Writing Web Applications with Web Services and Ajax

Mike Diehl

Abstract

An Ajax primer with Perl and PostgreSQL.

If you've done any Web development at all recently, you've no doubt heard the buzz going on about Web Services and Ajax. The industry hype is so prevalent that you'd almost think people were talking about the next Microsoft operating system. Fortunately, they're not. Web Services and Ajax are two Web technologies that allow developers to create more interesting Web applications as well make the development easier and less error-prone.

Now that I've added to the hype, let me take some time to outline what we mean when we say “Web Services and Ajax”.

A Web Service is a program that is accessible over the Internet and provides a specific service, such as searching a library's collection or getting bid history from eBay. We're not talking about a full-fledged application, but rather a Web-based Application Programming Interface (API) that can be called over the Internet by a given program to perform a needed function. Often, the results of the call to a given Web Service are returned in XML format, which the calling program can manipulate easily.

When people discuss Web Services, they often mention things like JSON, REST, SOAP or XML-RPC. These are simply a few of the protocols available for calling a Web Service. Being familiar with these protocols lets you make use of some of the really powerful Web Services being provided by the likes of Amazon, Google and eBay. However, for my personal Web development, I've found these protocols to be a bit heavy.

Ajax is a mechanism that allows a Web page to make calls back to the server without refreshing the page or using hidden frames. For example, if a user changes the value of a form field, the Web page could tell the server to make a change to a database, without having to refresh the Web page, as would be needed in the case of a standard CGI script. From the user's perspective, the update just happens.

In this article, I outline a set of very primitive Web Services that perform specific functions on a database. The calls to the Web Services will be done via Ajax. Essentially, we're going to build a simple contact management program that stores a person's first name, last name and phone number. We'll be able to move up and down through the database, make additions and corrections and delete records. The neat thing is that once the page is initially loaded, we won't have to refresh it again, even when we make changes.

Before we can get started though, we need to have a table in a database in which to store the information. I happen to use PostgreSQL as my preferred DBMS. For our simple application, we need only one table (Listing 1). Garrick, small font in Listing 1.
Listing 1. Preparing a PostgreSQL Sequence and Table for the Project

```sql
create sequence contacts_id_seq;
create table contacts ( 
    id integer default nextval('contacts_id_seq') not null, 
    first varchar(20), 
    last varchar(20), 
    phone varchar(20) 
);
```

The snippet of SQL in Listing 1 creates a sequence and a table. The table structure is pretty straightforward for our simple application. The only thing worth mentioning is the id field. By default, when records are inserted into our contacts table, the value of the id field is set to the next number in the contacts_id_seq sequence. The end result is that each of our contact records has a unique integer number that can be used to locate it.

Now that we have the database table defined, we can start to flesh out the actual application. Listing 2 shows the HTML for our simple application, and Figure 1 shows what the application looks like in a Web browser.

Listing 2. The Basic HTML for the Application

```html
<html>
<head>
    <title>Contact Application</title>
    <script src=http://contacts.js></script>
</head>
<body>
<form method=POST name=main>
    <input type=button name=new value="New" 
        onclick="insert_record();"> 
    <input type=button name=delete value="Delete" 
        onclick="delete_record(main.id.value);"> 
    <p> 
        Record Number: <input id=id name=id> 
    </p> 
    First: <input id=first name=first 
        onChange="update_record(main.id.value, 'first', main.first.value);"> 
    <br>
    Last: <input id=last name=last 
        onChange="update_record(main.id.value, 'last', main.last.value);"> 
    <br>
    Phone: <input id=phone name=phone 
        onChange="update_record(main.id.value, 'phone', main.phone.value);"> 
    <br>
    <input type=button name=previous value="Previous" 
        onClick="select_record(main.id.value-1);"> 
    <input type=button name=next value="Next" 
        onClick="select_record(Number(main.id.value) + 1);"> 
</form>
</body>
</html>
```
As you can see, our simple application is just that, simple. I've stripped it down to the bare necessities to make our discussion easier.

Figure 1 shows how our application allows us to insert a new contact record or delete the current record by pressing the buttons at the top. At the bottom of the application, we can move to the previous or next record in the database. Of course, we have input fields to hold the first and last name as well as the phone number. We also have a form field to display the record id number. In a real application, I'd probably make this a hidden field, but for the sake of instruction, I've left it visible.

Referring back to Listing 1, you can see that the page is fairly straightforward. Aside from importing the contacts.js JavaScript, the first part of the page is standard boilerplate. Things get interesting when we get to the form buttons and fields.

