Tutorial link on CORS:

http://www.html5rocks.com/en/tutorials/cors/

Asynchronous Javascript and XML (AJAX)

Introduction

- AJAX is a technique for creating fast and dynamic web pages. Asynchronously update PARTS of a webpage by exchanging small amounts of data with the server, without needing to load the entire page.

- Use cases / need:
  - make asynchronous calls to the server without making the user wait
  - refresh parts of a page
  - load web pages in stages
  - refresh a page periodically without waiting for the page to reload

- Major applications:
  - login forms
  - auto-complete
  - chat/instant messaging, etc.

- Advantages:

  (to be completed)

- Disadvantages:

  (to be completed)

- Techniques to simulate asynchronous events:
  1. iframes
  2. images
  3. XMLHttpRequest (XHR) object
  4. <script> tag
  5. <link> tag

Formats of an asynchronous response
---

1. Plain Text

   Advantage(s):
   - simple to use

   Disadvantage(s):
   - not suitable for sending structured data

2. JSON

   Advantage(s):
   - suitable for structured data - directly maps to Javascript object
   - very minimal effort to parse

   Disadvantage(s):
   - loading data from JSON into HTML requires code

3. XML

   Advantage(s):
   - has customizable tags, attaching semantic meaning
     Eg: <empName> Cowboy Tanaka </empName>
   - more data centric
   - easier to load XML data into tables
   - has validators (to ensure that the XML data is well-formed and follows a defined structure)
   - XLST (eXtensible Stylesheet Language) allows XML to quickly be converted to HTML given a set of rules

   Disadvantage(s):
   - If data needs to be processed, it needs code to unpack data

4. Binary data

   Used mostly for sending binary data.

   Prerequisite: before sending the asynchronous request, set the xhr.responseType property to "blob", and then extract the response from xhr.response.

5. HTML

Not recommended since it is not data centric, and has a lot of superfluous information

Conclusion: Use the response format that best fits your application.

Note: 1) xhr response variables -
xhr .responseText
(plain text, JSON and HTML)
.responseXML
(XML only)
.response
(binary data like images and videos)

2) Code to use when working with JSON:
(i) on the server side:

```
<?php
    ....
    echo json_encode(object);

    ?>
```

(ii) on the client side:

```
function processResults()
{
    if(xhr.readyState == 4 && xhr.status == 200)
    {
        responseObj = JSON.parse(xhr.response);
        // now responseObj is in the form of a JS object, you can do what you want with it
    }
}
```

Asynchronous requests with iframes

How it works:

1. There are two frames, one hidden and one visible. (Actually, the visible frame can be any markup like an input element or a div).

2. When an event is triggered on the visible frame, the source of the hidden frame is changed to point to a server source

3. A change in the source forces a request to the server. Data is then received from the server by the hidden frame.

Advantages:

- supports both GET and POST requests
- can store browser history

Disadvantages:

- cannot support cross domain (or even sub-domain) requests
- no error handling available for unsuccessful requests

How to issue an asynchronous GET request:

index.html

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">

            function sendRequest()
            {
                ifr = document.getElementById("hf");
                username = ____._________("user").value;

                ifr.src = "processRequest.php?user=" + username;
            }

            function processResponse(status)
            {
                if(status)
                    //      username not in use
```

```
                                        else if(!status)
                                                //      username in use
                                }

                        </script>
                </head>

                <body>
                        <form method="..." action="submit.php">
                                <input type="text" id="user" name="user"
onblur="sendRequest()"/>
                        </form>

                        <iframe id="hf" style="display:none;">
                </body>
        </html>

                processRequest.php
                --------------------------
<?php
        header("Content-type:text/html");
        extract($_GET);

        if(!isset($user)) die("...");

        $status = 1; // if status is 1, then the username isn't in use

        /* get usernames from some database, let the names reside in an array called
$usernames */

        foreach($usernames as $k => $v)
        {
                if($v == $user)
                {
                        status = !status;
                        break;
                }
        }

        echo '<html><head><script>parent.processResults(' . $status .
');</script></head></html';
?>
```

Summary:

- have an iframe
- have a function that is called when some event (like onblur) fires on some element
- in that function, set the iframe source to point to some php script
- in the php script mentioned above, do some basic validation and then echo markup which calls another function. Since the iframe has a parent-child relationship with the web page, use the syntax parent.function(parameters) (look at the php script for details).
- when the response comes back, that function is invoked.

How to issue an asynchronous POST request:

Sending async POST request has a different use case. For example, with a GET request, you'd want to perform validation. With POST requests, you can do async form submission, as so.

index.html

