works pretty efficiently and is rather fine looking. There is, however, one big criticism that can be leveled at this version, a charge that applies to quite a few Web 2.0 applications that you see in the real world.

The criticism is that, even though BuzzWatch is run in a browser and uses HTTP in a rather sensible way, it acts more like one of those client/server applications from the 1990s than like a good web citizen. One of the foundations of the World Wide Web is the notion of *hypertext*, which itself relies on URIs or URLs. This version of BuzzWatch doesn't expose proper hypertext documents or URLs for the objects that are being manipulated. Is this a real issue, and can you ignore it? Yes, this is a serious issue and you should not ignore it. Your users are used to keeping their favorite pages in bookmarks and sharing these bookmarks either using a bookmark sharing system such as del.icio.us or by just copying and pasting them in an e-mail or instant messaging system. If they like BuzzWatch, they'll want to do so with the watches that they use, too, and the current version of BuzzWatch, with its single URL that identifies only the application doesn't let them do so. Instead of being able to say "Have a look at `http://web2.0thebook.org/buzzwatch/wj-a`," they have to explain: "Open the BuzzWatch application, click the Go menu and choose Wiley (wj-a)...." A side-effect of not using URLs is that they can't use the Back and Forward buttons of their browsers with BuzzWatch. It took several years for Web 1.0 applications to provide a decent support of the Back and Forward buttons, and many Web 2.0 applications are still struggling with this issue!

Another consequence of this design is that BuzzWatch is not accessible. Web accessibility is an important concept that means that you should try to make your applications accessible to as wide an audience as possible. A good way to check whether your application is accessible is to open it with a text browser such as Lynx, and BuzzWatch doesn't perform well at all if you try that. Figure 1-4 shows the dismaying result.



```
                                                    BuzzWatch 2.0 (p1 of 2)

BuzzWatch 2.0

     * File
          + New
          + Save
     * Go
          + dummy
     * Configuration
     * View
          + Yahoo! FINANCE
               o Quotes
               o Chart
               o News
          + del.icio.us

[title]

   [description]

   Untitled
   Last trade:
   Trade time:
   Change:
   Open:
   High:
   Low:
   Volume:

   Source: Yahoo!

   Chart
   Chart

-- press space for next page --
  Arrow keys: Up and Down to move.  Right to follow a link; Left to go back.
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Figure 1-4

Why is it important that your web application displays well in a text web browser since almost nobody uses one any longer? It's important, because voice systems used by blind people see your web page and transcribe it orally in a way that is very similar to what a text browser displays. If you're not convinced by this argument, you may find it more convincing to know that the search engine web crawlers see your document approximately the same way as text browsers. Wouldn't you like it if the watch page for Microsoft was in the top search results for Microsoft on major search engines? You can be sure that this won't be the case for the current version of BuzzWatch.

> *The issue of serving web pages and web applications that can be used by a wide audience, including vision-impaired people using voice devices or Braille readers, users browsing the Web from small devices, web crawlers, and even the small number of geeks browsing the Web with Lynx is known as* web accessibility*. Chapter 4 has more on web accessibility.*

How can you update BuzzWatch so that it behaves like a good web citizen and keeps the look and feel and reactivity that makes it a real Web 2.0 application? Fortunately, the answer to this question is, at least in its principle, quite easy. Since you want one URL per watch, you should accept that users reload their pages to change URLs when they switch between watches. And since you want BuzzWatch to degrade nicely and display significant information even in browsers that do not support JavaScript, BuzzWatch pages should come populated with their content when they load, even if partial refreshes are operational later on for JavaScript-enabled browsers. In other words, BuzzWatch should be a Web 1.0 application with Web 2.0 features for users that have JavaScript available.

If you have installed BuzzWatch to try these examples by yourself, it's time to install version 2.0.

To implement these changes, you'll have to replace the static HTML document that the first version of BuzzWatch served with a PHP script that generates pages where the information is already pre-populated in the different panels. The classical way of doing so in PHP is to embed PHP statements in an HTML document. However, since BuzzWatch has already implemented methods in JavaScript that create this content client side by manipulating the DOM, you may prefer to port the same methods in PHP. In that case, your script loads the same HTML document that was sent to the browser up to now in a DOM, update this DOM to add the information that is needed in the panels, and send the serialization of this DOM to the browser. Client side, the scripts need to be updated so that they do not immediately refresh the panels' content (that would be a waste of bandwidth), but start a timeout instead and refresh the panels after this timeout.

The `index.php` script does quite a few tasks that are similar to those done by the scripts sending the XML for the different panels, and some refactoring is welcome to define these common actions as functions. The body of the script is:

