

# 4

## AJAX and Connection Manager

As far as web interface design techniques are concerned, AJAX is definitely the way to go. So what JavaScript library worth its salt these days wouldn't want to include a component dedicated to this extremely useful and versatile method of client/server communication?

The term AJAX has been part of the mainstream development community's vocabulary since early 2005 (with the advent of Google Mail). Although some of the key components that AJAX consists of, such as the XMLHttpRequest object, have been around for much longer (almost a decade in fact). The goal of asynchronously loading additional data after a web page has rendered is also not a new concept or requirement.

Yet AJAX reinvented existing technologies as something new and exciting, and paved the way to a better, more attractive, and interactive web (sometimes referred to loosely as web 2.0) where web applications feel much more like desktop applications.

AJAX can also perhaps be viewed as the godfather of many modern JavaScript libraries. Maybe it wasn't the sole motivating factor behind the growing plethora of available libraries, but it was certainly highly influential and orchestral in their creation and was at least partly responsible for the first wave of modern, class-based JavaScript libraries.

Like many other cornerstone web techniques developed over the years, AJAX was (and still is) implemented in entirely different ways by different browsers. I don't know if developers just finally had enough of dealing with these issues.

The very first JavaScript libraries sprang into existence as a means of abstracting away the differences in AJAX implementation between platforms, thereby allowing developers to focus on the important things in web design instead of worrying about compatibility issues.

The result in many cases is a quick and easy way for developers to cut down on the amount of code they are required to produce, and a better more interactive experience for end users and website visitors.

## **The Connection Manager—A Special Introduction**

The Connection Manager utility is by no means the smallest, most light-weight component included with the YUI, but it's certainly not the largest either, yet it packs so much functionality into just 12Kb (for the `-min` version).

Connection Manager provides a fast and reliable means of accessing server-side resources, such as PHP or ASP scripts and handling the response. A series of supporting objects manage the different stages of any asynchronous transactions, whilst providing additional functionality where necessary.

Connection is one of just a few of the utilities that are supported by a single class; this makes looking up its methods nice and straight-forward. It also doesn't have any properties at all (although the objects that it creates all have their own members which hold various pieces of information), which makes using it even easier!

This utility is what is known as a singleton utility, which means that there can only be one live instance of the utility at any one time, differing from many of the other components of the library. Don't worry though, this doesn't restrict you to only making one request; Connection will manage as many separate requests as you need.

Because this utility is a singleton, there are important considerations that advanced coders may want to take note of. Unlike some of the other library components, Connection cannot be subclassed – all of its class's members are static, meaning that they won't be picked up when using the YAHOO global `.extend()` method.

It wraps up the cross-browser creation of the `XMLHttpRequest` (XHR) object, as well as a simple to use object-based method of accessing the server response and any associated data, into a simple package which handles the request from start to finish. This requires minimal input from you, the developer, saving time as well as effort.

Another object created by Connection is the response object, which is created once the transaction has completed. The response object gives you access via its members to a rich set of data including the `id` of the transaction, the HTTP status code and status message, and either the `responseText` and `responseXML` members depending on the format of the data returned.

Like most of the other library components, Connection Manager provides a series of global custom events that can be used to hook into key moments during any transaction. We'll look at these events in more detail later on in the chapter but rest assured, there are events marking the start and completion of transactions, as well as success, failure, and abort events.

The Connection utility has been a part of the YUI since its second public release (the 0.9.0 release) and has seen considerable bug fixing and refinement since this time. This makes it one of the more reliable and better documented utilities available.

Connection is such a useful utility that it's used by several other library components in order to obtain data from remote sources. Other components that make use of Connection Manager include the AutoComplete control, DataTable, DataSource, and Tabview.

It is one of the only library components not dependant on the DOM utility. All it requires are the YAHOO global utility and the Event utility. That doesn't mean that you can't include a reference to the DOM utility, however, to make use of its excellent DOM manipulation convenience methods.

## The XMLHttpRequest Object Interface

When working with the Connection utility you'll never be required to manually create or access an XHR object directly. Instead, you talk to the utility and this works with the object for you using whichever code is appropriate for the browser in use. This means that you don't need separate methods of creating an XHR object in order to keep each browser happy.

A transaction describes the complete process of making a request to the server and receiving and processing the response. Connection Manager handles transactions from beginning to end, providing different services at different points during a request.

Transactions are initiated using the `.asyncRequest()` method which acts as a constructor for the connection object. The method takes several arguments: the first specifies the HTTP method that the transaction should use, the second specifies the URL of your server-side script while the third allows you to add a reference to a callback object.

