Chapter 18. AJAX: Asynchronous JavaScript and XML

Table of Contents

Objectives

At the end of this unit you will be able to:

- Make HTTP requests using AJAX;
- Use XML HttpRequest objets;
- Get data to and from the server using AJAX;
- Perform asynchronous communication using AJAX.

18.1 Introduction

With the growth of Web applications, websites have been trying to make user interaction similar to that found in desktop applications. The largest push is towards making Web interfaces more dynamic: if data changes, say a new email arrives in your inbox in GMAIL, the Web page should update and display the new email without reloading the whole page. The page should always be available to the user and always responsive to their input. While the DOM allows for such dynamic updates to occur, AJAX is an important group of technologies that allows a Web page to request more information from the server (such as an updated list of email in an inbox) without having to reload the whole page. Together, AJAX and the DOM can be used to create dynamic Web pages, and this chapter will introduce you to how that can be done.

AJAX stands for **Asynchronous JavaScript and XML**. It allows a Web page to make a request to a Web server for information using standard HTTP, but without reloading the page, and without automatically displaying the information returned from the server. These requests are all made programmatically, using JavaScript, and data communication is often done using XML, as JavaScript can easily parse this data. After receiving the data from the server, the JavaScript script can use the returned data however it wishes. The requests can also be made in a such a way that the JavaScript code does not have to wait for a reply from the server. Instead, the JavaScript code is notified when the page has finished receiving the information. In this way, the script can continue to perform useful actions while the data downloads from the server – this makes the communication asynchronous to any action that the Web page is performing.

It is important to realise that AJAX itself refers to a group of related technologies, not all of which are standardised, and not all of which need to be used at once. For instance, it can often be more convenient to communicate with the server using plain text rather than XML, and these communications need not occur asynchronously. Various other technologies are often employed in addition to those making up the AJAX

name itself. The original article that defined the term AJAX lists the following technologies:

- XHTML and CSS, which defines what is being displayed and how it is displayed.
- The Document Object Model, which allows us to programmatically alter the content and how it is displayed.
- XML and XSLT, which is used to transfer data between the server and Web browser (using XML), and to manipulate that data (using XSLT).
- XMLHttpRequest, which is the object used to communicate with the Web server over HTTP. This object provides the asynchronous communication abilities.
- JavaScript, which is the programming language implemented in most Web browsers, and is used to bring together all the other technologies listed above.

Most of these technologies are discussed elsewhere in these notes. The rest of this chapter will examine how XMLHttpRequest can be used to communicate with the Web server.

18.2 Initialisation

The first step to making an HTTP request using AJAX is to create an instance of one of the XMLHttpFunction function objects. There are two different objects that can be used, depending on the browser: Mozilla based browsers – such as Firefox – and Safari (used on OS X) both use XMLHttpRequest() objects. Internet Explorer, on the other hand, uses ActiveXObject(). An object can easily be created using the following code:

```
var httpRequest;
if (window.XMLHttpRequest) { // Mozilla, Safari, ... httpRequest =
    new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE
    httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
}
```

The methods and properties associated with both of these objects are identical: the major difference between them is how they are initialised. During these notes we will speak of XMLHttpRequest to mean both the Mozilla / Safari XMLHttpRequest objects, and Microsofts XMLHTTP ActiveX object, unless stated otherwise.

18.3 Synchronous example

This simple example will show you to use XML HttpRequest objects to request a file using JavaScript. In the same directory, create two files: one called sync-file.html, and the other called text.txt. The text.txt can contain any small amount of text that you like; we suggest using, Hello, world! This is the file that you will be requesting using the XMLHttpRequest object. Make sure that the files are in the same directory, as most browsers have security restrictions that will stop XMLHttpRequest objects from being able to access files outside of the directory that the HTML file making the request is in.

Add the following HTML to sync-file.html:

```
<html>
  <head>
    <title>This is a title</title>

  </head>
  <body>
    <script type="text/javascript">

      function create_request() { var httpRequest;
         if (window.XMLHttpRequest) { httpRequest = new
```

```
            XMLHttpRequest();
         }
         else if (window.ActiveXObject) {
            httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
         }
         else
         {
            document.write("Unable to create XMLHttpRequest
            object.");
         }


         return httpRequest;
      }

      var httpRequest = create_request();

      httpRequest.open("GET", "file://<directory>/text.txt", false);
      httpRequest.send(null);
      if (httpRequest.status == 0) // For non http requests.
      {
         document.write(httpRequest.responseText);
      }
      else
      {
         document.write("There has been a problem opening the
         file.");
      }

   </script>

  </body>
</html>
```

There are a few things that you should note about this example. First, the code to create the XMLHttpRequest object has been hidden in a function called create_request(). Note that this is not a constructor function. We will continue to use this function throughout this chapter. Second, since this code is synchronous, the request to read the document (described in detail below) returns before the page continues to be displayed by the browser. This lets us make use of document.write(), which we will not be able to use so easily after the page has finished being loaded.