```
// Open the database and fetch the current
// watch if needed.
$db=openDb();
$query = queryOneWatch($db);
$watchRow = $query->fetch(SQLITE_ASSOC);
// Create a DOM and load the document
$document = new DOMDocument();
$document->validateOnParse = TRUE;
$document->load("template.html");
// Populate the menu bar and the panels
populateMenuGo($db);
populateForms($watchRow);
populateQuotes($watchRow);
```

```
  populateChart($watchRow);
  populateFinancialNews($watchRow);
  populateDelicious($watchRow);
  // Send the result
  header("Cache-Control: max-age=60");
  header("Content-type: text/html");
  print $document->saveXML();
```

The validation of the document is necessary for libxml2, the XML parser on which PHP5 relies, if you want to be able to get elements by their identifiers.

Like their JavaScript equivalents, the functions that populate the panels receive their data as XML. They could get this XML accessing the web server locally through HTTP. However, you can save the server the extra load of a full HTTP request for each panel when the page is created by exposing the XML cached document through PHP functions. Following this principle, the function that populates the quotes panel in `index.php` is:

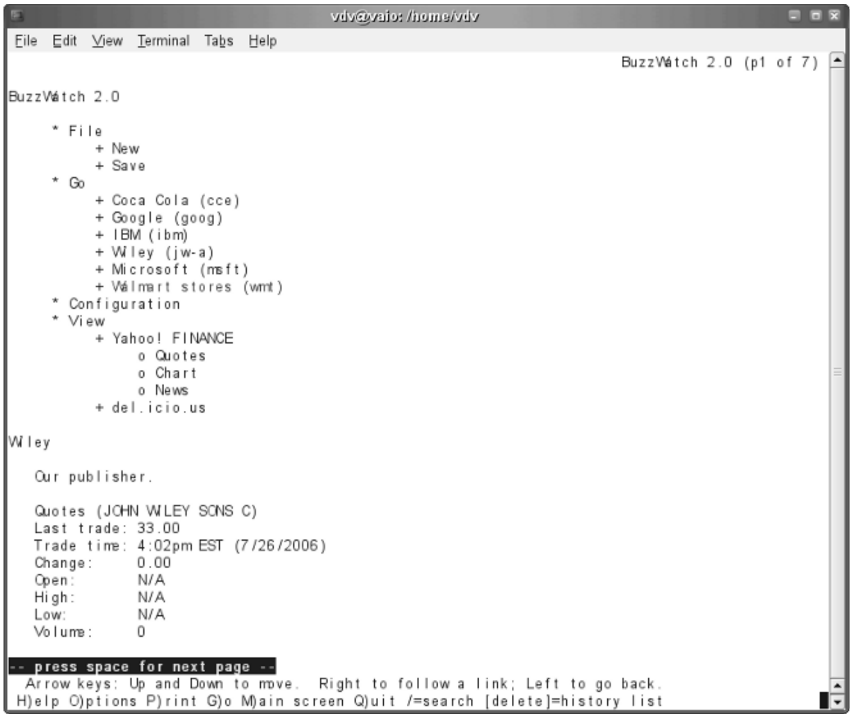
```
  function populateQuotes($watchRow) {
    global $document;
    if (!$watchRow) {
      return;
    }
    $quote = simplexml_load_string(
      get_cached_data(
        getUrlQuotes($watchRow['symbol']),
        get_quotes_as_xml,
        YAHOOFINANCE_QUOTES_LIFETIME
      )
     );
    setValue('yahoofinance.quotes.title', "Quotes ({$quote->name})");
    setValue('yahoofinance.quotes.last_price', $quote->lastTrade->price);
    setValue('yahoofinance.quotes.last_time',
      "{$quote->lastTrade->time} EST ({$quote->lastTrade->date})" );
    setValue('yahoofinance.quotes.change', $quote->change);
    setValue('yahoofinance.quotes.open', $quote->open);
    setValue('yahoofinance.quotes.high', $quote->high);
    setValue('yahoofinance.quotes.low', $quote->low);
    setValue('yahoofinance.quotes.volume', $quote->volume);
  }
```

This function is very similar to the JavaScript method that refreshes the panel client side in `panel.js`:

```
    YAHOO.buzzWatch.panels["yahoofinance.quotes"].callback["success"] = function(o) {
      if(o.responseText !== undefined){
        this.isEmpty = false;
        this.setTimeout(getMaxAge(o));
        var xotree = new XML.ObjTree();
        var tree = xotree.parseDOM( o.responseXML.documentElement );
        if (tree.quote.name != undefined) {
          setValue('yahoofinance.quotes.title', "Quotes ("+tree.quote.name+")");
          setValue('yahoofinance.quotes.last_price', tree.quote.lastTrade.price);
          setValue('yahoofinance.quotes.last_time',
            tree.quote.lastTrade.time + " EST (" + tree.quote.lastTrade.date +")" );
          setValue('yahoofinance.quotes.change', tree.quote.change);
```

```
            setValue('yahoofinance.quotes.open', tree.quote.open);
            setValue('yahoofinance.quotes.high', tree.quote.high);
            setValue('yahoofinance.quotes.low', tree.quote.low);
            setValue('yahoofinance.quotes.volume', tree.quote.volume);
          }
        }
      }
```

In addition to the differences of syntax between PHP and JavaScript, there are also differences in the libraries used to manipulate XML as objects. One difference is that in PHP, SimpleXML merges the root element and the document itself (you write `$quote->high`); this is not the case with JKL.ParseXML (you write `tree.quote.high`). You would find more differences in more complex functions such as the one that displays RSS channels, but the PHP and JavaScript versions are still very close. After this update, BuzzWatch is more accessible and you can see information in the pages if you try again to access `http://localhost/buzzwatch/?name=jw-a` using Lynx, as you can see in Figure 1-5.



```
                                                          BuzzWatch 2.0 (p1 of 7)

BuzzWatch 2.0

     * File
          + New
          + Save
     * Go
          + Coca Cola (cce)
          + Google (goog)
          + IBM (ibm)
          + Wiley (jw-a)
          + Microsoft (msft)
          + Walmart stores (wmt)
     * Configuration
     * View
          + Yahoo! FINANCE
               o Quotes
               o Chart
               o News
          + del.icio.us

Wiley

   Our publisher.

   Quotes (JOHN WILEY SONS C)
   Last trade: 33.00
   Trade time: 4:02pm EST (7/26/2006)
   Change:     0.00
   Open:       N/A
   High:       N/A
   Low:        N/A
   Volume:     0

-- press space for next page --
  Arrow keys: Up and Down to move.  Right to follow a link; Left to go back.
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Figure 1-5

Making BuzzWatch More Maintainable

To make a good web citizen out of BuzzWatch, you had to develop a set of functions twice, in two different languages. The number of lines affected by this duplication is low (about 120 lines in PHP) and you may think that this isn't a big deal. On the other hand, if BuzzWatch is successful, it will probably grow and each new feature will have to be implemented both in PHP and JavaScript. Web 2.0 applications also need to be reactive to bug reports and feature requests posted by their users, and each fix will also have to be implemented twice.

To avoid that and minimize the maintenance costs, you have two options. The first of these options is to perform the operations that are duplicated between the server and the browser at only one location. That was what BuzzWatch did with its first version, where the menu bar and the panels were built only client side in JavaScript. If it doesn't work well to do this client side, the other option is to try to do these operations only server side. How can you do that without losing the benefits of Ajax? This is quite easy: instead of sending XML documents to the browser, use Ajax to send (X)HTML fragments!

