Bernd Owsnicki-Klewe

Distributed Software Architectures

2. Client-Side Programming

2. Client-Side Programming (CSP)

CSP describes the part of a distributed application that uses a client (usually a http client, aka browser) to actually execute part of the application code. In many cases, some parts of such an application can well be executed on the client, because they might not require data to be transmitted from a server, thereby reducing client-server transmission bandwidth and response time.

On the downside, the history of CSP has been bothered with proprietary extravaganzas that make it difficult for a Software Engineer to come up with platform-independent solutions – a highly frustrating field known as "cross-browser compatibility".[1]

As a way out, many engineers resort to retaining functionality on the server side where they have more control over the run-time environment or using dedicated libraries that iron out these idiosyncrasies.[2]

So, let's look at the principles of JavaScript (JS) as the foundation of browser-based CSP first. We will look at language basics and then concentrate on some aspects of its functional programming principles together with the JS event model.

After that, we will have a brief look at one of those libraries that are frequently used to streamline the implementation of client-side functionalities – in this case *jQuery*. A relatively comprehensive list of JS libraries can be found under http://en.wikipedia.org/wiki/List_of_JavaScript_libraries.

One of the most important aspects will be a short introduction into *AJAX* (Asynchronous JavaScript And XML) that ties CSP together with the XML technology examined in the previous chapter.

2.1 JavaScript

JavaScript (JS) is a programming language that was developed as an implementation of the *EcmaScript* standard that was developed in the 90s of the 20th century. Trying to characterize JS can be difficult, because it has many properties that can be considered central to the language design.

In general, the following terms are found:

- *object-oriented*: even though the language name suggests a close relationship to Java, the object model of JS is distinctly different. We won't go deeper into this issue here, because it's not too important for the actual use – although it's very interesting to study in itself.

- *interpreted*: the language is not compiled into machine or byte code as are C, C+

1 Connect to UWF Argus with Firefox from unix and you're getting a taste of this :-)

2 I just spelled this "idiosyncrazies"! Freudian slip?

+ and Java. The interpreter is embedded in the browser and analyzes the JS source code line by line. It shares this with languages such as *Scheme* and *Prolog*, but also with *PHP*, *Haskell* and others.

- *functional*: JS treats functions (à la C) in a much more general fashion – it actually can be compared to functional languages like *Scheme* or *Haskell*. Functions are *first-class objects*, meaning they can be built at run-time, passed as parameters into or returned from functions and they can be stored as variable values or inside complex data structures. That allows for interesting programming techniques, such as *delayed/lazy evaluation*, *infinite data structures* (!), *closures and continuations*.

- *dynamic*: JS acknowledges that the value of a variable has a particular type (like a numeric type, a String type or a function type) *at a particular moment during program execution* but it provides *no way to assign types to variables*, like Java does. Technically, that means that a variable can have a numeric value, then maybe take a string and later even take a function as a value. It can also dynamically execute strings that represent JS code by a mechanism called *evaluation*.

From a programming language standpoint, JS is a highly interesting language forming a stark contrast to static languages, such as C, C++ and Java. Here we will concentrate on languages features that make it interesting for programming a client in a distributed setting.

Since there are a lot of good introductions into JS available, let's study it along the lines of a couple of examples!

2.1.1 JS and HTML

JS can be a) embedded into an HTML file or b) in a separate file referenced from an HTML file. Whenever a script element is detected by the browser, it will immediately execute the contained JS code. The following examples simplify the structure of an HTML document in order to make it easier to understand (the scripting part is shown in red):

```
<html>
   <head>
      <title>Say Hello!</title>
   </head>
   <body>
      <p>
         <script type="text/javascript">
            document.write('<b>Hello, world!</b>');
         </script>
```

```
      </p>
   </body>
</html>
```

In this case, the browser executes the JS code inside the script element and dynamically inserts the HTML fragment over the document.write method.