Let's look at the “New” button:

```html
<input type=button name=new value="New"
onclick="insert_record();">
```

This button simply calls a JavaScript function called insert_record() any time a user presses the button. The Delete, Previous and Next buttons all work similarly. The magic is in the JavaScript. Let's look at the JavaScript first (Listing 3). *Garrick, small font in Listing 3.*

**Listing 3. JavaScript code handles the database actions and data processing.**

```javascript
var req;

function insert_record () {
    send_transaction("/cgi-bin/insert.pl");
    return 1;
}

function select_record (i) {
    send_transaction("/cgi-bin/select.pl?id=" + i);
    return 1;
}
```
function delete_record (i) {
    send_transaction("/cgi-bin/delete.pl?id=" + i);
    var id = document.getElementById("id");
    select_record(id);
    return;
}

function update_record (i, field, value) {
    send_transaction("/cgi-bin/update.pl?id=" + i + 
        "&field=" + field + 
        "&value=" + value);
    return 1;
}

function send_transaction (i) {
    if (window.XMLHttpRequest) {
        req = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        req = new ActiveXObject("Microsoft.XMLHTTP");
    }
    if (req) {
        req.onreadystatechange = process_results;
        req.open("GET", i, true);
        req.send(null);
    }
}

function process_results () {
    var name = "";
    var value = "";
    var fields;
    var i;
    var length;
    if (req.readyState < 4) { return 1; } // transaction
    // not done, yet
    var xml = req.responseXML;
    var result = xml.getElementsByTagName("result").item(0);
    fields = result.getElementsByTagName("field");
    length = fields.length;
    for (i=0; i<length; i++) {
        var field = fields[i];
        name = field.getAttribute("name");
        value = field.getAttribute("value");
        var form_field = document.getElementById(name);
        form_field.value = value;
    }
    return 1;
}

The insert_record() JavaScript function, which is called when a user presses the New button, is the simplest of the JavaScript functions. All insert_record() does is use the send_transaction() function to call the insert.pl Web Service. In fact, the insert_record(), delete_record(), select_record() and update_record() functions are all wrappers for send_transaction().
The send_transaction() function is where the Ajax comes into our application. This function takes the URL of the service that needs to be called as well as any parameters that need to be passed to the service via HTTP's GET method. Then, the function creates an object that allows the service to be called. We have to jump through a small hoop, because Microsoft chose to call this object ActiveXObject while almost every other browser calls it XMLHttpRequest. Once the object is created, by whatever name, we tell the object to call our Web Service and then call our process_results() function when the call has returned its results. This is done in the line that assigns the function name to the object's onreadystatechange property.

Well, I lied a little bit. It turns out that the browser will call our process_results() function up to four times at various stages during the service request. Each time the function is called, the value of the readyState property is changed to reflect what phase of the transaction is occurring. Unfortunately, there doesn't seem to be much agreement on when the function is called. The only thing that all browsers seem to agree on is that when the transaction is complete, the readyState property is set to 4. Checking for this value is the first thing our process_results() function does. If the transaction isn't complete, we simply return quietly.

Once the transaction is complete, we can recover the resulting XML from the request object's responseXML property. Once we have the XML, we loop over each field element, making a note of both the field name and value. Then we find the corresponding field in the HTML document and assign the new value to it. So by sending the appropriate XML, the Web Services can arrange for any, or all, of the Web form fields to be updated.

If you think the JavaScript was easy to follow, wait until you see the Perl scripts that implement the Web Services; they're even easier to understand and debug. The insert.pl program is shown in Listing 4.

Listing 4. The server-side Perl script handles the database insert action.

```perl
#!/usr/bin/perl
use DBI;
$dbh = DBI->connect("dbi:Pg:dbname=database", "postgres", "password");
$dbh->do("insert into contacts (first,last,phone) values (NULL,NULL,NULL)");
$sth = $dbh->prepare("select last_value from contacts_id_seq");
$sth->execute();
($index) = $sth->fetchrow_array();
print "Content-type: text/xml\n\n\n";
print "<result>
";
print "<field name="id" value="$index"></field>
";
print "</result>\n";
```

All this program does is connect to a database, insert an empty record into the contacts table, retrieve the id of the newly created record and return the results in a block of XML with a text/xml MIME type. The resulting XML resembles that shown in Listing 5.

Listing 5. Resulting XML.
<result>
 <field name="id" value="25"></field>
</result>

The select.pl, delete.pl and update.pl services are very similar, as shown in Listings 6, 7 and 8, respectively.