To implement this strategy with BuzzWatch, you would update the PHP scripts that generate the XML for the panels and the list of watches used by the menu bar so that they send XHTML fragments instead of XML. You could do that using the `populateXXX` functions that were developed in the previous version, and the result would be quite simple to roll out. Client side, the JavaScript would be modified to copy the XHTML received by the XML HTTP requests straight into the document instead of reformatting it as was required up to now.

Would that be a good thing? The answers that you will find to this question are often distorted by quasi-religious wars between XML proponents and opponents. You will see in Chapter 8 that formats other than XML can be used by Ajax applications. A popular option, JSON, is to send the text declarations of JavaScript objects. When you receive a JSON document, your JavaScript objects are already packed in something similar to what you get after loading an XML document in a JKL.ParseXML tree. Even if it is simpler to get JavaScript objects straight away, the fact that libraries such as JKL.ParseXML exist means that this doesn't make a big difference. Compared to JSON, XML has the benefit of being totally language- and environment-agnostic: You've serialized PHP arrays with SimpleXML into XML and loaded this XML into JavaScript objects with JKL.ParseXML without any problem, and without having to see a single angle bracket. You could have done the same between Java, C#, Python, Perl, Ruby, and so on. Doing so with JSON would have required that each of these languages use JSON libraries to support the JavaScript syntax for expressing literals.

XML is not only agnostic about programming languages and environment, but also about the usage that can be done with the data. Sending the same data in (X)HTML introduces a bias toward presentation that can be considered as bad as the bias toward JavaScript that JSON represents. It does not make any difference for BuzzWatch as you've seen it up to now, but sending a list of watches as the HTML `ul` list that constitutes the Go menu represents a loss of information compared to the list of watches that BuzzWatch uses up to now. Web 2.0 is about re-purposing information and finding new usages for existing data. Good Web 2.0 applications should not only exploit existing data to present them to their users but also contribute by publishing information is a way that makes it easy to reuse. If you publish a list of watches in XML, you make it easy for other Web 2.0 applications to reuse this information. If you publish the same list in XHTML, this is still possible but less straightforward.

Because (X)HTML is presentation-oriented and hides the real structure of the document, people often present formatting XML documents in (X)HTML server side as a semantic firewall that protects your data from being stolen. This is, of course, a very Web 1.0–ish way of seeing things!

To add semantic information in XHTML, you can also use *microformats* (which are covered in Chapter 10). Microformats are a way to use (X)HTML class attributes to convey the semantic information that is lost when you transform an XML document into (X)HTML. You could define your own microformat for the XHTML that BuzzWatch would send to the browser. However, even though microformats are a very cool hack for defining formats that are ready to be presented while keeping some of the information that would be available in an equivalent XML format, their rules are so flexible that they remain more difficult to process than XML formats.

Another option is to expose the same information both as XML and as (X)HTML, and that would be possible without much redundancy by the same set of common PHP functions. However, there is a second option to avoid the duplication of code between the server and the browser: using the same language on both sides. Of course, running PHP client side wouldn't be very realistic. Server-side JavaScript was the language of choice in the Netscape web servers, and it has been a popular choice for people using these servers but has almost faded out. You may consider that unfortunate, since JavaScript isn't worse than other script languages around and using it both client and server side would be very coherent; however, you won't find many tools and frameworks to develop server-side JavaScript applications any longer.

If you want to use the same language, which is neither JavaScript nor PHP, to transform XML into (X)HTML client and server side, you need to find a higher-level language supported in both environments. XSLT is such a language. You learn more about XSLT in Chapter 5. XSLT is a programming language targeted to defining transformations between XML documents and transforming XML into (X)HTML is one of its most common usages. XSLT is available both server side and client side where its support by recent versions of Internet Explorer and Firefox is excellent. XSLT is interoperable between environments, and it is possible to write a transformation that transforms both an XML consolidation of the various information into a full (X)HTML document server side and the individual XML documents into (X)HTML fragments client side.