The use of embedded JS is in practice limited , mostly due to maintenance and reuse issues. Generally, JS is stored in a separate file and executed from there. So the following could be stored in a file hello.js:

```
document.write("<p>Hello, world!</p>");
```

This file can be referenced then by writing

```
<html>
   <head>
        <title>Say Hello!</title>
   </head>
   <body>
      <p>
         <script type="text/javascript" src="hello.js"></script>
      </p>
   </body>
</html>
```

As mentioned, JS places a lot of emphasis on functions. A JS function looks a bit like a C function, except there are no return or parameter types. A file speak.js can define a function:

```
function speak() {
   document.write("<p>Hello, world!</p>");
}
```

Looks pretty much like C! Now we come to something like the generic “blueprint” of an HTML page that uses JS:

```
<html>
   <head>
      <title>Say Hello!</title>
      <script type="text/javascript" src="speak.js"></script>
   </head>
   <body>
      <script type="text/javascript">speak();</script>
   </body>
</html>
```

The first script (in the head section) loads the function definition(s) from the given JS

file, the second script (in the body section) calls one of these functions. Generally there can be more than one load script and more than one call script, of course.

2.1.1 Variables and typing

As mentioned, typing in JS is dynamic; a variable is declared without a type and accordingly the type of its value can change over time – whether this is something you actually want to use is another question, of course:

```
<html>
  <head>
    <title>Variables 1</title>
  </head>
  <body>
     <script type="text/javascript">
        var x = "I am a string ";
        document.write(x);
        x = 12;
         document.write("<br />and now I'm the number ");
        document.write(x);
     </script>
  </body>
</html>
```

Variables, function definitions and function calls can be mixed as seems fit – functions can actually be called before they are textually defined:

```
<html>
   <head>
      <title>Variables and values</title>
   </head>
   <body>
      <script type="text/javascript">
         var firstname;
         var x = 5;
         var y = 10;
         firstname="Bernd";
         document.write(x+y);
         br();
         document.write(firstname);
         br();
         firstname="OK";
         document.write(firstname);

         function br()
         {
            document.write("<br />");
         }
      </script>
      <p>The script above declares some variables, does arithmetic
               and calls a function</p>
   </body>
</html>
```

Variables that are declare in a lexical context (i. e. inside or outside a function definition) are valid just for this context – in the example glob1 is declared global for all scripts in this page:

```
<html>
   <head>
      <title>Global variables</title>
   </head>
   <body>
      <script type="text/javascript">
         var glob1 = 100;
         foo(12);
         function foo (param) {
            var local = param + glob1;
            document.write(local);
            glob1 = param;
            bar();
         }
         function bar() {
            document.write(" " + glob1);
         }
      </script>
      <script type="text/javascript">bar();</script>
   </body>
</html>
```

This will print “112 12 12”.

2.1.2 Events

Still, there are fine points in the run-time behavior of a JS script when the workings of a browser are taken into account. Most of these problems are centered around the question *when exactly* is the script executed? In the above case it's not really too complicated, but when the body of a page contains many elements that need to be loaded separately, it might be interesting to have more control about the exact timing of the script.

This is done by tying the execution of a script to an *event*, not unlike the events that are defined in a SWING environment:

```
<html>
   <head>
      <title>Say Hello!</title>
      <script type="text/javascript" src="speak.js"></script>
   </head>
   <body onload="speak();">
   </body>
</html>
```

The onload event (attached to the body of an HTML page) occurs *when the page in question is completely loaded*; so, for example, all images are in place, all forms are set

up, etc. This allows to refer to HTML elements from a scripting context regardless of the position of the script in the text – everything is now guaranteed to be loaded when the script is executed.

There are quite a few event types in JS, we will look at some important ones. A list of the JS event types is given at http://www.w3schools.com/jsref/dom_obj_event.asp.

```
<html>
  <head>
    <title>Some events </title>
  </head>
  <body onLoad="alert('Welcome!');" onUnLoad="alert('Bye!');">
      <img src="dom.png" alt="Just an image" width="229" height="459" border="0"
          onMouseOver="alert('An onMouseOver event');return false"
          />
       
      <img src="xampp.png" alt="Just an image" width="454" height="384" border="0"
          onClick="alert('An onClick event');return false"
          />
  </body>
</html>
```

Here are four events with the associated *event handlers*. The only not obvious one it onUnLoad which occurs when the user leaves the page, either by following a hyperlink, manually changing the address or closing the window/tab.

