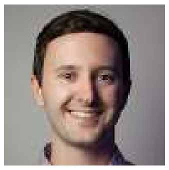
New Tricks in XMLHttpRequest2



By Eric Bidelman

Published: May 27th, 2011
Comments: 4

Introduction

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)