If you have installed BuzzWatch on your station to run these examples, it's time to install version 3.0. Server side, the script `index.php` packs all the information needed to build the whole page into a single document with a root document element. To do so, it creates a DOM, inserts the root element, and appends the different information:

```
$document = new DOMDocument();
$root = $document->createElement("root");
$document->appendChild($root);
appendDocument(
  $document,
  getAllWatches($db)
  );
if ($watchRow) {
appendDocument(
  $document,
  getAWatch($db, $_GET['name'])
  );
appendDocument(
  $document,
  get_cached_data(
    getUrlQuotes($watchRow['symbol']),
    get_quotes_as_xml,
    YAHOOFINANCE_QUOTES_LIFETIME
    )
  );
appendDocument(
  $document,
  get_cached_data(
    getUrlFinancialNews($watchRow['symbol']),
        defaultCacheGet,
        YAHOOFINANCE_NEWS_LIFETIME
        )
      );
    appendDocument(
      $document,
      get_cached_data(
        getUrlDelicious($watchRow['tag']),
        defaultCacheGet,
        DELICIOUS_LIFETIME
        )
      );
    }
```

The function `appendDocument()` parses an XML document received as a string, imports its root element into the target document, and appends the result to the document

```
    function appendDocument($document, $xml){
      $fragment = new DOMDocument();
      $fragment->loadXML($xml);
      $importedFragment = $document->importNode($fragment->documentElement, true);
      $document->documentElement->appendChild($importedFragment);
    }
```

The complete document has the following structure:

```
    <?xml version="1.0"?>
    <root>
      <watches>
    .
    . The list of watches needed to create the Go menu is included here
    .
      </watches>
      <watch>
    .
    . The definition of the current watch is included here
    .
      </watch>
      <quote>
    .
    . The current stock quote is included here
    .
      </quote>
      <rss version="2.0">
    .
    . The Yahoo finance news channel is included here
    .
      </rss>
      <rdf:RDF>
    .
    . The del.icio.us channel is included here
    .
      </rdf:RDF>
    </root>
```

It is transformed with the XSLT transformation `format.xsl` and the result is sent to the browser:

```
$xsltSource = new DOMDocument();
$xsltProc = new XSLTProcessor();
$xsltSource->load('format.xsl');
$xsltProc->importStyleSheet($xsltSource);
header("Cache-Control: max-age=60");
header("Content-type: text/html");
print $xsltProc->transformToXML($document);
```

To manipulate XSLT client side in a consistent way between different browsers, you need a JavaScript API such as Sarissa. The principle is the same, except that you can load the transformation at load time and keep the `XSLTProcessor` object to reuse it several times. You can load the transformation asynchronously like any other XML resource. The request is initiated by the following instruction in `controller.js`:

```
YAHOO.util.Connect.asyncRequest('GET', "format.xsl", this.xsltCallback);
```

And the result is kept to be used when needed:

```
BuzzWatchController.prototype.handleXSLTSuccess = function(o) {
  if(o.responseText !== undefined){
    this.xsltDom = o.responseXML;
  }
}
```

XSLT processors are created and initialized with the XSLT transformation by the following statements:

```
    var processor = new XSLTProcessor();
    processor.importStylesheet(this.xsltDom);
```

The different callbacks that receive XML to update a panel can then be replaced by a generic one that uses such a XSLT processor in `panel.js`:

```
BuzzWatchPanel.prototype.handleSuccess = function(o) {
    if(o.responseText !== undefined){
      this.isEmpty = false;
      this.setTimeout(getMaxAge(o));
      var processor = YAHOO.buzzWatch.controller.getXSLTProcessor();
      var result = processor.transformToDocument(o.responseXML);
      YAHOO.buzzWatch.controller.returnXSLTProcessor(processor);
      var o = document.getElementById(this.name);
      var n = document.importNode(result.documentElement, true);
      o.parentNode.replaceChild(n, o);
      this.panelConfig = this.panel.cfg.getConfig();
      this.panel.init(this.name, this.panelConfig);
      this.panel.render();
    }
  }
```

