New Tricks in XMLHttpRequest2

html5rocks.com/en/tutorials/file/xhr2

One of the unsung heros in the HTML5 universe is `XMLHttpRequest`. Strictly speaking XHR2 isn't HTML5. However, it's part of the incremental improvements browser vendors are making to the core platform. I'm including XHR2 in our new bag of goodies because it plays such an integral part in today's complex web apps.

Turns out our old friend got a huge makeover but many folks are unaware of its new features. XMLHttpRequest Level 2 introduces a slew of new capabilities which put an end to crazy hacks in our web apps; things like cross-origin requests, uploading progress events, and support for uploading/downloading binary data. These allow AJAX to work in concert with many of the bleeding edge HTML5 APIs such as File System API, Web Audio API, and WebGL.

This tutorial highlights some of the new features in `XMLHttpRequest`, especially those that can be used for working with files.

Fetching data

Fetching a file as a binary blob has been painful with XHR. Technically, it wasn't even possible. One trick that has been well documented involves overriding the mime type with a user-defined charset as seen below.

The old way to fetch an image:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);

// Hack to pass bytes through unprocessed.
xhr.overrideMimeType('text/plain; charset=x-user-defined');

xhr.onreadystatechange = function(e) {
  if (this.readyState == 4 && this.status == 200) {
    var binStr = this.responseText;
    for (var i = 0, len = binStr.length; i < len; ++i) {
      var c = binStr.charCodeAt(i);
      //String.fromCharCode(c & 0xff);
      var byte = c & 0xff;  // byte at offset i
    }
  }
};

xhr.send();
```

While this works, what you actually get back in the `responseText` is not a binary blob. It is a binary string representing the image file. We're tricking the server into passing the data back, unprocessed. Even though this little gem works, I'm going to call it black magic and advise against it. Anytime you resort to character code hacks and string manipulation for coercing data into a desirable format, that's a problem.

Specifying a response format

In the previous example, we downloaded the image as a binary "file" by overriding the server's mime type and processing the response text as a binary string. Instead, let's leverage `XMLHttpRequest`'s new `responseType` and `response` properties to inform the browser what format we want the data returned as.

xhr.responseType

Before sending a request, set the `xhr.responseType` to "text", "arraybuffer", "blob", or "document", depending on your data needs. Note, setting `xhr.responseType = ''` (or omitting) will default the response to "text".

xhr.response

After a successful request, the xhr's response property will contain the requested data as a `DOMString`, `ArrayBuffer`, `Blob`, or `Document` (depending on what was set for `responseType`.)

With this new awesomeness, we can rework the previous example, but this time, fetch the image as an `Blob` instead of a string:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'blob';

xhr.onload = function(e) {
  if (this.status == 200) {
    // Note: .response instead of .responseText
    var blob = new Blob([this.response], {type: 'image/png'});
    ...
  }
};

xhr.send();
```

Much nicer!

ArrayBuffer responses

An ArrayBuffer is a generic fixed-length container for binary data. They are super handy if you need a generalized buffer of raw data, but the real power behind these guys is that you can create "views" of the underlying data using JavaScript typed arrays. In fact, multiple views can be created from a single `ArrayBuffer` source. For example, you could create an 8-bit integer array that shares the same `ArrayBuffer` as an existing 32-bit integer array from the same data. The underlying data remains the same, we just create different representations of it.