A fourth, optional, argument can also be used to specify a `POST` message if the `HTTP` method is set to use `POST` giving you an easy means of sending data to the server as well as retrieving it. This is rarely required however, even when using the `POST` method, as we shall see later in the chapter.

It's also very easy to build in query string parameters if these are required to obtain the data that you are making the request for. It can be hard coded into a variable and then passed in as the second argument of the `Connection` constructor instead of setting the second argument manually within the constructor.

`Connection Manager` takes these arguments and uses them to set up the `XHR` object that will be used for the transaction on your behalf. Once this has been done, and `Connection` has made the request, you then need to define a new object yourself that will allow you to react to a range of responses from the server.

## **A Closer Look at the Response Object**

I briefly mentioned the response object that is created by the utility automatically once a transaction has completed, let's now take a slightly more in-depth view at this object. It will be created after any transaction, whether or not it was considered a success.

The callback functions you define to handle successful or failed transactions (which we'll examine in more detail very shortly) are automatically passed to the response object as an argument. Accessing it is extremely easy and requires no additional intervention from yourself.

If the transaction fails, this object gives you access to the `HTTP` failure code and `HTTP` status message which are received from the server. Examining these two members of the response object can highlight what happened to make the request fail, making it integral to any `Connection` implementation.

If the transaction was a success, these two members will still be populated but with a success code and status message, and additional members such as `responseText` or `responseXML` will also contain data for you to manipulate.

If you need to obtain the `HTTP` response headers sent by the server as part of the response, these can be obtained using either the `getResponseHeader` collection, or the `getAllResponseHeaders` member.

In the case of file uploads, some of these members will not be available via the response object. File uploads are the one type of transaction that do not make use of the `XHR` object at all, so the `HTTP` code and status message members cannot be used. Similarly, there will not be either a textual or `XML`-based response when uploading files.

---

## Managing the Response with a Callback Object

In order to successfully negotiate the response from the server, a literal callback object should be defined which allows you to deal quickly and easily with the information returned whether it is a success, failure, or another category of response.

Each member of this object invokes a callback function or performs some other action relevant to your implementation. These members can be one of several types including another object, a function call or even a string or integer depending on the requirements of your application.

The most common members you would use in your callback object would usually be based upon `success` and `failure` function calls to handle these basic response types, with each member calling its associated function when a particular HTTP response code is received by the response object.

It's also very easy to add an additional member to this object which allows you to include data which may be useful when processing the response from the server. In this situation, the `argument` object member can be used and can take a string, a number, an object, or an array.

Other optional members include a `customevents` handler to deal with custom, per-transaction events as opposed to the global events that are available to any and all transactions, a `scope` object used to set the scope of your handler functions, and a timeout count used to set the wait time before Connection aborts the transaction and assumes failure.

The remaining member of the callback object is the `upload` handler which is of course a special handler to deal specifically with file uploads. As I already mentioned, the response object will be missing success or failure details when dealing with file uploads, however, you can still define a callback function to be executed once the upload transaction has completed.

## Working with responseXML

In this example we're going to look at a common response type you may want to use—`responseXML`. We can build a simple news reader that reads headlines from a remote XML file and displays them on the page.

We'll also need an intermediary PHP file that will actually retrieve the XML file from the remote server and pass it back to the Connection Manager. Because of the security restrictions placed upon all browsers we can't use the XHR object to obtain the XML file directly because it resides on another domain.

This is not a problem for us however, because we can use the intermediary PHP file that we're going to have to create anyway as a proxy. We'll make requests from the browser to the proxy thereby sidestepping any security issues, and the proxy will then make requests to the external domain's server. The proxy used here in this example is a cut down version of that created by Jason Levitt that I modified specifically for this example.

In order to complete this example you'll need to use a full web server setup, with PHP installed and configured. Our proxy PHP file will also make use of the cURL library, so this will also need to be installed on your server.

The installation of cURL varies depending on the platform in use, so full instructions for installing it is beyond the scope of this book, but don't worry because there are some excellent guides available online that explain this quick and simple procedure.