```
<!DOCTYPE html>
<html>
    ...
    <body>
        <form method="POST" target="hf" action="submit.php">
            <input type="text" id="user" name="user" />
            ... <!-- other fields go here -->
            <input type="submit" value="Register" />
        </form>

        <iframe id="hf" name="hf" style="display:none;">
    </body>
</html>
```

submit.php

```
<?php
    header("Content-type:text/html");
    extract($_POST);

    ...     // all variables from the form will be available
?>
```

Summary:

- make sure the 'name' attribute of the iframe is set, and then set the 'target' attribute of the form to the 'name' attribute of the ifra
- in the php script, extract $_POST and then perform your operations on the data

Do note that we're sending form data asynchronously using POST, not performing validation on the username field as the example in GET request showed.

Asynchronous requests with images

Suitable technique when boolean response is needed from the server side. At the front-end a placeholder for an image is created with the <img> tag and its source is made to point to a script file which should return the source to an image or the image itself. The width/height of the image is the boolean response required.

How it works:

1. Create the image on the front-end side

```
image = document.createElement("img");
```

2. Set the callback function for onload and onerror events on the image

```
image.onload = processResult;
image.onerror = processError;          // optional
```

3. Set the image source

```
image.src = "script.php?param=" + someParameterHere;
```

4. On the server side... either

(i) Redirect to an image:

```
<?php
        header("Content-type:image/jpeg");
        header("Location:image.jpeg");
?>
```

OR

(ii) Create the image, allocate a colour and place it into the output stream:

```
<?php
        header("Content-type:image/jpeg"); // png also
works
        $image = imagereate(width, height);  // if you want
to return positive result, set width to 1. Else, set width to > 1.

        imagecolorallocate($image, 255, 255, 255); //
allocate colour

        imagejpeg($image); // place into the output stream

        imagedestroy($image); // free up memory

?>
```

5. Back again on the client side, now when the response comes back, either onload or onerror should fire. The corresponding functions are invoked.

If there was no error, then the processResults() function is invoked. Inside this function, sample image.width attribute to check if it is 1 (for success) or 2 (for failure)

Note: Textual data can be sent through cookies. On the server side, use the setcookie(key, value) function to set cookies, and in the front-end, access cookies using the document.cookie property.

```
<?php
    header(....);
    ob_start();
    extract($_GET);
    if($user == .. || ...)
    {
        … // Taken

        setcookie("Status", "0");
    }
    else
    {
        … // Free
        setcookie("Status", "1");
    }
    ...
    ob_flush();
?>
```

Advantages:

- can handle cross domain requests
- has decent enough error handling support

Disadvantages:

- can only support GET requests
- cannot store history
- textual data can only be retrieved through cookies which is limiting as well as dangerous
- will not work when images/cookies are disabled

Note: Sending async requests with script tags is done similar to images. The tag is programmatically created and the "src" attribute is pointed to a php file that returns an arbitrary amount of Javascript. Its advantages and disadvantages are similar to using images, with the difference that the image tag does not need to be added to the dom for the asynchronous call to go through.

Asynchronous requests with XMLHttpRequest (XHR) objects

How it works:

1. Create an XMLHttpRequest object

```
xhr = new XMLHttpRequest(); // XDomainRequest() for IE
```

(similar to XHR but with additional security features)

2. Specify a function to execute when the response is ready using the onreadystatechange event:

```
xhr.onreadystatechange = function()
    {
        if(this.readyState == 4 && this.status == 200)
        {
            // request was a success, now process the data
        }
    }
```

- onreadystatechange is the event that holds the function (or a reference to the function) to be executed each time the readyState property changes

- readyState holds the status of the XHR object. It can take the values 0 (not initialised), 1 (connection established), 2 (request received), 3 (processing request), or 4 (finished).

- status tells us whether the request was successful or not (status is 200 is successful, anything else otherwise).

3. Open a new request to the server

a) xhr.open(__mthd__, __url__, true);

(i) __mthd__ specifies the type of request, is mostly "GET" or "POST"
(ii) __url__ is the server/file location
(iii) true implies the request must be asynchronous

b) OPTIONAL, involves setting headers using xhr.setRequestHeader("...", "...")
Eg: Before sending form data in a POST request, set the request header appropriately:

xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

4. Send the request to the server

xhr.send(); // for GET
or
xhr.send(string) // for POST

Note: which method is better?
GET is simpler and faster than POST, it is used for sending small amounts of data. GET may also cache results, so it is desirable in many cases.

POST is better for sending larger amounts of data as there are no size limitations, and it is more robust and secure than GET.

5. Process the results (inside the onreadystatechange callback function). The results will be inside one of

(i) xhr.responseText - if HTML/text/JSON, or
(ii) xhr.responseXML - for XML data (can be processed like a dom object)

Advantages:

- both GET and POST supported
- more robust and secure (appropriate error handling mechanism with onerror)
- cross-domain requests supported with Access-Control-* headers

Disadvantage:

- cannot save history (needs to be implemented using window.onpopstate and window.pushState())

Note: One small issue with XHR is the chance that it may reuse cached data. This can be controlled on the server side by setting the header "Cache-control" to "no-cache" (disallows caching) or "no-store" (invalidates the current cache).

Uploading files with XHR

Uploading files with AJAX can easily be done using the FormData API.

1. First, initialise the FormData() object, as so:

```
obj = new FormData();
```

2. Use the append() function to add key-value pairs to the object:

```
obj.append("label", file, "filename");   // the third argument is optional, and when omitted, will take on the default value, that is the actual name of the file */
```

If the data is a file, it'll be accessible in the $_FILES superglobal. Other data will be available when you extract $_POST.

3. Open and send an XHR POST request:

```
xhr.open("POST", "....", true);
```

```
                xhr.send(obj);
```

Example:

index.html

```
----------
    ...
    <form>
        <input id="cv" type="file" />
        <input type="submit" onclick="uploadCV()">
    </form>

    <script type="...">

        function
    </script>

        function uploadCV()
        {
            fileInput = document.getElementById("cv");
            file = fileInput.files[0]; // grab the file from a "list" of files

            obj = new FormData();
            obj.append("mycv", file);
            obj.append("...", "..."); // some other metadata if you want

            xhr.onreadystatechange = function()
            {

            }

            xhr.open("POST", "processRequest.php", true);
            xhr.send(obj);
        }
    </script>
```

processRequest.php

```
-------------------------
    <?php
        extract($_POST); // you will find

        echo $_FILES['mycv']['name'];
```

Maintaining History with XHR

By default, XHR objects do not maintain history. But we can use some properties of the window object to support it:

1. window.onpopstate event, and
2. window.pushState() function

Example:

```
<script type="text/javascript">
	window.onpopstate = restoreState();
	nextCount = 0;

	obj =
	{
		xhr: new XMLHttpRequest(),

		getDetails	:	function()
		{
			this.xhr.onreadystatechange = this.processResults;
			this.xhr.open("GET", "...", true);
			this.xhr.send();
		}

		processResults	:	function()
		{
			/* we use this.readyState, and not this.xhr.readyState since the function was registered on xhr, not obj
			if(this.readyState == 4 && this.status == 200)
			{
				... // access this.responseText to get some values, say v1, v2, and v3 that you want to save

				saveState = new Object();
				saveState.v1 = v1;
				saveState.v2 = v2;
				saveState.v3 = v3;

				// Now we can save the state of this object
				// format: .pushState(objectToBePushed, "LABEL", "SOME URL")
```

```
                                        window.history.pushState(saveState, "LABEL",
"http://localhost/myProgram?state=" + nextState++);

                                        xhr.abort();
                                    }
                                }
                            }

                            function restoreState(event)
                            {
                                saveState = event.state; // assigns the state of the event to an
object

                                ...     /* now we can access saveState     .v1
                                                                           .v2
                                                                           .v3
                                        and do whatever we want with it */
                            }
                        </script>
```

Cross-Origin-Resource-Sharing (CORS)

(to be completed)

Really Simple Syndication (RSS)

Introduction

- Really Simple Syndication / Rich Site Summary is a format for delivering regularly changing web content.

- Need for RSS: For situations where the server regularly publishes a lot of data frequently, users must have a mechanism wherein they can fetch this data without manually having to visit the site. RSS aggregators are programs that grab news feeds from servers, sort it out, and display it to users.

- To check whether an HTML website has an RSS equivalent, there are two indicators:
    1. <link rel="alternate" href="..." type="rss+xml" />
    2. The RSS logo

- A little history:
    - Initiated by Microsoft as Content Definition Format (CDF), an XML based format for defining updates to a resource. However, the format was complicated and news aggregators started failing.

    - Netscape began RDF (Resource Description Framework) Site Summary (RSS), soon after handed over to Harvard

    - Now known as Really Simple Syndication (RSS). Current version is 2.1.

- How RSS works:
    1. A web server publishes data using RSS as a newsfeed. This can be done manually or through software. This data needs to be validated to ensure it is in proper format.

    2. A vendor will subscribe to the newsfeed.

    3. A feed reader will read the feed and display the news to the vendor. Only the latest data is gathered and displayed. Vendors can customize the content they see.

- Advantages of RSS:
    For the user:
        - all news is in one place
        - freedom from email spamming

    For the server:
        - much easier to publish data (server need not maintain a database of subscribers)
        - the writing process is much simpler

- Some common applications:
    - news
    - entertainment
    - job postings
    - auctions
    - realtors, etc

Structure of RSS