2.1.3 JS and the HTML DOM

Since HTML (even in its non-strict, non-XHTML form) can be considered an XML vocabulary, every HTML document has its own DOM. Admittedly, using XHTML makes the DOM better defined, but even a page that sends your browser into “quirks mode” has such a DOM.

JS has a comprehensive API to access the DOM of the underlying HTML document in order to retrieve elements, create new elements or make modifications to existing elements. This API is so extensive that we can just sketch some of the more frequently used features; for a complete overview, consult http://www.w3schools.com/js/js_ex_dom.asp.

To give an idea of JS DOM programming, let's look at some of the most important objects that constitute the base of the DOM.

The document object supports a couple of methods that provide access to the elements of the current document – we already know about the print method that inserts text (generally HTML code) directly into the page.

document offers a couple of properties that allow direct access to HTML elements –

generally available as arrays: anchors[], forms[], images[] and links[]. Methods to access HTML elements inside the current page use element ids (most common), element names or tag names.

Elements provide properties and methods to access the HTML code, to access style information, event listeners and the like. The following example is the “normal” way of placing a value (in this case obtained by reading it from a prompt) into a specific element, generally a div or a span (referred to as “filling the div”).

```
<html>
   <head>
      <title>Asking for your name</title>
      <script type="text/javascript">
         function ask() {
            var name = prompt("Enter your name","");
            document.getElementById("response").innerHTML = "Hello, " + name;
         }
      </script>
   </head>
   <body onload="ask();">
      <div id="response"></div>
   </body>
</html>
```

Event listeners are generally implemented as parameterless functions; in best “functional” style they're used as *function literals* (or constants). This is an example of such a function literal:[3]

```
function () {
  alert("Hello, world");
}
```

When this function is called (how to do that we'll see in a moment!), it will display an alert box with the given text in it.

Calling such an “anonymous” function is accomplished by “applying” it to the respective number of actual parameters, either by storing the function as a variable value or directly:

f0 in this example is a variable that holds the parameterless function – calling it is done by writing down the variable name followed by a list of actual parameters:[4]

```
var f0 = function () { alert("Hello, world!"); };
f0();
```

3 I know it's not really a function, since it doesn't return anything. There are no “procedures” in JS, so we'll use this term in a technical rather than a formal sense.

4 Looks like a C function call? That's no coincidence, a named function as in C is equivalent to an anonymous function that's stored on a variable; it's that easy!

Defining and applying a function with one parameter works exactly the same way:

```
var f1 = function (name) { alert("Hello, " + name); };
f1("UWF");
```

It looks even nicer when you omit the variable and apply the function literal directly to the actual parameters – you only have to surround the function with an extra pair of parentheses to “hold it together”:

```
(function () { alert("Hello, world!"); })();
(function (name) { alert("Hello, " + name); })("UWF");
```

Equipped with this knowledge writing event handlers is now pretty easy:

```
<html>
   <head>
      <title>Mouse Warning</title>
      <script type="text/javascript">
         function go() {
            document.getElementById("magic")
            .addEventListener("mouseover", function () { alert("Told you!!!"); }, false);
         }
      </script>
   </head>
   <body onload="go();">
      <div style="width:200" id="magic">Don't come near with the mouse!</div>
   </body>
</html>
```

The next example is intentionally confusing:

```
<html>
  <head>
    <title>More Event Listeners</title>
  </head>
  <body>
    <form action=""
        onsubmit="(function (msg) {
                    var text='Hello, '+msg; alert(text)})
                  (document.getElementById('name').value);">
      Name: <input type="text" id="name" name="name">
      <input type="submit" name="submit">
    </form>
    <a href="http://google.com"
        onClick="function () {window.open('http://google.com');
                 return false}">Google</a>
  </body>
</html>
```

2.2 AJAX

AJAX stands for “Asynchronous JavaScript And XML” and describes a communication technique between a JS client and a server that delivers XML documents. The normal way is sketched below.