The result of the transformation is imported into the window's document and replaces the corresponding division. After this operation, the YUI panel needs to be reinitialized and rendered. The XSLT transformation itself (`format.xsl`) is too verbose to be printed here, and you'd need the introduction that you'll find in Chapter 5 to understand it. To give you a first glimpse of it, here is the template (a template is a rule) that replaces the value of the `src` attribute of the `img` element with the chart:

```
<xsl:template match="x:img[@id='yahoofinance.chart.img']/@src" mode="html">
  <xsl:attribute name="src">
    <xsl:if test="$watch/symbol">
      <xsl:value-of select="concat('yahoo_chart.php?tag=', $watch/symbol)"/>
    </xsl:if>da
  </xsl:attribute>
</xsl:template>
```

The conditions that trigger the template are defined in its `match` attribute. This one will apply to the `src` attributes (trailing `@src`) whose parent element is `x:img` (in that case, `img` elements from the XHTML namespace) and whose `id` attribute is equal to `yahoofinance.chart.img`. The template replaces such an attribute by a new attribute (`xsl:attribute` statement) with the same name. The content of this new attribute is the concatenation of `yahoo_chart.php?tag=` and the symbol value which is found as the `symbol` element of the variable `$watch` (`xsl:value-of` statement) only if the symbol exists (`xsl:if` statement).

Applying the Final Touch

A lot of features have to be added and a lot of improvements to be performed before BuzzWatch can compete with the most popular Web 2.0 applications. However, its technical basis is now relatively stable and the necessary improvements are out of the scope of this chapter. One point still remains weak, and you'll have a chance to improve on it before moving on to Chapter 2.

You may have noticed that there are six different PHP scripts: `index.php` serves the pages; `watch.php` lists watches, provides the definition of a single watch, and manages saving watches; and one script per external source: `yahoo_quotes.php`, `yahoo_chart.php`, `yahoo_finance_news.php` and `delicious.php`. There is nothing wrong with splitting BuzzWatch server-side operations into six and only six scripts, but this is an implementation decision that may change over time and that's not necessarily something to expose to your users.

If you don't do anything to avoid that, your users will have to use URLs with query strings such as `http://web2.0thebook.org/buzzwatch/index.php?name=wj-a` or `http://web2.0thebook.org/delicious.php?tag=google` to identify the resources handled by BuzzWatch. Even if a lot of Web applications expose URLs such as these ones, this is considered a bad practice for a number of reasons:

- ❑ Exposing the technology used server side through file extensions (here .php) is a bad idea: if you decide to change this technology for example to move from PHP to Python or Java, you'll have to change your URLs and everyone knows that cool URIs don't change. Furthermore, such information is used by hackers to identify target sites on which they can test known security flaws. Using a search engine such as Google, they can easily get a list of sites running PHP on which they can try to exploit the latest weaknesses discovered in PHP. Of course, hiding this information isn't an adequate response if they've decided to hack your site, but exposing it contributes to make your site one of these low-hanging fruits that they prefer.

- ❑ Even if you keep the same technology server side, you may decide to change the distribution of the functions in the different scripts. For example, you may decide that you prefer to have a single script for all the data sources with an additional parameter to specify the source, or you may decide that you want three different scripts instead of one to get a watch, get a list of watches, and save a watch. Again, these implementation choices shouldn't have an impact on your users.
- ❑ URLs are addresses for Web resources and using a single address with query parameters is like having a care-of address in the real world; that method works, but it's better to provide individual addresses to each resource.

