Introduction

HTML5 finally provides a standard way to interact with local files, via the File API specification. As example of its capabilities, the File API could be used to create a thumbnail preview of images as they're being sent to the server, or allow an app to save a file reference while the user is offline. Additionally, you could use client-side logic to verify an upload's mime type matches its file extension or restrict the size of an upload.

The spec provides several interfaces for accessing files from a 'local' filesystem:

1. File - an individual file; provides read only information such as name, file size, mimetype, and a reference to the file handle.
2. FileList - an array-like sequence of File objects. (Think <input type="file" multiple> or dragging a directory of files from the desktop).
3. Blob - Allows for slicing a file into byte ranges.

When used in conjunction with the above data structures, the FileReader interface can be used to asynchronously read a file through familiar JavaScript event handling. Thus, it is possible to monitor the progress of a read, catch errors, and determine when a load is complete. In many ways the APIs resemble XMLHttpRequest's event model.

Selecting files

The first thing to do is check that your browser fully supports the File API:

```
// Check for the various File API support.
if (window.File && window.FileReader && window.FileList && window.Blob) {
  // Great success! All the File APIs are supported.
} else {
  alert('The File APIs are not fully supported in this browser.');
}
```

Of course, if your app will only use a few of these APIs, modify this snippet accordingly.

Using form input for selecting

The most straightforward way to load a file is to use a standard <input type="file"> element. JavaScript returns the list of selected File objects as a FileList. Here's an example that uses the 'multiple' attribute to allow selecting several files at once:

```
<input type="file" id="files" name="files[]" multiple />
<output id="list"></output>

<script>
  function handleFileSelect(evt) {
    var files = evt.target.files; // FileList object

    // files is a FileList of File objects. List some properties.
    var output = [];
    for (var i = 0, f; f = files[i]; i++) {
      output.push('<li><strong>', escape(f.name), '</strong> (', f.type ||
'n/a', ') - ',
                  f.size, ' bytes, last modified: ',
                  f.lastModifiedDate ? f.lastModifiedDate.toLocaleDateString() :
'n/a',
                  '</li>');
    }
    document.getElementById('list').innerHTML = '<ul>' + output.join('') +
'</ul>';
  }

  document.getElementById('files').addEventListener('change',
handleFileSelect, false);
</script>
```

Example: Using form input for selecting. Try it!

Using drag and drop for selecting

Another technique for loading files is native drag and drop from the desktop to the browser. We can modify the previous example slightly to include drag and drop support.

```
<div id="drop_zone">Drop files here</div>
<output id="list"></output>

<script>
  function handleFileSelect(evt) {
    evt.stopPropagation();
    evt.preventDefault();

    var files = evt.dataTransfer.files; // FileList object.

    // files is a FileList of File objects. List some properties.
    var output = [];
    for (var i = 0, f; f = files[i]; i++) {
      output.push('<li><strong>', escape(f.name), '</strong> (', f.type ||
'n/a', ') - ',
                  f.size, ' bytes, last modified: ',
                  f.lastModifiedDate ? f.lastModifiedDate.toLocaleDateString() :
'n/a',
                  '</li>');
    }
    document.getElementById('list').innerHTML = '<ul>' + output.join('') +
'</ul>';
  }

  function handleDragOver(evt) {
    evt.stopPropagation();
    evt.preventDefault();
    evt.dataTransfer.dropEffect = 'copy'; // Explicitly show this is a copy.
  }

  // Setup the dnd listeners.
  var dropZone = document.getElementById('drop_zone');
  dropZone.addEventListener('dragover', handleDragOver, false);
```

```
  dropZone.addEventListener('drop', handleFileSelect, false);
</script>
```

Example: Using drag and drop for selecting. Try it!

Drop files here

Note: Some browsers treat <input type="file"> elements as native drop targets. Try dragging files onto the input field in the previous example.

Reading files

Now comes the fun part!