Now look closely at the following lines:

```
httpRequest.open("GET", "file://<directory>/text.txt", false);
httpRequest.send(null);
```

<directory> should be replaced with the directory that the files have been saved in.

The **open** method does several things. Firstly, it does not actually open the file or Web resource requested by the method. Rather, it sets up a request for a file. The **send** method sends this request, along with any other information the programmer specifies. Only when **send** is called will any data transfer take place.

The first argument to **open** specifies the HTTP request method, as described in the previous chapter on Web application development.

The second argument is the resource that should be accessed. In this case it is a file on the hard drive, but it will usually be a resource on the Web. Note that browser security will generally stop an HTML page or JavaScript file retrieved over the Internet from accessing files on your hard drive – only html files that you saved onto your drive can generally do this. The third argument specifies whether the data should be transferred asynchronously (with the value

true), or synchronously (false). The difference to the programmer is that data transferred synchronously will be available immediately for use as soon as the **send** method is called. In fact, the **send** method will not return until all the data has been sent and received, which can cause the **send** method to take a fair amount of time to return. In turn, this can cause the JavaScript script to completely freeze while waiting for data from the server. On the other hand, if asynchronous communication is requested, the **send** method will return immediately, and all the communication will occur in the background. Unlike with synchronous communication, the data will not be ready when the **send** method returns.

18.4 A more detailed example

The previous example is quite trivial, since the AJAX call takes place while the page is being loaded. This example will go a step further: the AJAX request will occur after the page has completely loaded, and the page will be dynamically updated. This will be done by showing a button to the user. Pressing the button will make an AJAX request to read a file, the contents of which it will be printed on the page. While this could be done without AJAX, AJAX will allow us to do this without reloading the page.

First, create two files, one called text.txt and another called div-update.html. Text.txt should contain a small message, just as it previously did. Div-update.html should contain the following:

```
<html>
  <head>
    <title>This is a title</title>

  </head>
  <body>
    <script type="text/javascript">

      function create_request() { var httpRequest;
         if (window.XMLHttpRequest) { httpRequest = new
            XMLHttpRequest();
         }
         else if (window.ActiveXObject) {
            httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
         }
         else
         {
            document.write("Unable to create XMLHttpRequest
            object.");
         }

         return httpRequest;
      }

      function get_data_synchronously_from(file_name)
      {
        var httpRequest = create_request();

        httpRequest.open("GET", file_name, false);
        httpRequest.send(null);
        if (httpRequest.status != 0) // For non http requests.
        {
          alert("There has been an error reading the file."); return
          null;
        }
        else
        {
          return httpRequest.responseText
        }
      }
```

```
      function button_press()
      {
 document.getElementById("display-div").innerHTML=
 get_data_synchronously_from(
      }

   </script>

   <p>
     Click on the button below to load data from the file in the
     text box.
   </p>

   <form>
     <input type="text" name="file-name"
     value="file:///home/rudy/Desktop/web/
     <input type="button" value="Print the file"
     onclick="button_press();"/>
   </form>

 <p> <b>The file currently contains: </b></p>
 <div id="display-div">
  <p>The file has not yet been loaded.</p>
 </div>

                                  </body>
</html>
```

The example is similar to the previous one in how it handles the AJAX requests. Notice that we've encapsulated the request in the get_data_synchronously_from() function, which takes a file name and now returns the contents of the file. The HTML page contains a DIV with the ID “display-div”. It is the DIV which we will use to display the contents of the file. The page also contains a form with two elements, a text box, which we have named “file-name”, to contain the name of a file which the user wants to view, and a button. The button calls the **button_press()** function when it is clicked. This function does all of the interesting work of this example: it uses **getElementsByName()** to access the text box (notice that **getElementsByName()** returns an array of elements with the given name – in this case there is only one element, which is why we take the first), whose value is used as the argument to get_data_synchronously_from(). **getElementById()** is then used to access the DIV, and we set its contents using its **innerHTML()** attribute to the contents in the file.