As an example, the following fetches our same image as an `ArrayBuffer`, but this time, creates an unsigned 8-bit integer array from that data buffer:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'arraybuffer';

xhr.onload = function(e) {
  var uInt8Array = new Uint8Array(this.response); // this.response ==
uInt8Array.buffer
  // var byte3 = uInt8Array[4]; // byte at offset 4
  ...
};

xhr.send();
```

Blob responses

If you want to work directly with a Blob and/or don't need to manipulate any of the file's bytes, use `xhr.responseType='blob'`:

```
window.URL = window.URL || window.webkitURL;  // Take care of vendor prefixes.

var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'blob';

xhr.onload = function(e) {
  if (this.status == 200) {
    var blob = this.response;

    var img = document.createElement('img');
    img.onload = function(e) {
      window.URL.revokeObjectURL(img.src); // Clean up after yourself.
    };
    img.src = window.URL.createObjectURL(blob);
    document.body.appendChild(img);
    ...
  }
};

xhr.send();
```

A `Blob` can be used in a number of places, including saving it to indexedDB, writing it to the HTML5 File System, or creating an Blob URL, as seen in this example.

Sending data

Being able to download data in different formats is great, but it doesn't get us anywhere if we can't send these rich formats back to home base (the server). `XMLHttpRequest` has limited us to sending `DOMString` or `Document` (XML) data for some time. Not anymore. A revamped `send()` method has been overridden to accept any of the following types: `DOMString`, `Document`, `FormData`, `Blob`, `File`, `ArrayBuffer`. The examples in the rest of this section demonstrate sending data using each type.

Sending string data: xhr.send(DOMString)

```
function sendText(txt) {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/server', true);
  xhr.onload = function(e) {
    if (this.status == 200) {
      console.log(this.responseText);
    }
  };

  xhr.send(txt);
}

sendText('test string');
```

```
function sendTextNew(txt) {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/server', true);
  xhr.responseType = 'text';
  xhr.onload = function(e) {
    if (this.status == 200) {
      console.log(this.response);
    }
  };
  xhr.send(txt);
}

sendTextNew('test string');
```

There's nothing new here, though the right snippet is slightly different. It sets `responseType='text'` for comparison. Again, omitting that line yields the same results.

Submitting forms: xhr.send(FormData)

Many people are probably accustomed to using jQuery plugins or other libraries to handle AJAX form submissions. Instead, we can use FormData, another new data type conceived for XHR2. `FormData` is convenient for creating an HTML `<form>` on-the-fly, in JavaScript. That form can then be submitted using AJAX:

```
function sendForm() {
  var formData = new FormData();
  formData.append('username', 'johndoe');
  formData.append('id', 123456);

  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/server', true);
  xhr.onload = function(e) { ... };

  xhr.send(formData);
}
```

Essentially, we're just dynamically creating a `<form>` and tacking on `<input>` values to it by calling the append method.

Of course, you don't need to create a `<form>` from scratch. `FormData` objects can be initialized from and existing `HTMLFormElement` on the page. For example:

```
<form id="myform" name="myform" action="/server">
  <input type="text" name="username" value="johndoe">
  <input type="number" name="id" value="123456">
  <input type="submit" onclick="return sendForm(this.form);">
</form>
```

```
function sendForm(form) {
  var formData = new FormData(form);

  formData.append('secret_token', '1234567890'); // Append extra data before
send.

  var xhr = new XMLHttpRequest();
  xhr.open('POST', form.action, true);
  xhr.onload = function(e) { ... };

  xhr.send(formData);

  return false; // Prevent page from submitting.
}
```

An HTML form can include file uploads (e.g. `<input type="file">`) and `FormData` can handle that too. Simply append the file(s) and the browser will construct a `multipart/form-data` request when `send()` is called:

```
function uploadFiles(url, files) {
  var formData = new FormData();

  for (var i = 0, file; file = files[i]; ++i) {
    formData.append(file.name, file);
  }

  var xhr = new XMLHttpRequest();
  xhr.open('POST', url, true);
  xhr.onload = function(e) { ... };

  xhr.send(formData);  // multipart/form-data
}

document.querySelector('input[type="file"]').addEventListener('change', function(e)
{
  uploadFiles('/server', this.files);
}, false);
```

Uploading a file or blob: xhr.send(Blob)

We can also send `File` or `Blob` data using XHR. Keep in mind all `File`s are `Blob`s, so either works here.

This example creates a new text file from scratch using the `Blob()` constructor and uploads that `Blob` to the server. The code also sets up a handler to inform the user of the upload's progress:

```
<progress min="0" max="100" value="0">0% complete</progress>

function upload(blobOrFile) {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/server', true);
  xhr.onload = function(e) { ... };

  // Listen to the upload progress.
  var progressBar = document.querySelector('progress');
  xhr.upload.onprogress = function(e) {
    if (e.lengthComputable) {
      progressBar.value = (e.loaded / e.total) * 100;
      progressBar.textContent = progressBar.value; // Fallback for unsupported
browsers.
    }
  };

  xhr.send(blobOrFile);
}

upload(new Blob(['hello world'], {type: 'text/plain'}));
```

Uploading a chunk of bytes: xhr.send(ArrayBuffer)

Last but not least, we can send `ArrayBuffer`s as the XHR's payload.

```
function sendArrayBuffer() {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/server', true);
  xhr.onload = function(e) { ... };

  var uInt8Array = new Uint8Array([1, 2, 3]);

  xhr.send(uInt8Array.buffer);
}
```

Cross Origin Resource Sharing (CORS)

CORS allows web applications on one domain to make cross domain AJAX requests to another domain. It's dead simple to enable, only requiring a single response header to be sent by the server.

Enabling CORS requests

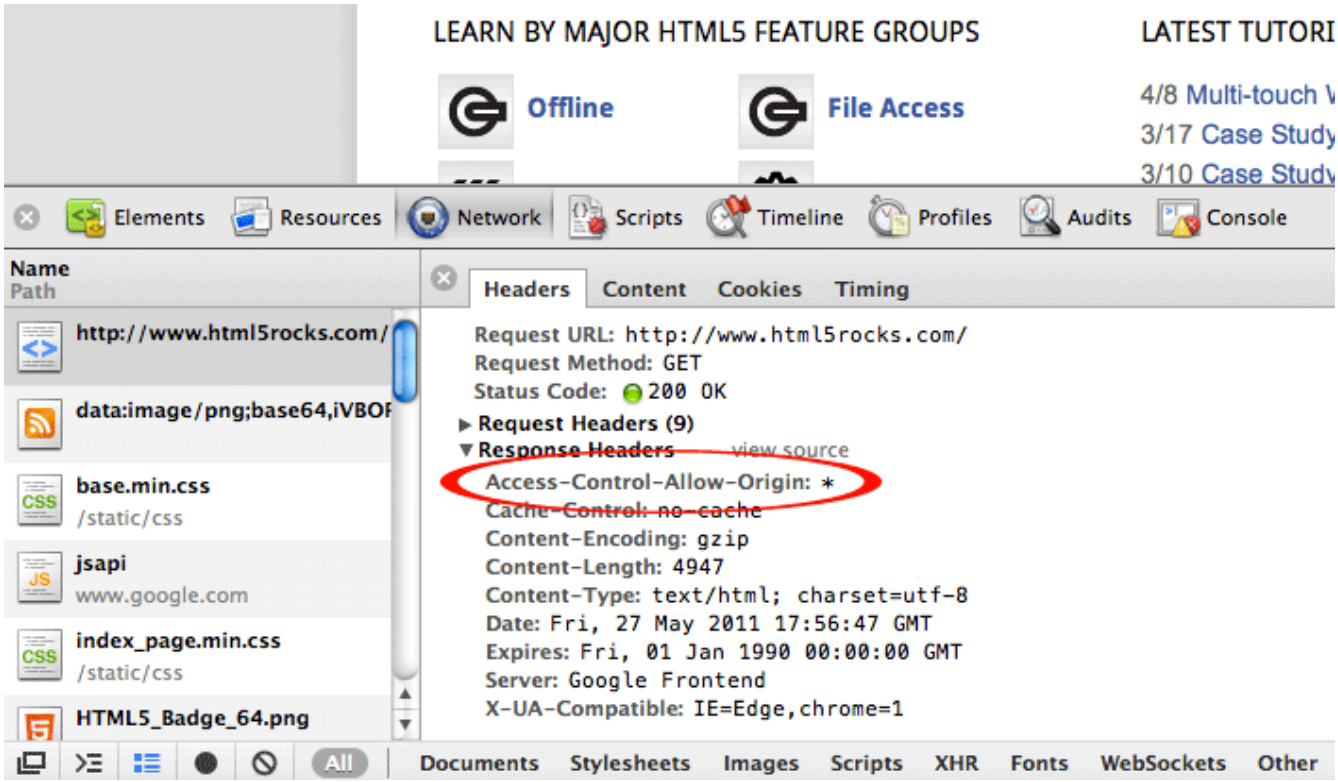
Let's say your application lives on `example.com` and you want to pull data from `www.example2.com`. Normally if you tried to make this type of AJAX call, the request would fail and the browser would throw an origin mismatch error. With CORS, `www.example2.com` can choose to allow requests from `example.com` by simply adding a header:

```
Access-Control-Allow-Origin: http://example.com
```

`Access-Control-Allow-Origin` can be added to a single resource under a site or across the entire domain. To allow *any* domain to make a request to you, set:

```
Access-Control-Allow-Origin: *
```

In fact, this site (html5rocks.com) has enabled CORS on all of its pages. Fire up the Developer Tools and you'll see the `Access-Control-Allow-Origin` in our response:



`Access-Control-Allow-Origin` header on html5rocks.com

Enabling cross-origin requests is easy, so please, please, please enable CORS if your data is public!

Making a cross-domain request

If the server endpoint has enabled CORS, making the cross-origin request is no different than a normal `XMLHttpRequest` request. For example, here is a request `example.com` can now make to `www.example2.com`:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://www.example2.com/hello.json');
xhr.onload = function(e) {
  var data = JSON.parse(this.response);
  ...
}
xhr.send();
```

Practical examples

Download + save files to the HTML5 file system

Let's say you have an image gallery and want to fetch a bunch of images then save them locally using the HTML5 File System. One way to accomplish this would be to request images as `Blob`s and write them out using `FileWriter`:

```
window.requestFileSystem  = window.requestFileSystem ||
window.webkitRequestFileSystem;

function onError(e) {
  console.log('Error', e);
}

var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'blob';

xhr.onload = function(e) {

  window.requestFileSystem(TEMPORARY, 1024 * 1024, function(fs) {
    fs.root.getFile('image.png', {create: true}, function(fileEntry) {
      fileEntry.createWriter(function(writer) {

        writer.onwrite = function(e) { ... };
        writer.onerror = function(e) { ... };

        var blob = new Blob([xhr.response], {type: 'image/png'});

        writer.write(blob);

      }, onError);
    }, onError);
  }, onError);
};

xhr.send();
```

Note: to use this code, see browser support & storage limitations in the "Exploring the FileSystem APIs" tutorial.

Slicing a file and uploading each portion

Using the File APIs, we can minimize the work to upload a large file. The technique is to slice the upload into multiple chunks, spawn an XHR for each portion, and put the file together on the server. This is similar to how GMail uploads large attachments so quickly. Such a technique could also be used to get around Google App Engine's 32MB http request limit.

```
function upload(blobOrFile) {
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/server', true);
  xhr.onload = function(e) { ... };
  xhr.send(blobOrFile);
}

document.querySelector('input[type="file"]').addEventListener('change', function(e)
{
  var blob = this.files[0];

  const BYTES_PER_CHUNK = 1024 * 1024; // 1MB chunk sizes.
  const SIZE = blob.size;

  var start = 0;
  var end = BYTES_PER_CHUNK;

  while(start < SIZE) {
    upload(blob.slice(start, end));

    start = end;
    end = start + BYTES_PER_CHUNK;
  }
}, false);

})();
```

What is not shown here is the code to reconstruct the file on the server.