Even though Connection Manager only requires the YAHOO and Event utilities to function correctly, we can make use of some of the convenient functionality provided by the DOM utility, so we will use the aggregated `yahoo-dom-event.js` instead of the individual YAHOO and Event files. We'll also need `connection-min.js` and `fonts.css` so make sure these are all present in your `yui` folder and begin with the following HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                                "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
                                charset=utf-8">
    <title>Yui Connection Manager Example</title>
    <script type="text/javascript"
            src="yui/yahoo-dom-event.js"></script>
    <script type="text/javascript"
            src="yui/connection-min.js"></script>
    <link rel="stylesheet" type="text/css"
          href="yui/assets/fonts-min.css">
    <link rel="stylesheet" type="text/css" href="responseXML.css">
  </head>
  <body>
    <div id="newsreader">
      <div class="header">Recent News</div>
```

```

<div id="newsitems"></div>
<div id="footer"><a class="link"
                href="http://news.bbc.co.uk/1/hi/
                help/rss/4498287.stm">
                Copyright: &copy; British
                Broadcasting Corporation</a></div>

<div>
</body>
</html>

```

We'll start off with this very simple page which at this stage contains just the markup for the newsreader and the references to the required library files. There's also a `<link>` to a custom stylesheet which we'll create in a little while.

## Adding the JavaScript

Directly after the final closing `</div>` tag, add the following `<script>`:

```

<script type="text/javascript">
  //create namespace object for this example
  YAHOO.namespace("yuibook.newsreader");

  //define the initConnection function
  YAHOO.yuibook.newsreader.initConnection = function() {

    //define the AJAX success handler
    var successHandler = function(o) {

      //define the arrays
      var titles = new Array();
      var descs = new Array();
      var links = new Array();

      //get a reference to the newsitems container
      var newsitems = document.getElementById("newsitems");

      //get the root of the XML doc
      var root = o.responseXML.documentElement;

      //get the elements from the doc we want
      var doctitles = root.getElementsByTagName("title");
      var docdescs = root.getElementsByTagName("description");
      var doclinks = root.getElementsByTagName("link");

      //map the collections into the arrays
      for (x = 0; x < doctitles.length; x++){
        titles[x] = doctitles[x];
        descs[x] = docdescs[x];

```

```
        links[x] = doclinks[x];
    }

    //removed the unwanted items from the arrays
    titles.reverse();
    titles.pop();
    titles.pop();
    titles.reverse();
    descsc.reverse();
    descsc.pop();
    descsc.reverse();
    links.reverse();
    links.pop();
    links.pop();
    links.reverse();

    //present the data from the arrays
    for (x = 0; x < 5; x++) {

        //create new elements
        var div = document.createElement("div");
        var p1 = document.createElement("p");
        var p2 = document.createElement("p");
        var a = document.createElement("a");

        //give classes to new elements for styling
        YAHOO.util.Dom.addClass(p1, "title");
        YAHOO.util.Dom.addClass(p2, "desc");
        YAHOO.util.Dom.addClass(a, "newslink");

        //create new text nodes and the link
        var title =
            document.createTextNode(titles[x].firstChild.nodeValue);
        var desc =
            document.createTextNode(descsc[x].firstChild.nodeValue);
        var link = links[x].firstChild.nodeValue;
        a.setAttribute("href", link);

        //add the new elements to the page
        a.appendChild(desc);
        p1.appendChild(title);
        p2.appendChild(a);
        div.appendChild(p1);
        div.appendChild(p2);
        newsitems.appendChild(div);
    }
}
```

```
    }  
  
    //define the AJAX failure handler  
    var failureHandler = function(o) {  
  
        //alert the status code and error text  
        alert(o.status + " : " + o.statusText);  
    }  
  
    //define the callback object  
    var callback = {  
        success:successHandler,  
        failure:failureHandler  
    };  
  
    //initiate the transaction  
    var transaction = YAHOO.util.Connect.asyncRequest("GET",  
                                                    "myproxy.php", callback, null);  
}  
  
