Chapter 89: File API, Blobs and FileReaders

Property/Method	Description
`error`	A error that occurred while reading the file.
`readyState`	Contains the current state of the FileReader.
`result`	Contains the file contents.
`onabort`	Triggered when the operation is aborted.
`onerror`	Triggered when an error is encountered.
`onload`	Triggered when the file has loaded.
`onloadstart`	Triggered when the file loading operation has started.
`onloadend`	Triggered when the file loading operation has ended.
`onprogress`	Triggered whilst reading a Blob.
`abort()`	Aborts the current operation.
`readAsArrayBuffer(blob)`	Starts reading the file as an ArrayBuffer.
`readAsDataURL(blob)`	Starts reading the file as a data url/uri.
`readAsText(blob[, encoding])`	Starts reading the file as a text file. Not able to read binary files. Use readAsArrayBuffer instead.

Section 89.1: Read file as string

Make sure to have a file input on your page:

```
<input type="file" id="upload">
```

Then in JavaScript:

```
document.getElementById('upload').addEventListener('change', readFileAsString)
function readFileAsString() {
    var files = this.files;
    if (files.length === 0) {
        console.log('No file is selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        console.log('File content:', event.target.result);
    };
    reader.readAsText(files[0]);
}
```

Section 89.2: Read file as dataURL

Reading the contents of a file within a web application can be accomplished by utilizing the HTML5 File API. First, add an input with `type="file"` in your HTML:

```
<input type="file" id="upload">
```

Next, we're going to add a change listener on the file-input. This examples defines the listener via JavaScript, but it could also be added as attribute on the input element. This listener gets triggered every time a new file has been selected. Within this callback, we can read the file that was selected and perform further actions (like creating an image with the contents of the selected file):

```
document.getElementById('upload').addEventListener('change', showImage);

function showImage(evt) {
    var files = evt.target.files;

    if (files.length === 0) {
        console.log('No files selected');
        return;
    }

    var reader = new FileReader();
    reader.onload = function(event) {
        var img = new Image();
        img.onload = function() {
            document.body.appendChild(img);
        };
        img.src = event.target.result;
    };
    reader.readAsDataURL(files[0]);
}
```

Section 89.3: Slice a file

The `blob.slice()` method is used to create a new Blob object containing the data in the specified range of bytes of the source Blob. This method is usable with File instances too, since File extends Blob.

Here we slice a file in a specific amount of blobs. This is useful especially in cases where you need to process files that are too large to read in memory all in once. We can then read the chunks one by one using `FileReader`.

```
/**
* @param {File|Blob} - file to slice
* @param {Number} - chunksAmount
* @return {Array} - an array of Blobs
**/
function sliceFile(file, chunksAmount) {
  var byteIndex = 0;
  var chunks = [];

  for (var i = 0; i < chunksAmount; i += 1) {
    var byteEnd = Math.ceil((file.size / chunksAmount) * (i + 1));
    chunks.push(file.slice(byteIndex, byteEnd));
    byteIndex += (byteEnd - byteIndex);
  }

  return chunks;
}
```

Section 89.4: Get the properties of the file

If you want to get the properties of the file (like the name or the size) you can do it before using the File Reader. If we have the following html piece of code:

```
<input type="file" id="newFile">
```

You can access the properties directly like this:

```
document.getElementById('newFile').addEventListener('change', getFile);
```

```
function getFile(event) {
    var files = event.target.files
        , file = files[0];

    console.log('Name of the file', file.name);
    console.log('Size of the file', file.size);
}
```

You can also get easily the following attributes: `lastModified` (Timestamp), `lastModifiedDate` (Date), and `type` (File Type)

Section 89.5: Selecting multiple files and restricting file types

The HTML5 file API allows you to restrict which kind of files are accepted by simply setting the accept attribute on a file input, e.g.:

```
<input type="file" accept="image/jpeg">
```

Specifying multiple MIME types separated by a comma (e.g. `image/jpeg,image/png`) or using wildcards (e.g. `image/*` for allowing all types of images) give you a quick and powerful way to restrict the type of files you want to select. Here's an example for allowing any image or video:

```
<input type="file" accept="image/*,video*">
```

By default, the file input lets the user select a single file. If you want to enable multiple file selection, simply add the `multiple` attribute:

```
<input type="file" multiple>
```

You can then read all the selected files via the file input's `files` array. See read file as dataUrl

Section 89.6: Client side csv download using Blob

```
function downloadCsv() {
  var blob = new Blob([csvString]);
  if (window.navigator.msSaveOrOpenBlob){
    window.navigator.msSaveBlob(blob, "filename.csv");
  }
  else {
    var a = window.document.createElement("a");

    a.href = window.URL.createObjectURL(blob, {
      type: "text/plain"
    });
    a.download = "filename.csv";
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
  }
}
var string = "a1,a2,a3";
downloadCSV(string);
```

Source reference ; https://github.com/mholt/PapaParse/issues/175