The client uses JavaScript to send an http request to the server, requesting a particular XML document. The server receives the request and responds over http with the document (just like an HTML page). The client receives the response and starts working with the document.
There are some smaller problems with this approach: first of all, the server response might come in after a certain delay due to transmission time, server load, etc. The client can a) wait for the response – or a time-out (synchronous model) or b) continue with other tasks and be notified of the response when it arrives (asynchronous model).

In order to manage the asynchronous model, the client sets up a special handler for changes in the request “state” (readyState). A request can be in one of 5 different states encoded by numbers:[5]

0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready

In state 4 the response can be processed. It's mainly characterized by the http status

5 http://www.w3schools.com/dom/dom_http.asp

code (status). The most common status codes are:[6]

200: OK
403: forbidden
404: not found
500: internal server error[7]

```
<html>
   <head>
      <script type="text/javascript">
         function process()
         {
            var xmlhttp;
             // 1. construct http carrier object
            if (window.XMLHttpRequest)
            { // code for IE7+, Firefox, Chrome, Opera, Safari
               xmlhttp=new XMLHttpRequest();
            }
            else
            { // code for IE6, IE5
               xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
            }
             // 2. provide state change handler
            xmlhttp.onreadystatechange=function()
            {
               if (xmlhttp.readyState==4 && xmlhttp.status==200)
               {
                  // process the document
               }
            }
             // 3. send it out and hope for the best
            xmlhttp.open("GET",documentURL,true);
            xmlhttp.send();
         }
      </script>
   </head>
   <body>
   ...
   </body>
</html>
```

This little example shows already some of the major issues with “plain” AJAX programming: the implementation of the http request varies (varied?) strongly between browsers. Most of this confusion is more or less gone, but to be worried about “legacy” browsers is still one of the major issues for client-side developers.

---

6 http://en.wikipedia.org/wiki/List_of_HTTP_status_codes
7 Everybody's favorite!

This and a general demand for making CSP a little less complicated lead to the development of “JavaScript libraries”, collections of higher-level functions that embody a lot of those pesky details that makes life so difficult for developers.

A lot of JS libraries are out there and it's mostly a matter of taste (or company policy) which one to use – start your exploration at Wikipedia: http://en.wikipedia.org/wiki/List_of_JavaScript_libraries.

Here we'll deal with jQuery (http://jquery.com/) - more or less a personal preference.

2.3 jQuery

jQuery provides a multitude of functionality combined with a uncommon but powerful notation. It might take a moment to get used to it, but after a while writing in jQuery becomes pretty natural.

2.3.1 jQuery concepts

As usual, let's dive right into it by looking at a moderately complex example:

```
<html>
   <head>
      <title>jQuery First Cnntact</title>
      <script type="text/javascript"
              src="http://ajax.googleapis.com/ajax/libs/jquery/1.6/jquery.min.js">
      </script>
      <script type="text/javascript">
         $(document).ready(function() {
            $("a").css("color", "red");

            $("a.#anchor1").click(function(event) {
               alert("Thanks for clicking!");
               event.preventDefault();
            });
            $("a.#anchor2").click(function(event) {
               $("a.#anchor1").hide();
               event.preventDefault();
            });
         });
      </script>
   </head>
   <body>
      <a id="anchor1" href="#">Click me!</a> 
      <a id="anchor2" href="#">Hide the other anchor!</a>
   </body>
</html>
```

The first (green) script loads the current version of jQuery from a Google server – you can download it locally, but deployment and version maintenance are easier if you load it from a central server.