//execute initConnection when DOM is ready  
YAHOO.util.Event.onDOMReady( YAHOO.yuibook.newsreader.  
initConnection);  
</script>
```

Once again, we can make use of the `.onDOMReady()` method to specify a callback function that is to be executed when the YUI detects that the DOM is ready, which is usually as soon as the page has finished loading.

The code within our master initialization function is split into distinct sections. We have our `success` and `failure` callback functions, as well as a callback object which will call either the success or failure function depending on the HTTP status code received following the request.

The failure handler code is very short; we can simply alert the HTTP status code and status message to the visitor. The callback object, held in the variable `callback`, is equally as simple, just containing references to the success and failure functions as values.

The pseudo-constructor which sets up the actual request using the `.asyncRequest()` method is just a single line of code. It's arguments specify the response method (`GET`), the name of the PHP file that will process our request (`myproxy.php`), the name of our callback object (`callback`), and a null reference.

Connection Manager is able to accept URL query string parameters that are passed to the server-side script. Any parameters are passed using the fourth argument of the `.asyncRequest()` method and as we don't need this feature in this implementation, we can simply pass in `null` instead.

Most of our program logic resides in the `successHandler()` function. The response object (`o`) is automatically passed to our callback handlers (both success and failure) and can be received simply by including it between brackets in the function declaration. Let's break down what each part of our `successHandler()` function does.

We first define three arrays; they need to be proper arrays so that some useful array methods can be called on the items we extract from the remote XML file. It does make the code bigger, but means that we can get exactly the data we need, in the format that we want. We also grab the `newsItems` container from the DOM.

The `root` variable that we declare next allows us easy access to the root of the XML document, which for reference is a news feed from the BBC presented in RSS 2.0 format. The three variables following `root` allow us to strip out all of the elements we are interested in.

These three variables will end up as collections of elements, which are similar to arrays in almost every way except that array methods, such as the ones we need to use, cannot be called on them, which is why we need the arrays. To populate the arrays with the data from our collections, we can use the `for` loop that follows the declaration of these three variables.

Because of the structure of the RSS in the remote XML file, some of the `title`, `description`, and `link` elements are irrelevant to the news items and instead refer to the RSS file itself and the service provided by the BBC. These are the first few examples of each of the elements in the file.

So how can we get rid of the first few items in each array? The standard JavaScript `.pop()` method allows us to discard the last item from the array, so if we reverse the arrays, the items we want to get rid of will be at the end of each array.

Calling `reverse` a second time once we've popped the items we no longer need puts the array back into the correct order. The number of items popped is specific to this implementation, other RSS files may differ in their structural presentation and therefore their `.pop()` requirements.

Now that we have the correct information in our arrays, we're ready to add some of the news items to our reader. The RSS file will contain approximately 20 to 30 different news items depending on the day, which is obviously far too many to display all at once in our reader.

There are several different things we could do in this situation. The first and arguably the most technical method would be to include all of the news items and then make our reader scroll through them. Doing this however would complicate the example and take the focus off of the Connection utility. What we'll do instead is simply discard all but the five newest news items. The `for` loop that follows the `.reverse()` and `.pop()` methods will do this.

The loop will run five times and on each pass it will first create a series of new `<p>`, `<div>`, and `<a>` elements using standard JavaScript techniques. These will be used to hold the data from the arrays, and to ultimately display the news items.

We can then use the DOM utility's highly useful `.addClass()` method (which IE doesn't ignore, unlike `setAttribute("class")`) to give class names to our newly created elements. This will allow us to target them with some CSS styles.

Then we obtain the `nodeValue` of each item in each of our arrays and add these to our new elements. Once this is done we can add the new elements and their `textNodes` to the `newsitems` container on the page. Save the file as `responseXML.html`.

## Styling the Newsreader

To make everything look right in this example and to target our newly defined classes, we can add a few simple CSS rules. In a fresh page in your text editor, add the following selectors and rules:

```
#newsreader {
  width:240px;
  border:2px solid #980000;
  background-color:#cccccc;
}
.header {
  font-weight:bold;
  font-size:123.1%;
  background-color:#980000;
  color:#ffffff;
  width:96%;
  padding:10px 0px 10px 10px;
}
.title {
  font-weight:bold;
  margin-left:10px;
  margin-right:10px;
}
```

```
.desc {
  margin-top:-14px;
  *margin-top:-20px;
  margin-left:10px;
  margin-right:10px;
  font-size:85%;
}
.newslink {
  text-decoration:none;
  color:#000000;
}
.link {
  text-decoration:none;
  color:#ffffff;
}
#footer {
  background-color:#980000;
  color:#ffffff;
  font-size:77%;
  padding:10px 0px 10px 10px;
}
```

Save this file as `responseXML.css`. All of the files used in this example, including the YUI files, will need to be added to the content-serving directory of your web server in order to function correctly.

Finally, in order to actually get the XML file in the first place, we'll need a little PHP. As I mentioned before, this will act as a proxy to which our application makes its request. In a blank page in your text editor, add the following PHP code:

```
<?php
define ('HOSTNAME', 'http://newsrss.bbc.co.uk/rss/
                newsonline_uk_edition/world/rss.xml');
$session = curl_init();
curl_setopt($session, CURLOPT_URL, HOSTNAME);
curl_setopt($session, CURLOPT_RETURNTRANSFER, true);
$xml = curl_exec($session);
curl_close($session);

if (empty($xml))
{
  print "Error extracting RSS file!";
}
else
{
```

```
header("Content-Type: text/xml");  
echo $xml;  
}  
  
?>
```

Save this as `myproxy.php`. The newsreader we've created takes just the first (latest) five news items and displays them in our reader. This is all we need to do to expose the usefulness of the Connection utility, but the example could easily be extended so scroll using the Animation and DragDrop utilities.

Everything is now in place, if you run the HTML file in your browser (not by double-clicking it, but by actually requesting it properly from the web server) you should see the newsreader as in the following screenshot:



## Useful Connection Methods

As you saw in the last example, the Connection Manager interface provides some useful methods for working with the data returned by an AJAX request. Let's take a moment to review some of the other methods provided by the class that are available to us.

The `.abort()` method can be used to cancel a transaction that is in progress and must be used prior to the `readyState` property (a standard AJAX property as opposed to YUI-flavoured) being set to 4 (complete).

The `.asyncRequest()` method is a key link in Connection's chain and acts like a kind of pseudo-constructor used to initiate requests. We already looked at this method in detail so I'll leave it there as far as this method is concerned.

A public method used to determine whether the transaction has finished being processed is the `.isCallInProgress()` method. It simply returns a boolean indicating whether it is true or not and takes just a reference to the connection object.

Finally `.setForm()` provides a convenient means of obtaining all of the data entered into a form and submitting it to the server via a GET or a POST request. The first argument is required and is a reference to the form itself, the remaining two arguments are both optional and are used when uploading files. They are both boolean: the first is set to `true` to enable file upload, while the third is set to `true` to allow SSL uploads in IE.

## A Login System Fronted by YUI

In our first Connection example we looked at a simple GET request to obtain a remote XML file provided by the BBC. In this example, let's look at the sending, or Posting of data as well.

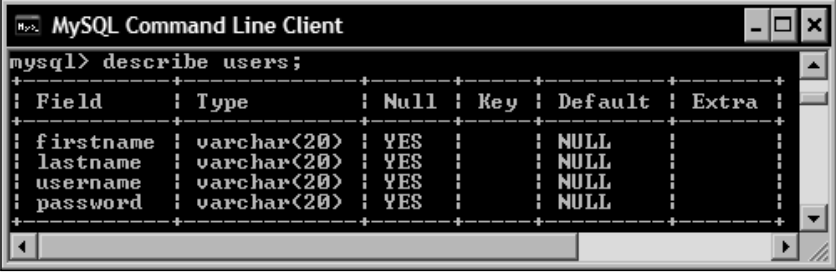
We can easily create a simple registration/login system interface powered by the Connection utility. While the creation of session tokens or a state-carrying system is beyond the scope of this example, we can see how easy it is to pass data to the server as well as get data back from it.

Again, we'll need to use a full web server set up, and this time we can also include a MySQL database as the end target for the data posted to the server. You'll probably want to create a new table in the database for this example.

In order to focus just on the functionality provided by the YUI, our form will have no security checks in place and data entered into the form will go directly into the database. Please do not do this in real life!

Security in a live implementation should be your primary concern and any data that goes anywhere near your databases should be validated, double-checked, and then validated again if possible, and some form of encryption is an absolute must. The MD5 hashing functions of PHP are both easy to use and highly robust.


Create a new table in your database using the MySQL Command Line Interface and call it **users** or similar. Set it up so that a `describe` request of the table looks like that shown in the screenshot overleaf:



```
mysql> describe users;
```

| Field     | Type        | Null | Key | Default | Extra |
|-----------|-------------|------|-----|---------|-------|
| firstname | varchar(20) | YES  |     | NULL    |       |
| lastname  | varchar(20) | YES  |     | NULL    |       |
| username  | varchar(20) | YES  |     | NULL    |       |
| password  | varchar(20) | YES  |     | NULL    |       |

For the example, we'll need at least some data in the table as well, so add in some fake data that can be entered into the login form once we've finished coding it. A couple of records like that shown in the figure below should suffice.