```
<?xml version="1.0" encoding="UTF 8" ?>
<rss version="2.0"> -------- (i)
```

```
<channel>      ---- (ii)
    <title>          ...      </title> ----------\
    <link>           ...      </link> -------------------(iii)
    <description>    ...      </description> ---/
    <ttl>            ...      </ttl>

    <item>   --------- (iv)
        <title>          ...      </title>   ----------\
        <link>           ...      </link>  ---------- (v)
        <description>    ...      </description> --/

    </item>

    <item>
        ...
    </item>

    ...

</channel>
</rss>
```

(i) <rss> is the root node of the document

(ii) <channel> tag has no purpose. It's only Microsoft's legacy.

(iii) <title>, <link> and <description> tags are the required child elements of the <channel> tag

(iv) <item> describes a news item in an RSS feed. An RSS feed may have 1 or more items.

(v) Each <item> also as the 3 tags for describing the item (but these are not compulsory for the item.

Note:

- all tags are case sensitive
- all opening tags must have a corresponding closing tag

Structure of Atom

Atom is a syndication format similar to RSS, but has stricter rules. RSS does not specify how developers handle HTML markup in their elements. This is specified by atom. You can choose the content type of an element and also specify via attributes how it should be handled.

```
<?xml version="1.0" encoding="UTF 8" ?>

<feed version="1.0" xmlns="http://www.w3.org/2005/atom"> -------- (i)
    <title>          ...          </title>
    <link>                ...          </link>
    <description>  ...          </description>
    <ttl>                 ...          </ttl>

    <entry>   --------- (ii)
        <title>               ...          </title>
        <link>                ...          </link>
        <description>  ...          </description>

    </entry>

    <entry>
        ...
    </entry>

    ...
</feed>
```

(i) The root node is <feed> instead of <rss>. The namespace should reside in http://www.w3.org/2005/atom otherwise parsers may not be able to parse it.

(ii) The <item> field in RSS is replaced by <entry> in Atom.

Comparing RSS and Atom

- <rss> vs <feed> as the root node
- <item> vs <entry> for each individual news entry in the news feed
- no <channel> tag in Atom newsfeeds

- <title>, <link>, <description> tags are all optional for RSS items. However, these are compulsory for Atom entries, along with more semantic tags.

- RSS has 2 protocols (blogger and metaweblog) and there are compatibility issues between the two, whereas Atom has a single unified publishing protocol.

- Atom has strict rules regarding the format of the document (text? HTML?)

Third Party Feed Readers - xparser.js

Using the xparser.js library, RSS and Atom newsfeeds can easily be parsed.

The class hierarchy is as follows:

BaseFeed
- RssFeed
- AtomFeed

BaseItem
- RssItem
- AtomItem

The most important function to use is:

```
xparser.getFeed(url, callback[, callbackScope])
```

where,

url -
the url of the newsfeed

callback -
the entire XML is JSONified and passed to the callback function as an argument

callbackScope -
the object on which the callback function is called

Fetching and Parsing RSS Feeds

(to be completed)

Bootstrap

Introduction

- Developed by Twitter in 2011. Current version is 3.

- A powerful front-end framework which makes web development faster and easier. Uses a combination of HTML, CSS, and Javascript.

- Bootstrap follows a "mobile-first" approach

- Major advantages of Bootstrap:
    1. Responsive design - meaning apps automatically adjust themselves to look good on all browsers
    2. Consistency across all browsers
    3. Open source and hence,
    4. Customizable

- Bootstrap can be applied in two ways, either by using (i) CSS, or (ii) Javascript. This uses JQuery.

Grid System

In the grid system, each row is divided into 12 columns. Elements in a row can span any number of columns, but the total must add up to 12. Gutter width is 15px on either side of the column (so 30px total gutter width).

The classes key to the responsiveness in the grid system are:

1. col-xs-* (< 768px)
2. col-sm-* (768 - 990px)
3. col-md-* (990 - 1200px)
4. col-lg-* (> 1200px)

Where * specifies the number of columns spanned by the element.

Example:

```
<div class="container">
    <div class="col-md-6"> ... </div>
    <div class="col-md-6"> ... </div>
</div>
```

This applies for all devices having xs screen size OR more.

There are two types of container classes:

1. container

   specifies that the layout will have fixed width for a given screen size range

2. container-fluid

   specifies that the layout will stretch to meet the screen size of the current device