The user can enter the name of any text file in the text box. The file should be in the same directory as the HTML file, otherwise some browsers will report a security error. Also, the file should have its full path specified, otherwise, again, many browsers will report a security error. When the user presses the button, the HTML page is dynamically modified (modified without reloading the page) to contain the contents of the file. Notice that the file can contain HTML code as well, and this will be displayed.

18.5 Getting data from a server

Our previous examples have all only read data from a file on your computer. However, AJAX is typically used to read files off of a Web server. The only substantial difference is that the a Web server will return a response status other than 0 (typically, if you recall from the previous chapter, the value 200), which is what we have been testing for in the examples.

This example will make use of the Tomcat Web server from the previous chapter. Create a directory in the Tomcat webapps subdirectory titled “random”. The first thing that we will work on is a simple Servlet that will return random numbers, using Java Random objects. In the classes directory, add the class for this Java file:

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*; import java.util.Random;

public class RandomServlet extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse
        response)
    {
 try {
     PrintWriter out = response.getWriter(); out.println((new
     Random()).nextInt());
 }
 catch (IOException e) {
     response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
 }
    }
}
```

The RandomServlet class merely instantiates the Random class, and returns the next random integer. It also handles any errors.

The appropriate web.xml file (added, as you can recall, to the WEB-INF directory), is:

```
<?xml version="1.0"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN_
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Random Numbers</display-name>
  <description>
    Returns random numbers
  </description>

  <servlet>
    <servlet-name>RandomServlet</servlet-name>
    <description>
      A simple servlet that returns HTML fragments of random numbers.
    </description>

    <servlet-class>RandomServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>RandomServlet</servlet-name>
    <url-pattern>/random</url-pattern>
  </servlet-mapping>
</web-app>
```

You can test this Web application by starting Tomcat and visiting the URL: http://localhost:8080/ random/random

The AJAX portion of this example is the index.html file, which should be added directly into the random directory:

```
<html>
  <head>
    <title>Press a button to get a number!</title>

  </head>

  <body>
    <script type="text/javascript">
```

```
        function create_request() { var httpRequest;
           if (window.XMLHttpRequest) { httpRequest = new
              XMLHttpRequest();
           }
           else if (window.ActiveXObject) {
              httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
           }
           else
           {
              document.write("Unable to create XMLHttpRequest
              object.");
           }

           return httpRequest;
        }

        function get_data_synchronously_from(url_name)
        {
          var httpRequest = create_request();

          httpRequest.open("GET", url_name, false);
          httpRequest.send(null);
          if (httpRequest.status != 200) // For non http requests.
          {
            alert("There has been an error reading the URL."); return
            null;
          }
          else
          {
            return httpRequest.responseText
          }
        }

        function button_press()
        {
     document.getElementById("display-div").innerHTML=
     get_data_synchronously_from(
        }

                                  </script>

     <h1> Random numbers from the server: </h1>
      <form>
            <input type="button" value="Print a number!"
            onclick="button_press();"/
                                   </form>
     <div id="display-div">
      <p>The button hasn't yet been pressed.</p>
     </div>

      </body>
    </html>
```

This file is very similar to the previous example. Notice that get_data_synchronously_from() has been updated to retrieve data from an URL. Importantly, the status of the response object now returns the value 200 when the the HTTP request has been successful.

Again, there is a DIV tag with an ID whose data is replaced with the random number. The random number obtained from the RandomServlet we previously wrote.

Play with this application. You will see that the random number is always updated without reloading the page. Also notice that no matter what random number is displayed on the page, if you ask the browser to show you the page's source, you will never find the number in the source. This is because the Web page is always being dynamically updated.

18.6 Sending data to a server

For this example we will update the greetings Web application from the Web application chapter to use AJAX.

Create a new directory in webapps called “ajax-greetings”. We will use the following class, which you should compile and place in the classes directory:

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*; import java.util.Random;

public class ResponseServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
    {
 try {

     String name = request.getParameter("name"); PrintWriter out =
     response.getWriter();

     if (name == null) {
  out.println("No name has been given."); return;
     }

     out.println(getGreeting() + name);
 }
 catch (IOException e) {
     response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERR
     OR);
 }
    }

    String getGreeting()
    {
 switch((new Random()).nextInt(5)) { case 0:
     return "Hello, "; case 1:
     return "Nice to meet you, "; case 2:
     return "Hi, "; case 3:
     return "Yo, "; case 4:
     return "Hey, ";
 }
 return "Hello, ";
    }
}
```

This class accepts POST requests only. It is expecting to be passed the user's name in the “name” parameter, and then uses the getGreeting() method to find a random way of greeting the person. The Servlet then returns a single line, greeting the user.

