The FileEntry

Files in the sandboxed filesystem are represented by the `FileEntry` interface. A `FileEntry` contains the types of properties and methods one would expect from a standard filesystem.

Properties

`isFile`

Boolean. True if the entry is a file.

`isDirectory`

Boolean. True if the entry is a directory.

`name`

DOMString. The name of the entry, excluding the path leading to it.

`fullPath`

DOMString. The full absolute path from the root to the entry.

`filesystem`

`FileSystem`. The filesystem on which the entry resides.

Methods

`getMetadata (successCallback, opt_errorCallback)`

Look up metadata about this entry.

`moveTo (parentDirEntry, opt_newName, opt_successCallback, opt_errorCallback)`

Move an entry to a different location on the filesystem.

`copyTo (parentDirEntry, opt_newName, opt_successCallback, opt_errorCallback)`

Copies an entry to a different parent on the filesystem. Directory copies are always recursive. It is an error to copy a directory inside itself or to copy it into its parent if a new name is not provided.

`toURL ();`

Returns a `filesystem:` URL that can be used to identify this file. See Chapter 7.

`remove (successCallback, opt_errorCallback)`

Deletes a file or directory. It is an error to attempt to delete the root directory of a filesystem or a directory that is not empty.

`getParent (successCallback, opt_errorCallback)`

Return the parent `DirectoryEntry` containing this entry. If this entry is the root directory, its parent is itself.

`createWriter (successCallback, opt_errorCallback)`

Creates a new `FileWriter` (See Writing to a File) which can be used to write content to this `FileEntry`.

`file (successCallback, opt_errorCallback)`

Returns a `File` representing the `FileEntry` to the success callback.

To better understand `FileEntry`, the rest of this chapter contains code recipes for performing common tasks.

Creating a File

After Opening a Filesystem, the `FileSystem` that is passed to the success callback contains the root `DirectoryEntry` (as `fs.root`). To look up or create a file in this directory, call its `getFile()`, passing the name of the file to create.

For example, the following code creates an empty file called *log.txt* in the root directory.

Example 4-1. Creating a file and printing its last modified time

```
function onFs(fs) {

  fs.root.getFile('log.txt', {create: true, exclusive: true},
      function(fileEntry) {
        // fileEntry.isFile === true
        // fileEntry.name == 'log.txt'
        // fileEntry.fullPath == '/log.txt'

        fileEntry.getMetaData(function(md) {
          console.log(md.modificationTime.toDateString());
        }, onError);

      },
      onError
  );
}

window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, onFs, onError);
```

The first argument to `getFile()` can be an absolute or relative path, but it must be valid. For instance, it is an error to attempt to create a file whose immediate parent does not exist. The second argument is an object literal describing the function's behavior if the file does not exist. In this example, `create: true` creates the file if it doesn't exist and throws an error if it does (`exclusive: true`). Otherwise if `create: false`, the file is simply fetched and returned. By itself, the `exclusive` option has no effect. In either case, the file contents are not overwritten. We're simply obtaining a reference entry to the file in question.

Reading a File by Name

Calling `getFile()` only retrieves a `FileEntry`. It does not return the contents of a file. For that, we need a `File` object and the `FileReader` API. To obtain a `File`, call `FileEntry.file()`. Its first argument is a success callback which is passed the file, and its second is an error callback.

The following code retrieves the file named *log.txt*. Its contents are read into memory as text using the `FileReader` API, and the result is appended to the DOM as a new `<textarea>`. If *log.txt* does not exist, an error is thrown.

Example 4-2. Reading a text file

```
function onFs(fs) {

  fs.root.getFile('log.txt', {}, function(fileEntry) {

    // Obtain the File object representing the FileEntry.
    // Use FileReader to read its contents.
    fileEntry.file(function(file) {
       var reader = new FileReader();

       reader.onloadend = function(e) {
         var textarea = document.createElement('textarea');
         textarea = this.result;
         document.body.appendChild(textarea);
       };

       reader.readAsText(file); // Read the file as plaintext.
    }, onError);

  }, onError);

}
```

```
window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, onFs, onError);
```

Writing to a File

The API exposes the `FileWriter` interface for writing content to a `FileEntry`.