```
mysql> insert into users (firstname, lastname, username, password) values ('David', 'Mitchell', 'mitchd', 'mypass1'), ('Robert', 'Webb', 'webbr', 'mypass2');
```

|        |          |        |         |
|--------|----------|--------|---------|
| David  | Mitchell | mitchd | mypass1 |
| Robert | Webb     | webbr  | mypass2 |

Start off with the following basic web page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
      charset=utf-8">
    <title>Yui Connection Manager Example 2</title>
    <script type="text/javascript"
      src="yui/yahoo-dom-event.js"></script>
    <script type="text/javascript"
      src="yui/connection-min.js"></script>
  </head>
  <body>
    <form id="signin" method="get" action="#">
```

```
<fieldset>
  <legend>Please sign in!</legend>
  <label>Username</label><input type="text" id="uname"
    name="uname">
  <label>Password</label><input type="password" id="pword"
    name="pword">
  <button type="button" id="login">Go!</button>
  <button type="reset">Clear</button>
</fieldset>
</form>
<form id="register" method="post" action="#">
  <fieldset>
    <legend>Please sign up!</legend>
    <label>First name:</label><input type="text" name="fname">
    <label>Last name:</label><input type="text" name="lname">
    <label>Username:</label><input type="text" name="uname">
    <label>Password:</label><input type="password" name="pword">
    <button type="button" id="join">Join!</button>
    <button type="reset">Clear</button>
  </fieldset>
</form>
</body>
</html>
```

The `<head>` section of the page starts off almost exactly the same as in the previous example, and the body of the page just contains two basic forms. I won't go into specific details here. The mark up used here should be more than familiar to most of you. Save the file as `login.html`.

We can also add some basic styling to the form to ensure that everything is laid out correctly. We don't need to worry about any fancy, purely aesthetic stuff at this point, we'll just focus on getting the layout correct and ensuring that the second form is initially hidden.

In a new page in your text editor, add the following CSS:

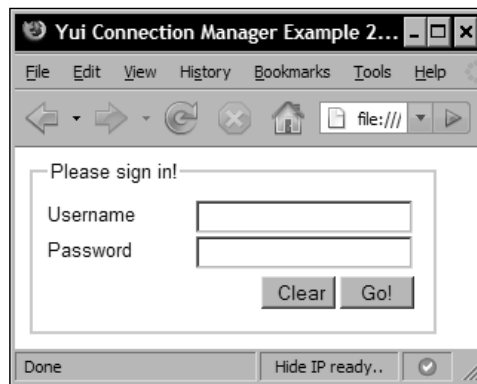
```
fieldset {
  width:250px;
  padding:10px;
  border:2px solid lightblue;
}
label {
  margin-top:3px;
  width:100px;
  float:left;
```

```

}
input {
  margin-top:2px;
  *margin-top:0px;
}
button {
  float:right;
  margin:5px 3px 5px 0px;
  width:50px;
}
#register {
  display:none;
}

```

Save the file as `login.css` and view it in your browser. The code we have so far should set the stage for the rest of the example and appear as shown in the figure below:



Now let's move on to the real nuts and bolts of this example—the JavaScript that will work with the Connection Manager utility to produce the desired results. Directly before the closing `</body>` tag, add the following `<script>`:

```

<script type="text/javascript">
  //create namespace object
  YAHOO.namespace("yuibook.login");

  //define the submitForm function
  YAHOO.yuibook.login.submitForm = function() {

    //have both fields been completed?
    if (document.forms[0].uname.value == "" ||
        document.forms[0].pword.value == "") {
      alert("Please enter your username AND password to login");
      return false;
    }
  }

```

```
    } else {  
        //define success handler  
        var successHandler = function(o) {  
            alert(o.responseText);  
  
            //if user not found show register form  
            if (o.responseText == "Username not found") {  
                YAHOO.util.Dom.setStyle("register", "display", "block");  
                document.forms[0].uname.value = "";  
                document.forms[0].pword.value = "";  
            }  
        }  
  
        //define failure handler  
        var failureHandler = function(o) {  
            alert("Error " + o.status + " : " + o.statusText);  
        }  
  
        //define callback object  
        var callback = {  
            success:successHandler,  
            failure:failureHandler  
        }  
  
        //harvest form data ready to send to the server  
        var form = document.getElementById("signin");  
        YAHOO.util.Connect.setForm(form);  
  
        //define a transaction for a GET request  
        var transaction = YAHOO.util.Connect.asyncRequest("GET",  
                                                         "login.php", callback);  
    }  
}  
  
//execute submitForm when login button clicked  
YAHOO.util.Event.addListener("login", "click", YAHOO.yuibook.login.  
submitForm);  
</script>
```

We use the Event utility to add a listener for the `click` event on the login button. When this is detected the `submitForm()` function is executed. First of all we should check that data has been entered into the login form. As our form is extremely small, we can get away with looking at each field individually to check the data has been entered into it.

Don't forget that in a real-world implementation, you'd probably want to filter the data entered into each field with a regular expression to check that the data entered is in the expected format, (alphabetical characters for the first and last names, alphanumeric characters for the username, and password fields).