The appropriate web.xml file is:

```
<?xml version="1.0"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>A Greetings Application</display-name>
  <description>Asks the user's name and then greets
  them.</description>

  <servlet>
    <servlet-name>ResponseServlet</servlet-name>
    <description>Performs the actual greeting.</description>

    <servlet-class>ResponseServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ResponseServlet</servlet-name>
    <url-pattern>/response</url-pattern>
  </servlet-mapping>
</web-app>
```

The index.html file, which should be placed directly in the ajax-greetings directory, follows:

```
<html>
                                <head>
  <title>Some Greetings</title>
                                </head>
                                <body>
  <script type="text/javascript">

   function create_request() { var httpRequest;
    if (window.XMLHttpRequest) { httpRequest = new
     XMLHttpRequest();
    }
    else if (window.ActiveXObject) {
       httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else
    {
     document.write("Unable to create XMLHttpRequest object.");
    return httpRequest;
         }

   function get_data_synchronously_from(file_url, variables)
   {
    var httpRequest = create_request();

    httpRequest.open("POST", file_url, false);
    httpRequest.setRequestHeader("Content-Type", "application/x-
    www-form-urlenc httpRequest.send(variables);
    if (httpRequest.status != 200) // For non http requests.
    {
     alert("There has been an error reading the URL. " +
     httpRequest.status); return null;
    }
    else
```

```
    {
     return httpRequest.responseText
    }
   }

   function button_press()
   {
    document.getElementById("greeting-div").innerHTML=
    get_data_synchronously_f
   }
      </script>

  <div id="greeting-div"> What is your name?
  </div>
  <form>
   <input type="text" name="user_name">
   <input type="button" value="Say Hello!"
   onclick="button_press();">
    </form>
                              </body>
</html>
```

The page appears as before, with a text box for the user to enter their name and a button to press to return a greeting. The page displays this greeting in a DIV with an ID, which is how we have dynamically updated Web pages previously in this chapter. The bulk of the work is done in the get_data_synchronously_from() function, which has been updated to accept a second argument, “variables”. This is a list of data to be sent to the server, and is formatted as a list of names and their values: name=value&name2=value2&name3=value3 and so on. In this case, we only provide one name / value pair when calling this function. We do this in the button_press() function, which takes the value from the text box.

get_data_synchronously_from() also makes a call to setRequestHeader() to tell the server that data has been sent. If this call is not made, the server will ignore all data sent to it by this request. Next, variables is passed to the send function. The rest of the function operates exactly as before.

You can now test the application. You should notice that the Web application greets the user without reloading the page. The application also uses multiple, random greetings.

18.7 Performing asynchronous communication

When creating an interactive application it is important to not let the application unduly freeze while it is performing an action. This is equally true when making a request to a server using an XMLHttpRequest object: when making a request from the server, the interface should remain usable, responding to actions from the user, even if only to say that the Web application is currently busy. One important place that special care may be required is when the XMLHttpRequest results in enough information being transferred from the server that there is a noticeable delay in the browser receiving it. For instance, this can easily occur when downloading a large document to embed in the Web page. But remember that often there can be delays even when transferring very small amounts of information, since some users may be connecting to the Web application using slower connection methods, or data may have been lost during transit.

Application freezing usually occurs when send is used and the open method – which you should remember is called just before calling send – was called with the third argument set to false. In this case send does not return until it has finished receiving all the data being sent from the server. If it takes ten seconds to receive the data, it takes ten seconds for the send function to return from its call. And for those ten second the browser will not perform any other action, including accepting user input.

We call the form of communication where we wait for the server to finish sending the browser information before continuing to execute the Web application synchronous communication. As previously mentioned, AJAX allows for asynchronous communication: while the data is being transferred, the Web application can continue to execute, and can accept user input or perform other functions.

Asynchronous communication is fairly easy to perform: instead of passing false to the open function, we pass true to indicate that we want to communicate asychronously with the server. When send is then called, send will immediately return. In the synchronous case this would have meant that communication with the server is now complete, and any information that it was sending to the browser is now available. In the asynchronous case this is not true. Rather, we supply the XMLHttpRequest object with a function which it should call when the state of the communication changes. For instance, the method is called when the communication begins, and when the communication ends. The XMLHttpRequest object has a property called readyState, which indicates when the communication has finished: if it holds the integer value 4, then communication is done, and the data is ready to be used. The data may then be read from the XMLHttpRequest object in the usual way.

The example for this section is a rewrite of the Random Web application we developed above so that it performs its request for a random number asynchronously. Begin by copying the contents of the random directory to a new directory in webapps called async-random.