**Listing 6. The Perl Script for Selecting Data**

```perl
#!/usr/bin/perl
use CGI;
use DBI;

$dbh = DBI->connect("dbi:Pg:dbname=database",
                   "postgres", "password");

print "Content-type: text/xml\n\n\n";
print "<result>\n";
$cgi = new CGI;
$id = $cgi->param("id");

$sth = $dbh->prepare("select * from contacts where id=$id");
$sth->execute();
$a = $sth->fetchrow_hashref();
foreach $key (keys %$a) {
    print "<field name="$key" value="$a->{$key}" /></field>\n";
}
print "</result>\n";
```

The select.pl service shown in Listing 6 takes a single parameter—the id number of the record to be retrieved. The result is an XML file containing all the fields in the record and the appropriate values. This allows us to call the function with a record id and retrieve all the fields of that record for later manipulation.

**Listing 7. The Perl Script for Deleting a Record**

```perl
#!/usr/bin/perl
use CGI;
use DBI;

$dbh = DBI->connect("dbi:Pg:dbname=database",
                     "postgres", "password");

$cgi = new CGI;
$id = $cgi->param("id");

$dbh->do("delete from contacts where id=$id");
$sth = $dbh->prepare("select max(id) from contacts where id<$id");
$sth->execute();
($index) = $sth->fetchrow_array();

print "Content-type: text/xml\n\n\n";
```

The select.pl service shown in Listing 6 takes a single parameter—the id number of the record to be retrieved. The result is an XML file containing all the fields in the record and the appropriate values. This allows us to call the function with a record id and retrieve all the fields of that record for later manipulation.
The delete.pl service shown in Listing 7 takes a record id and deletes the record with that id. Then, the program finds the next lowest record number and returns that record id.

Listing 8. The Perl Script for Updating a Record

```
#!/usr/bin/perl
use CGI;
use DBI;

$dbh = DBI->connect("dbi:Pg:dbname=database",
    "postgres", "password");

$cg = new CGI;
$id = $cg->param("id");
$field = $cg->param("field");
$value = $cg->param("value");

$dbh->do("update contacts set $field='\'$value' where id=$id");
```

Finally, the update.pl service shown in Listing 8 takes a record id, a field name and a new value as parameters. Then, the program updates the given field of the selected record with the new value. The new field value is then returned via XML.

Granted, our little application is fairly trivial, but it does perform all of the basic functions that need to be performed on a database: insert, delete, update and search. More important, no single element of this application is difficult to write, debug or understand. In fact, with a few improvements that I outline next, the Web Service scripts and much of the JavaScript can be reused for other parts of a larger application or even many different applications. The Web Services simply become bricks that are glued together with JavaScript to build applications, and this is what makes using Web Services such an elegant method of Web development.

From the user's perspective, using Ajax to perform the database functions is a major win. As mentioned before, once the application is loaded, users never have to incur the cost of re-downloading it and having their browsers re-render it. On more complex pages, this can be a significant delay. Furthermore, because the results of a given operation are returned in small snippets of XML, the bandwidth requirements are minimal. It's arguable that not only would users perceive this type of application as faster, but it also would put lower demands on the server and network infrastructure that provide the application.

But, how hard would it be to add a new field, perhaps an e-mail address, to our application? Well, we'd have to add an appropriate field to our database table scheme. Then, we'd have to add the field, with the same name, to our HTML document. We could use the other form fields as a template of course. And, that should just about do it.
So, how could we improve our code? First, we'd need to take care of some glaring security issues. Our Web Services should use some form of authentication to make sure that only authorized users can perform database functions. More subtly though, the Web Services need to perform some basic validation on the parameters they receive. The delete.pl service accepts a record number in the form of id=25 as a parameter. What if someone wanted to be mean and, instead, sent id=25 or 1=1 to our service? Well, our database would be gone because 1=1 is always true, and our program would delete all records. So, we would have to take care of such issues before we could use these services in the wild.

You may have noticed that all of the fields in our database are of type varchar(20). That's not very flexible or efficient. To be truly useful, our services would need to be able to query the database to determine what data type a given field was and act appropriately. For example, chars and varchars need to be quoted, but integers and booleans do not. The service should be able to determine how to handle these situations.

Finally, by simply sending the name of the table as one of the parameters, we can build a Web Service that can modify database tables other than our contacts table. We'd be able to use the same services to update a shopping list, inventory or calendar. Generalizing our Web Services like this would make our simple contacts application easy to write as well as any other application in which we chose to use them.

So, by coupling Ajax with our own brand of Web Services, we're able to write applications that are more responsive to user input, less taxing on the server infrastructure, and much easier to write and maintain.