Ridiculously simple Ajax uploads with FormData

Pablo Brasero – January 10, 2012 UPDATED ON March 25, 2019

JAVASCRIPT, NEW BAMBOO, WEB

This post was originally published on the New Bamboo blog, before New Bamboo joined thoughtbot in London.

31/July/2014

Added the *Android inconsistencies* section

Introduction

Back in June 2010, I published a blog post detailing how to perform Ajax file uploads from your HTML forms. The result worked pretty well, but there was still some room for browser vendors to make things even simpler for us.

Turns out, they have. The specific improvement that made this possible is the `FormData` interface, first introduced in Safari 5, and later in Chrome 7 and Firefox 4.

Compatibility and detection

As usual, this doesn't really work across all browsers in the market (you know the usual suspects). Also as usual, the best way to know if the client supports this feature is object

detection. I have updated the function we used in 2010, adding a check for the `FormData` interface:

```
function supportAjaxUploadWithProgress() {
  return supportFileAPI() &amp;amp;&amp;amp; supportAjaxUploadProgressEvents()

  function supportFileAPI() {
    var fi = document.createElement('INPUT');
    fi.type = 'file';
    return 'files' in fi;
  };

  function supportAjaxUploadProgressEvents() {
    var xhr = new XMLHttpRequest();
    return !! (xhr &amp;amp;&amp;amp; ('upload' in xhr) &amp;amp;&amp;amp; ('onprogres
  };

  function supportFormData() {
    return !! window.FormData;
  }
}
```

If the function returns true, you are good to go. If not, it may still be that the old technique can still be used (although chances are slim). If neither is, you will have to resort to some other way to do this, such as Flash (argh!) or traditional form submission.

Submitting the whole form

I won't dwell in details such as the events, as they haven't changed since the last time. What has changed is what the `XMLHttpRequest.send()` call receives. Last year, we fed it an instance of the `File` interface. This time instead, we will give it an instance of `FormData`.

`FormData` gives us two ways to interface with it. The first and simplest is: get a reference to the `form` element and pass it to the `FormData` constructor, like so:

```
var form = document.getElementById('form-id');
var formData = new FormData(form);
```

This new `FormData` instance is all you need to pass on the `send()` call:

```
var xhr = new XMLHttpRequest();
// Add any event handlers here...
xhr.open('POST', '/upload/path', true);
xhr.send(formData);
```

This will send an Ajax request with all the fields of the form on it, not only file inputs. If there were also text areas, text fields, checkboxes or what have you, they'll be sent too. Any events that you may be listening to will be called, such as `onprogress` or `onreadystatechange`.

Submitting only the file

The solution proposed above, albeit useful, is a bit limiting in that we are forced to submit the whole form. Instead, we may want to submit the file independently of the rest of the form. This is common nowadays in places such as bulk photo uploaders on many social networks.

Fortunately `FormData` also allows us to do it this way. For this, it provides the method `append(name, value)`:

```
<!-- The HTML -->
<input id="the-file" name="file" type="file">
```

```
// The Javascript
var fileInput = document.getElementById('the-file');
var file = fileInput.files[0];
var formData = new FormData();
formData.append('file', file);
```

And now the `formData` object is ready to be sent using `XMLHttpRequest.send()` as in the previous example.

Progressive enhancement

One immediate advantage over the old method: the data received on the server side will be indistinguishable from a normal form submission. With the old method, we had to add specific code on the server side to handle the incoming data in a specialised way. This is not necessary anymore, and removed code is debugged code!

There is another great advantage: it is easier to add progressive enhancement. We can have a perfectly normal form, and add this Ajax on top. This way, if the necessary JS interfaces aren't supported, the form will still work (although not as nicely). It would be something like this:

```
<!-- The HTML -->
<form id="the-form" action="/upload/path" enctype="multipart/form-data'
  <input name="file" type="file">
  <input type="submit" value="Upload" />
</form>
```

```
// The Javascript
var form = document.getElementById('the-form');
form.onsubmit = function() {
  var formData = new FormData(form);

  formData.append('file', file);

  var xhr = new XMLHttpRequest();
  // Add any event handlers here...
  xhr.open('POST', form.getAttribute('action'), true);
  xhr.send(formData);

  return false; // To avoid actual submission of the form
}
```

In the example above, we send the XMLHttpRequest on submit of the form. We also read the `action` attribute of the form to know where to send the request. Of course, it's missing some event handlers that we'll use to update the interface.

The beauty of this example is: if the object detection script returns false, this form will still work, and therefore functionality will remain intact.

Android inconsistencies

But there's still one problem: some Android devices don't play entirely well with this technique. They sort of work, but the **load** and **progress** events won't fire. I have found this problem across versions of the OS and browser apps. After all, remember that there's no such thing as The Android Browser.

This is not the end of the world though. A way to mitigate this problem is to make our webapp work so that it doesn't really matter much if those two events are not fired. Obvious when you think of it:

1. Listen to the **loadstart** event and show some kind of pseudo-progress feedback. For example, an indeterminate progress bar or a spinner.
2. If and when **progress** is triggered, remove this pseudo-progress element and replace it with a real progress bar.
3. Don't rely on the **load** event to know when the upload has finished. Instead, use good old **readystatechange**. However, bear in mind there's a subtle difference in behaviour: **load** will trigger when the upload is finished, without waiting for the response from the server. On the other hand **readystatechange** will wait for a response from the server, with occurs later.

A complete example

But enough of theory. I have updated the old example to use this new technique. You can find this updated example at GitHub. There are comments at every step of the client code, but don't hesitate to ask if anything is not clear.

If you enjoyed this post, you might also like:

There and Back Again, A GraphQL Lifecycle Tale

What Good is a Flexible Paperclip?

Supporting Android permissions in React Native

Clearance

Dotfiles

More...

Locations

Austin, TX

Boston, MA

London, UK

New York, NY

Raleigh/Durham, NC

San Francisco, CA

Podcasts

The Bike Shed

Build Phase

Giant Robots

Tentative