Edit the index.html file so that it contains the following:

```
<html>
 <head>
  <title>Press a button to get a number!</title>

 </head>
 <body>

 <script type="text/javascript"> function create_request() {
  var httpRequest;
  if (window.XMLHttpRequest) { httpRequest = new XMLHttpRequest();
  }
  else if (window.ActiveXObject) {
   httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
  }
  else
  {
   document.write("Unable to create XMLHttpRequest object.");
  }

  return httpRequest;
       }

 function get_data_asynchronously_from(file_url, variables,
 event_handler)
 {

  var httpRequest = create_request();

  httpRequest.onreadystatechange = function() {
   event_handler(httpRequest);
  }

  if (variables != null)
  {
   httpRequest.open("POST", file_url, true);
   httpRequest.setRequestHeader("Content-Type", "application/x-
   www-form-urlenco httpRequest.send(variables);
  }
  else
  {
   httpRequest.open("GET", file_url, true);
   httpRequest.send(null);
  }

 }
```

```
function is_data_ready(request)
{
 return request.readyState == 4;
}

function get_value(request)
{
 if (request.status != 200) // For non http requests.
 {
  alert("There has been an error reading the URL. " +
  request.status); return null;
 }
 else
 {
  return request.responseText
 }
}

function button_press()
{
 function set_value(request)
 {
  if (is_data_ready(request))
               document.getElementById("display-
              div").innerHTML=get_value(request);
 }

 get_data_asynchronously_from("http://localhost:8080/random/rando
 m", null, set
}

</script>

<h1> Random numbers from the server: </h1>
<form>
 <input type="button" value="Print a number!"
 onclick="button_press();"/>
    </form>
<div id="display-div">
 <p>The button hasn't yet been pressed.</p>
</div>
 </body>
</html>
```

The first large change is the **get_data_asynchronously_from()** function, which now accepts a third argument, which will be the name of a function to be used in the XMLHttpRequest **onreadystatechange** event handler: this is the function that will handle the data retrieved from the server. The function passed to **get_data_asynchronously_from()** should always accept one argument, which will be the XMLHttpRequest object which is retrieving the data. This allows the function to test if the server has finished sending the data, and to obtain the data from the object. The first two arguments to **get_data_asynchronously_from()** are exactly the same as previously.

After creating the XMLHttpRequest object, the first thing that **get_data_asynchronously_from()** does is to set the event handler which will be called when the data is ready. Note that the event handler for **onreadystatechange** does not accept any arguments, so a new function is created which calls the event_handler function passed to **get_data_asynchronously_from()** and passes it the XMLHttpRequest object.

The next if statement determines if any data (contained in the **variables** variable) is being sent to the server, and decides on whether POST or GET request methods should be used.

Finally, **send** is called, either passing the data or passing null, as necessary.

There are two new functions: **is_data_read()** takes a XMLHttpRequest object and returns true if communications with the server has completed; otherwise it returns false.

get_value() takes an XMLHttpRequest object and is exactly the code from the old **get_data_synchronously_from()** functions that tests to see what status the server has given for the HTTP request, and returns the data.

button_press() has also been updated. First, it creates a function called set_value, which is the event handler we will pass to **get_data_asynchronously_from**. This function first tests to see that the XMLHttpRequest object has completed communications with the server (since the event handler is called whenever communication state with the server changes, and not only when communication completes). It then retrieves the sent data, and updates the Web page as in the previous example.

18.8 Advantages and disadvantages of AJAX based Web pages

AJAX is increasingly being used on the Web today. The main reasons for this are:

- AJAX allows a Web page to only reload those portions which have changed, rather than reloading the whole page. This can substantially reduce the amount of bandwidth that the Web application requires.

- Because AJAX allows for asynchronous communication, and for only reloading a portion of a Web document, it allows the Web application to be more frequently available to the user, since page reloads are reduced, and the user can continue using the application while data is being transmitted.

However, AJAX does have disadvantages, which need to be weighed up against the advantages, and mitigated where possible:

- Dynamically created pages, such as most of those using AJAX, often renders the browser's history functionality useless. If the user moves away from a page, and then presses the back button to return to it, the page is not necessarily shown in the same state as when the user left it. It may be useful to expose all distinct functionality as separate URLs, so that the user can visit them using the back and forward buttons, or the browser's history list.

- Similar to the above problem, it can be difficult to bookmark any useful page and return to it, since what the user was actually interested in bookmarking had been loaded dynamically.

- Search engines indexing a website are not likely to find any of the dynamically generated content, which will effect how the site appears in search engines' rankings.