After you've obtained a File reference, instantiate a FileReader object to read its contents into memory. When the load finishes, the reader's onload event is fired and its result attribute can be used to access the file data.

FileReader includes four options for reading a file, asynchronously:

- FileReader.readAsBinaryString(Blob|File) - The result property will contain the file/blob's data as a binary string. Every byte is represented by an integer in the range [0..255].
- FileReader.readAsText(Blob|File, opt_encoding) - The result property will contain the file/blob's data as a text string. By default the string is decoded as 'UTF-8'. Use the optional encoding parameter can specify a different format.
- FileReader.readAsDataURL(Blob|File) - The result property will contain the file/blob's data encoded as a data URL.
- FileReader.readAsArrayBuffer(Blob|File) - The result property will contain the file/blob's data as an ArrayBuffer object.

Once one of these read methods is called on your FileReader object, the onloadstart, onprogress, onload, onabort, onerror, and onload end can be used to track its progress.

The example below filters out images from the user's selection, calls reader.readAsDataURL() on the file, and renders a thumbnail by setting the 'src' attribute to a data URL.

```
<style>
  .thumb {
    height: 75px;
    border: 1px solid #000;
    margin: 10px 5px 0 0;
  }
</style>

<input type="file" id="files" name="files[]" multiple />
<output id="list"></output>

<script>
  function handleFileSelect(evt) {
    var files = evt.target.files; // FileList object

    // Loop through the FileList and render image files as thumbnails.
    for (var i = 0, f; f = files[i]; i++) {

      // Only process image files.
      if (!f.type.match('image.*')) {
        continue;
      }

      var reader = new FileReader();

      // Closure to capture the file information.
      reader.onload = (function(theFile) {
        return function(e) {
          // Render thumbnail.
          var span = document.createElement('span');
          span.innerHTML = ['<img class="thumb" src="', e.target.result,
                            '" title="', escape(theFile.name), '"/>'].join('');
          document.getElementById('list').insertBefore(span, null);
        };
      })(f);

      // Read in the image file as a data URL.
      reader.readAsDataURL(f);
```

```
    }
  }

  document.getElementById('files').addEventListener('change',
handleFileSelect, false);
</script>
```

Example: Reading files. Try it!

Try this example with a directory of images!

Slicing a file

In some cases reading the entire file into memory isn't the best option. For example, say you wanted to write an async file uploader. One possible way to speed up the upload would be to read and send the file in separate byte range chunks. The server component would then be responsible for reconstructing the file content in the correct order.

Lucky for us, the File interface supports a slice method to support this use case. The method takes a starting byte as its first argument, ending byte as its second, and an option content type string as a third.

```
var blob = file.slice(startingByte, endindByte);
reader.readAsBinaryString(blob);
```

The following example demonstrates reading chunks of a file. Something worth noting is that it uses the onloadend and checks the evt.target.readyStateinstead of using the onload event.

```
<style>
  #byte_content {
    margin: 5px 0;
    max-height: 100px;
    overflow-y: auto;
    overflow-x: hidden;
  }
  #byte_range { margin-top: 5px; }
```

```
</style>

<input type="file" id="files" name="file" /> Read bytes:
<span class="readBytesButtons">
  <button data-startbyte="0" data-endbyte="4">1-5</button>
  <button data-startbyte="5" data-endbyte="14">6-15</button>
  <button data-startbyte="6" data-endbyte="7">7-8</button>
  <button>entire file</button>
</span>
<div id="byte_range"></div>
<div id="byte_content"></div>

<script>
  function readBlob(opt_startByte, opt_stopByte) {

    var files = document.getElementById('files').files;
    if (!files.length) {
      alert('Please select a file!');
      return;
    }

    var file = files[0];
    var start = parseInt(opt_startByte) || 0;
    var stop = parseInt(opt_stopByte) || file.size - 1;

    var reader = new FileReader();

    // If we use onloadend, we need to check the readyState.
    reader.onloadend = function(evt) {
      if (evt.target.readyState == FileReader.DONE) { // DONE == 2
        document.getElementById('byte_content').textContent =
evt.target.result;
        document.getElementById('byte_range').textContent =
            ['Read bytes: ', start + 1, ' - ', stop + 1,
             ' of ', file.size, ' byte file'].join('');
      }
    };
```

```
    var blob = file.slice(start, stop + 1);
    reader.readAsBinaryString(blob);
  }

  document.querySelector('.readBytesButtons').addEventListener('click',
function(evt) {
    if (evt.target.tagName.toLowerCase() == 'button') {
      var startByte = evt.target.getAttribute('data-startbyte');
      var endByte = evt.target.getAttribute('data-endbyte');
      readBlob(startByte, endByte);
    }
  }, false);
</script>
```