Provided both fields have been completed, we then set the success and failure handlers, the callback object and the Connection Manager invocation. The failure handler acts in exactly the same way as it did in the previous example; the status code and any error text is alerted. In this example, the success handler also sends out an alert, this time making use of the `o.responseText` member of the response object as opposed to `responseXML` like in the previous example.

If the function detects that the response from the server indicates that the specified username was not found in the database, we can easily show the registration form and reset the form fields.

Next, we define our callback object which invokes either the `success` or `failure` handler depending on the server response. What we have so far is pretty standard and will be necessary parts of almost any implementation involving Connection.

Following this we can make use of the so far unseen `.setForm()` method. This is called on a reference to the first form. This will extract all of the data entered into the form and create an object containing `name:value` pairs composed of the form field's names and the data entered into them. Please note that your form fields must have `name` attributes in addition to `id` attributes for this method to work.

Once this has been done, we can initiate the Connection Manager in the same way as before. Save the HTML page as `login.html` or similar, ensuring it is placed into a content-serving directory accessible to your web server.

As you can see the `.setForm()` method is compatible with `GET` requests. We are using the `GET` method here because at this stage all we are doing is querying the database rather than making physical, lasting changes to it. We'll be moving on to look at `POST` requests very shortly.

Now we can look briefly at the PHP file that can be used to process the login request. In a blank page in your text editor, add the following code:

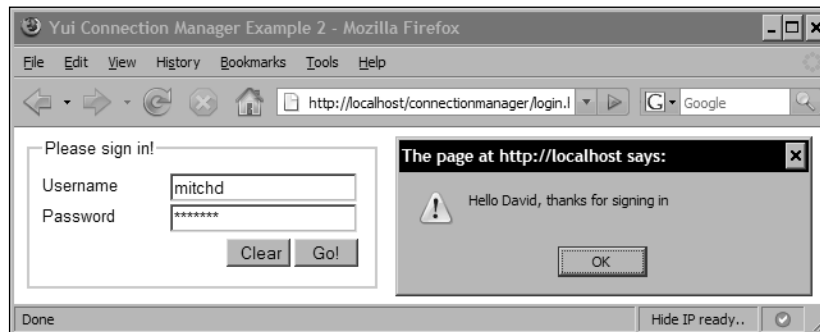
```
<?php
    $host = "localhost";
    $user = "root";
    $password = "mypassword";
    $database = "mydata";
    $uname = $_GET["uname"];
    $pword = $_GET["pword"];
    $server = mysql_connect($host, $user, $password);
```

```
$connection = mysql_select_db($database, $server);
$query = mysql_query("SELECT * FROM users WHERE username LIKE
                      '$uname%'");
$rows = mysql_num_rows($query);
if ($rows != 0)
{
    $row = mysql_fetch_array($query);
    $pass = $row['password'];
    if ($pass != $pword)
        echo "Password incorrect";
    else
        echo "Hello ".$row['firstname'].", thanks for signing in";
}
else
{
    echo "Username not found";
}
mysql_close($server);
?>
```

Here we set up the variables required to query our database and then extract any records where the username entered into our form matches the username associated with a user. There should only be one matching record, so we can then compare the stored password with that entered into our form.

All we're doing here is passing back an appropriate message to be displayed by our `successHandler` function back in the HTML page. Normally, after entering the correct credentials, the visitor would be redirected to their account page or some kind of personal home page, but a simple alert gives us everything we need for this example. Save the above file as `login.php` in the same directory as the web page and everything should be good to go.

Try it out and reflect upon the ease with which our task has been completed. Upon entering the username and password of one of our registered users, you should see something similar to the figure below:



So that covers GET requests, but what about POST requests? As I mentioned before, the `.setForm()` method can be put to equally good use with POST requests as well. To illustrate this, we can add some additional code which will let unregistered visitors sign up.