The syntax for locating an HTML element in jQuery differs from the slightly cumbersome way it's done in plain JS. An expression $(...) selects one or more elements described between the parentheses:

$(document)	is the complete page – the document object in plain JS.
$("name")	are all the elements with the given (tag) name
$("#id")	is the element with the given id
$(s1.s2.s3...)	cascades selectors starting with s1 and then down

So, for example, $("a.#anchor2") is the anchor with the id anchor2.[8]

In jQuery, the document object provides methods that aren't available in plain JS – especially a set of events usually associated with the HTML body element. Most prominently, the ready method takes a parameterless function that is executed when the document is completely loaded; it replaces the onload event in HTML – just don't use both together!

The css method on an element adds a new CSS rule to this element – in the example all anchors will appear in a red font.

The click method adds an event listener to what was the onclick event in HTML.

The preventDefault method deactivates the standard behavior of an event – so clicking an anchor would normally open the target, in this case nothing will happen.[9]

Another example – demonstrating the capability of creating/accessing HTML DOM elements:

8 Of course, "#anchor2" would have been sufficient here, assuming that ids are unique within the document!

9 In all these examples it's probably a good idea to first figure out what will happen before actually trying them.

```
<html>
   <head>
      ...
      <script type="text/javascript">
         $(document).ready(function() {
            $("<p>Here it goes!</p>").insertAfter("h1");

            $("a.#anchor").click(function(event) {
               $("#main").html("No idea what that means!")
               event.preventDefault();
            });
         });
      </script>
   </head>
   <body>
      <h1>Welcome to part 2</h1>
      <p id="main">
         Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
         do eiusmod tempor incididunt ut labore et dolore magna aliqua.
      </p>
      <a id="anchor" href="#">Explain!</a>
   </body>
</html>
```

Not a lot new here, just the expression $(" Here it goes! ") needs some explanation. A textual HTML target simply creates the corresponding element, so this here creates a p element with the respective inner HTML code “Here it goes!”. The html method corresponds to the modification of the innerHTML attribute, i. e. it modifies the content of the element.

The next example shows more of the targeting capabilities of jQuery:

```
<html>
   <head>
      ...
      <script type="text/javascript">
         $(document).ready(function(){
            $("li:contains('SciFi')").css("color", "red");
         });
      </script>
   </head>
   <body>
      <h1>Welcome to part 3</h1>
      <h3>Movies</h3>
      <ul>
         <li>Star Trek 1 (SciFi)</li>
         <li>Speed 1 (Action)</li>
```

```
        <li>Alien (SciFi)</li>
        <li>Harry Potter 1 (Fantasy)</li>
      </ul>
   </body>
</html>
```

One more:

```
<html>
   <head>
      ...
      <script type="text/javascript">
         $(document).ready(function() {
            $("a.#anchor1").click(function(event) {
               $("li:contains('SciFi')")
                  .append(" <strong>Some SciFi flick</strong>")
                  .css("color", "red");
               event.preventDefault();
            });
         });
      </script>
   </head>
   <body>
      <h1>Welcome to part 5</h1>
      <h3>Movies</h3>
      <ul>
         <li>Star Trek 1 (SciFi)</li>
         <li>Speed 1 (Action)</li>
         <li>Alien (SciFi)</li>
         <li>Harry Potter 1 (Fantasy)</li>
      </ul>
      <a id="anchor1" href="#">Click me!</a>
   </body>
</html>
```

Ok, slowly: The anchor with the id anchor1 gets a click event handler, that appends to every list item that contains the substring 'SciFi' the code Some SciFi flick and colors it all red. Easy!

2.3.2 jQuery and AJAX

The main point of jQuery for the purpose of this course is the relatively uncomplicated way AJAX calls can be made. It's more or less one method (suitably called ajax) that deals with everything. The complete description of this method is given at http://api.jquery.com/jQuery.ajax/, here let's study the most important used with some examples.