Example: Slicing a file. Try it!

Read bytes: 1-5 6-157-8 entire file

Monitoring the progress of a read

One of the nice things that we get for free when using async event handling is the ability to monitor the progress of the file read; useful for large files, catching errors, and figuring out when a read is complete.

The onloadstart and onprogress events can be used to monitor the progress of a read.

The example below demonstrates displaying a progress bar to monitor the status of a read. To see the progress indicator in action, try a large file or one from a remote drive.

```
<style>
  #progress_bar {
    margin: 10px 0;
    padding: 3px;
    border: 1px solid #000;
    font-size: 14px;
    clear: both;
    opacity: 0;
    -moz-transition: opacity 1s linear;
```

```
    -o-transition: opacity 1s linear;
    -webkit-transition: opacity 1s linear;
  }
  #progress_bar.loading {
    opacity: 1.0;
  }
  #progress_bar .percent {
    background-color: #99ccff;
    height: auto;
    width: 0;
  }
</style>

<input type="file" id="files" name="file" />
<button onclick="abortRead();">Cancel read</button>
<div id="progress_bar"><div class="percent">0%</div></div>

<script>
  var reader;
  var progress = document.querySelector('.percent');

  function abortRead() {
    reader.abort();
  }

  function errorHandler(evt) {
    switch(evt.target.error.code) {
      case evt.target.error.NOT_FOUND_ERR:
        alert('File Not Found!');
        break;
      case evt.target.error.NOT_READABLE_ERR:
        alert('File is not readable');
        break;
      case evt.target.error.ABORT_ERR:
        break; // noop
      default:
        alert('An error occurred reading this file.');
    };
```

```
  }

  function updateProgress(evt) {
    // evt is an ProgressEvent.
    if (evt.lengthComputable) {
      var percentLoaded = Math.round((evt.loaded / evt.total) * 100);
      // Increase the progress bar length.
      if (percentLoaded < 100) {
        progress.style.width = percentLoaded + '%';
        progress.textContent = percentLoaded + '%';
      }
    }
  }

  function handleFileSelect(evt) {
    // Reset progress indicator on new file selection.
    progress.style.width = '0%';
    progress.textContent = '0%';

    reader = new FileReader();
    reader.onerror = errorHandler;
    reader.onprogress = updateProgress;
    reader.onabort = function(e) {
      alert('File read cancelled');
    };
    reader.onloadstart = function(e) {
      document.getElementById('progress_bar').className = 'loading';
    };
    reader.onload = function(e) {
      // Ensure that the progress bar displays 100% at the end.
      progress.style.width = '100%';
      progress.textContent = '100%';
      setTimeout("document.getElementById('progress_bar').className='';",
2000);
    }

    // Read in the image file as a binary string.
    reader.readAsBinaryString(evt.target.files[0]);
```

```
  }

  document.getElementById('files').addEventListener('change',
handleFileSelect, false);
</script>
```

Example: Monitoring the progress of a read. Try it!

Cancel read

0%

Tip: To really see this progress indicator in action, try a large file or a resource on a remote drive.

References

- File API specification
- FileReader interface specification
- Blob interface specification
- FileError interface specification
- ProgressEvent specification