Add the following function directly before the closing `</script>` tag:

```
//define registerForm function
YAHOO.yuibook.login.registerForm = function() {
    //have all fields been completed?
    var formComp = 0;
    for (x = 1; x < document.forms[1].length; x++) {
        if (document.forms[1].elements[x].value == "") {
            alert("All fields must be completed");
            formComp = 0;
            return false;
        } else {
            formComp = 1;
        }
    }
    if (formComp != 0) {
        //define success handler
        var successHandler = function(o) {
            //show succes message
            alert(o.responseText);
        }
        //define failure handler
        var failureHandler = function(o) {
            alert("Error " + o.status + " : " + o.statusText);
        }
        //define callback object
        var callback = {
            success:successHandler,
            failure:failureHandler
        }
        //harvest form data ready to send to the server
        var form = document.getElementById("register");
        YAHOO.util.Connect.setForm(form);
        //define transaction to send stuff to server
        var transaction = YAHOO.util.Connect.asyncRequest(
            "POST", "register.php", callback);
    }
}
//execute registerForm when join button clicked
YAHOO.util.Event.addListener("join", "click",
    YAHOO.yuibook.login.registerForm);
```

In the same way that we added a listener to watch for clicks on the `login` button, we can do the same to look for clicks on the `join` button. Again we should check that data has been entered into each field before even involving the Connection Manager utility. As there are more fields to check this time, it wouldn't be efficient to manually look at each one individually.

We use a for loop and a control variable this time to cycle through each form field; if any field is left blank the `formComp` control variable will be set to 0 and the loop will exit. We can then check the state of `formComp` and provided it is not set to 0, we know that each field has been filled in.

The success and failure handlers are again based on simple alerts for the purpose of this example. We again use the `.setForm()` method to process the form prior to sending the data. We can then proceed to initiate the Connection Manager, this time supplying POST as the HTTP method and a different PHP file in the second argument of the `.asyncRequest()` method.

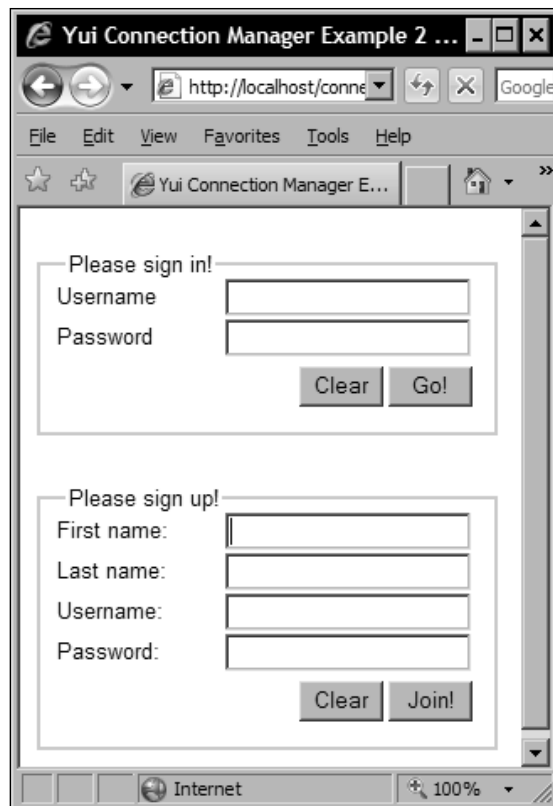
All we need now is another PHP file to process the registration request. Something like the following should suffice for this example:

```
<?php
    $host = "localhost";
    $user = "root";
    $password = "mypassword";
    $database = "mydata";
    $fname = $_POST["fname"];
    $lname = $_POST["lname"];
    $uname = $_POST["uname"];
    $pword = $_POST["pword"];

    $server = mysql_connect($host, $user, $password);
    $connection = mysql_select_db($database, $server);
    $query = mysql_query("INSERT INTO users VALUES ('$fname', '$lname',
                                                    '$uname',
                                                    '$pword')");

    echo "Thanks for joining us ".$fname;
    mysql_close($server);
?>
```

As we're using POST this time, we can use the `$_POST` superglobal to pull our values out, and can then run a simple INSERT query to add them to the database. Upon entering an unrecognized name into the first form, the registration form should then be displayed, as in the following screenshot:



If you register a new user now and then take a look at your database with the `mysql` Command Line Client, you should see the new data appear in your database:

The screenshot shows the MySQL Command Line Client window. The output is a table with the following data:

| firstname | lastname | username | password |
|-----------|----------|----------|----------|
| David     | Mitchell | mitchd   | mypass1  |
| Robert    | Webb     | webbr    | mypass2  |
| Peter     | Kay      | kayp     | mypass3  |

## Summary

The YUI Connection Manager utility provides an almost unequalled interface to AJAX scripting methods used today among the many JavaScript libraries available. It handles the creation of a cross-platform XMLHttpRequest object and provides an easy mechanism for reacting to success and failure responses among others.

It handles common HTTP methods such as GET and POST with equal ease and can be put to good use in connection (no pun intended) with a PHP (or other form of) proxy for negotiating cross-domain requests.