Properties

`position`

> Integer. The byte offset at which the next write will occur. For example, a newly-created `FileWriter` has position set to 0.

`length`

> Integer. The length of the file.

`error`

> `FileError`. The last error that occurred.

`readyState`

> One of 3 states: `INIT`, `WRITING`, `DONE`.

Methods

`abort ()`

> Aborts a write operation in progress. If `readyState` is `DONE` or `INIT`, an `INVALID_STATE_ERR` exception is thrown.

`write (blob)`

> Writes the supplied data to the file, starting at the offset given by `position`. The argument can be a `Blob` or `File` object.

`seek (offset)`

> Sets the file `position` at which the next write occurs. The argument is a byte offset into the file. If offset > length, length is used instead. If offset is < 0, `position` is set back from the end of the file.

`truncate (size)`

> Changes the `length` of the file to a new size. Shortening the file discards any data beyond the new length. Extending it beyond the current length zero-pads the existing data up to the new `length`.

Events

`onabort`

> Called when an in-progress write operation is cancelled.

`onerror`

> Called when an error occurs.

`onprogress`

> Called periodically as data is being written.

`onwrite`

> Called when the write operation has successfully completed.

`onwritestart`

Called just before writing is about to start.

`onwriteend`

Called when the write is complete, whether successful or not.

The following code creates an empty file called *log.txt* in a subfolder, */temp*. If the file already exists, it is simple retrieved. The text “Lorem Ipsum” is written to it by constructing a new `Blob` using the BlobBuilder API, and handing it off to `FileWriter.write()`. Event handlers are set up to monitor `error` and `writeend` events.

Example 4-3. Writing text to a file

```
function onFs(fs) {

  fs.root.getFile('/temp/log.txt', {create: true}, function(fileEntry) {

    // Create a FileWriter object for our FileEntry.
    fileEntry.createWriter(function(fileWriter) {

      fileWriter.onwrite = function(e) {
        console.log('Write completed.');
      };

      fileWriter.onerror = function(e) {
        console.log('Write failed: ' + e.toString());
      };

      var bb = new BlobBuilder(); // Create a new Blob on-the-fly.
      bb.append('Lorem Ipsum');

      fileWriter.write(bb.getBlob('text/plain'));

    }, onError);

  }, onError);

}

window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, onFs, onError);
```

Warning

The BlobBuilder API has been vendor prefixed in Firefox 6 and Chrome:

```
window.BlobBuilder = window.BlobBuilder || window.WebKitBlobBuilder || window.MozBlobBuilder;
```

If the folder */temp* did not exist in the filesystem, an error is thrown.

Appending Data to a File

Appending data onto an existing file is trivial with `FileWriter`. We can reposition the writer to the end of the file using `seek()`. Seek takes a byte offset as an argument, setting the file writer’s `position` to that offset. If the offset is greater than the file’s length, the current length is used instead. If offset is < 0, `position` is set back from the end of the file.

As an example, the following snippet appends a timestamp to the end of a log file. An error is thrown if the file does not yet exist.

Example 4-4. Logging a timestamp

```
window.BlobBuilder = window.BlobBuilder || window.WebKitBlobBuilder ||
                     window.MozBlobBuilder;

function append(fs, filePath, blob) {
  fs.root.getFile(filePath, {create: false}, function(fileEntry) {

    // Create a FileWriter object for our FileEntry.
    fileEntry.createWriter(function(fileWriter) {

      fileWriter.seek(fileWriter.length); // Start write position at EOF.
      fileWriter.write(bb.getBlob('text/plain'));

    }, onError);

  }, onError);
}

function onFs(fs) {
  var bb = new BlobBuilder();
  bb.append((new Date()).toISOString() + '\n');

  append(fs, 'log.txt', bb.getBlob('text/plain'));
}

window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, onFs, onError);
```

Importing Files

For security reasons, the HTML5 Filesystem API does not allow applications to write data outside of their sandbox. As a result of this restriction, applications cannot share filesystems and they cannot read or write files to arbitrary folders on the user's hard drive, such as their My Pictures or My Music folder. This leaves developers in a bit of a predicament. How does one import files into a web application if the application cannot access the user's full hard drive with all of their precious files?