For this introduction let's take another XML document – a DVD library library.xml– that might make for more decent examples:

```
<library>
   <dvd id="1">
      <title>Breakfast at Tiffany's</title>
      <format>Movie</format>
      <genre>Classic</genre>
   </dvd>
   <dvd id="2">
      <title>Contact</title>
      <format>Movie</format>
      <genre>Science fiction</genre>
   </dvd>
   <dvd id="3">
      <title>Little Britain</title>
      <format>TV Series</format>
      <genre>Comedy</genre>
   </dvd>
</library>
```

The following jQuery code retrieves this file over http GET, selects the first DVD and places its title into the page:

```
<html>
   <head>
    ...
      <script type="text/javascript">
         $(document).ready(function() {
            $.ajax({
               type: "GET",
               url: "library.xml",
               dataType: "xml",
               success:
                  function(xml) {
                    var dvd = $(xml).find('dvd')[0];
                    var title = $(dvd).find('title').text();
                    $('<strong></strong>').html(title).appendTo('#main');
            }
         }); //close $.ajax(
```

```
      }); //close $(
      </script>
   </head>
   <body>
      <h1>Welcome to AJAX 1</h1>
      <div id="main">My favorite movie: </div>
   </body>
</html>
```

The only thing worth an explanation is the success entry – a function that takes one argument (the XML DOM as retrieved from the server). This argument then is used to extract information and place that into the surrounding page.

2.4 Client-side XSLT

Producing XSL transformations on the client can technically be done in two ways:

1. If the browser is known to have a built-in XSLT processor, the browser can do the transformation if a complete XML document will be transformed into a complete HTML page for example.
2. Using JS for this purpose adds more flexibility, especially if the resulting page needs to be built dynamically from different XML and XSLT files.

The first case is easily handled by placing a processing instruction at the beginning of the XML file:

```
<?xml-stylesheet type="text/xsl" href="xyz.xsl"?>
```

When the browser requests the XML file, it will use the style sheet to transform it, before displaying it. This technique works well, but combining it with server-side XML synthesis makes it even more powerful. And – as mentioned – it just works with one XML file and one XSLT file.

In order to perform XSL transformations via JS I personally would use one of the many libraries out there, some with jQuery support some without. Here, I'll use the xslt.js package from http://johannburkard.de/software/xsltjs/. Some example code should make it easy to see how the basic mechanism works:

```
<html>
   <head>
      <title></title>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <link rel="stylesheet" href="libhtml.css" type="text/css" media="screen">
      <script type="text/javascript"
                src="http://ajax.googleapis.com/ajax/libs/jquery/1.5/jquery.min.js">
      </script>
      <script type="text/javascript" src="xslt.pack.js"></script>
      <script type="text/javascript">
        $(document).ready(function() {
           new Transformation().setXml("library.xml")
                     .setXslt("library.xsl").transform("main");
         });
      </script>
   </head>
   <body>
      <div id="main"></div>
   </body>
</html>
```

Here – for reference – the basic code of library.xml again:

```
<library>
   <dvd id="1">
      <title>Breakfast at Tiffany's</title>
      <format>Movie</format>
      <genre>Classic</genre>
   </dvd>
   <dvd id="2">
      <title>Contact</title>
      <format>Movie</format>
      <genre>Science fiction</genre>
   </dvd>
   <dvd id="3">
      <title>Little Britain</title>
      <format>TV Series</format>
      <genre>Comedy</genre>
   </dvd>
</library>
```

The style sheet for the JS version should (of course!) only define a segment of the HTML code, because the whole skeleton is already given in the page containing the JS code:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="html"/>
   <xsl:template match="/">
      <h1>Our DVD offers</h1>
      <table border="0" cellpadding="6" cellspacing="3" width="100%" >
         <thead>
            <tr>
               <td>Title</td>
               <td>Format</td>
               <td>Genre</td>
            </tr>
         </thead>
         <xsl:for-each select="library/dvd">
            <xsl:sort select="title" />
            <tr>
               <td class="title">
                  <xsl:value-of select="title" />
               </td>
               <td>
                  <xsl:value-of select="format" />
               </td>
               <td>
                  <xsl:value-of select="genre" />
               </td>
            </tr>
         </xsl:for-each>
      </table>
   </xsl:template>
</xsl:stylesheet>
```