To do so, you need to define the URL space that you'll be using for BuzzWatch (as covered in Chapter 7) and to implement it server side (as covered in Chapter 16). Designing a URL space includes, like any design, a good deal of subjectivity. One rule of the thumb is to make a list of the different objects that have their identifiers. BuzzWatch manipulates three different objects: watches identified by their names (the current version uses stock symbols as names but that's only a rather arbitrary implementation decision), companies identified by their stock symbols, and tags identified by tag names. A typical way of defining a URL space for these three classes is to give them their own URL roots, such as: `http://web2.0thebook.org/buzzwatch/watch/`, `http://web2.0thebook.org/buzzwatch/company/`, and `http://web2.0thebook.org/buzzwatch/tag/`. These roots are then used to define a URL per object, such as `http://web2.0thebook.org/buzzwatch/watch/wmt/`, `http://web2.0thebook.org/buzzwatch/company/msft/`, or `http://web2.0thebook.org/buzzwatch/tag/google/`. The next step is to use these URLs to define URLs per type of information. For a tag, we'd have `http://web2.0thebook.org/buzzwatch/tag/google/delicious` for the del.icio.us and that would leave the option of adding new resources for the same tag (for example, Technorati or Flickr search results). For a company, you already have Yahoo! financial news, quotes, and charts and we could add more of them. For a watch, you have the watch itself but need to differentiate the (X)HTML version from the XML description and, eventually, the concatenated document with all the information for a watch.

Now that you have designed your URL space, how do you implement it? The good news is that if you are using a Web server that supports URL rewriting, you won't have to change a single line in your PHP scripts. With Apache, for example, you would implement a URL space similar to what we've described above with the following directives in a `.htaccess` file:

.htaccess

```
RewriteEngine on
RewriteBase   /buzzwatch/

RewriteRule ^$  watch/welcome+page/ [R]
RewriteRule ^watch/welcome+page/$ index.php [L]
RewriteRule ^watch/([^/.]*)/$ index.php?name=$1 [L]
RewriteRule ^watch/list.xml$ watch.php [L]
RewriteRule ^watch/([^/.]*)/index.xml$ index.php?name=$1&format=xml [L]
RewriteRule ^watch/([^/.]*)/watch.xml$ watch.php?name=$1 [L]
RewriteRule ^tag/([^/.]*)/delicious.xml$ delicious.php?tag=$1 [L]
RewriteRule ^company/([^/.]*)/yahoo/finance/news.xml$ yahoo_finance_news.php?tag=$1
[L]
RewriteRule ^company/([^/.]*)/yahoo/finance/quotes.xml$ yahoo_quotes.php?tag=$1 [L]
RewriteRule ^company/([^/.]*)/yahoo/finance/chart.png$ yahoo_chart.php?tag=$1 [L]

RewriteCond %{THE_REQUEST}  ^.*.php\??.*$
RewriteRule ^.*.php$    nophp [G]
```

The line break between `yahoo_finance_news.php?tag=$1` *and* `[L]` *has been added to fit the text in the page and does not exist in the* `.htaccess` *file.*

Each `RewriteRule` is a rule that changes URLs through a regular expression. The first rule is to redirect the BuzzWatch home page to a page that has the same level in the hierarchy as the other pages (this is a hack to be able to use the same relative URIs as the other pages). The second one sets this home page. The third one is for the individual pages for the watches, the fourth for the list of watches, the fifth serves an XML document with the consolidated information that constitutes a watch, the next one is the address at which each watch can be loaded and saved, and the next ones are the different pieces of information that are aggregated. The rule with a `RewriteCond` prevents direct access to the PHP scripts so that users can use BuzzWatch only through your new URL space. It returns a `410 GONE` HTTP code that means, "Sorry, you can't access this resource any longer."

If you are running these examples on your system, these last changes are implemented in version 4.0. They include the `.htaccess` documents and the updates of all the URLs used by the application.

Conclusion

In this chapter, you learned that Web 2.0 applications can use Web 1.0 tools and infrastructure. Technically speaking, the main difference is the amount of JavaScript used to animate the pages and the scope of the modifications that these scripts apply the web pages after they're sent by the server. The sequence of exchanges between the browser and the server and their switches between JavaScript and PHP have shown you how intermingled are the treatments that are done client and server side. One of the main challenges for Web 2.0 developers is to keep all these interactions in mind. With the increasing popularity of Web 2.0 applications, a new class of tools is beginning to emerge that try to integrate all these interactions. One of the best examples of such frameworks is the very popular Ruby on Rails, but other options exist, such as using XForms with a client/server implementation such as Orbeon PresentationServer, as mentioned in Chapter 5.