There are four techniques to import data into the filesystem:

- Use `<input type="file">`. The user selects files from a location on their machine and the application duplicates those files into the app's HTML5 filesystem.
- Use HTML5 drag and drop. Some browsers support dragging in files from the desktop to the browser tab. Again, the selected files would be duplicated into the HTML5 filesystem.
- Use `XMLHttpRequest`. New properties in XMLHttpRequest 2 make it trivial to fetch remote binary data, then store that data locally using the HTML5 filesystem.
- Using copy and paste events. Apps can read clipboard information that contains file data.

Using <input type="file">

The first (and most common) way to import files into an app is to repurpose our old friend `<input type="file">`. I say repurpose because we're not interested in uploading form data—the typical usage of a file input. Instead, we can utilize the browser's native file picker, prompt users to select files, and save those selections into our app.

The following example allows users to select multiple files using `<input type="file" multiple>` and creates copies of those files in the app's sandboxed filesystem.

Example 4-5. Duplicating user-selected files

```
<input type="file" id="myfile" multiple />

// Creates a file if it doesn't exist.
// Throws an error if a file already exists with the same name.

var writeFile = function(parentDirectory, file) {
  parentDirectory.getFile(file.name, {create: true, exclusive: true},
      function(fileEntry) {


        fileEntry.createWriter(function(fileWriter) {
          fileWriter.write(file);
        }, onError);

      },
      onError
  );
};

document.querySelector("input[type='file']").onchange = function(e) {
  var files = this.files;

  window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, function(fs) {
    for (var i = 0, file; file = files[i]; ++i){
      writeFile(fs.root, file);
    }
  }, onError);
};
```

As noted in the comment, `FileWriter.write()` accepts a `Blob` or `File`. This is because `File` inherits from `Blob`, and therefore, all files are blobs. The reverse is not true.

Consider allowing users to import an entire folder using `<input type="file" webkidirectory>`. By including this attribute, the browser allows users to select a folder and recursively read all the files in it. The result is a `FileList` of every file in the folder.

Example 4-6. Importing a directory

```
<input type="file" id="myfile" webkitdirectory />

// Creates a file if it doesn't exist.
// Throw an error if a file already exists with the same name.

var writeFile = function(parentDirectory, file) {
  parentDirectory.getFile(file.name, {create: true, exclusive: true},
      function(fileEntry) {

        // Write the file. write() can take a File or Blob.
        fileEntry.createWriter(function(fileWriter) {
          fileWriter.write(file);
        }, onError);

      },
      onError
  );
};

document.querySelector("#myfile").onchange = function(e) {
  for (var i = 0, f; f = e.target.files[i]; ++i) {
    console.log(f.webkitRelativePath);
  }
};
```

What’s not shown in the above example is the writing of each file to the proper directory. Creating folders is covered in the next chapter.

Using HTML5 Drag and Drop

The second method for importing files is to use HTML5 drag and drop. Some people love it. Some people hate it. But whether or not you’re a fan of HTML5’s drag and drop design, it is here to stay. That said, one really nice thing drag and drop gives us is a familiar way for users to import data into our web applications.

Chrome, Safari 5, and Firefox 4 extend HTML5 drag and drop events by allowing files to be dragged in from the desktop to the browser window. In fact, the process for setting up event listeners to handle dropped file(s) is exactly the same as handling other types of content. The only difference is the way the files are accessed in the drop handler. Typically, dropped data is read from the event’s `dataTransfer` property (as `dataTransfer.getData()`). However, when handling files, data is read from `dataTransfer.files`. If that looks suspiciously familiar, it should be! This is the drag and drop equivalent of the previous example using `<input type="file">`.

The following example allows users to drag in files from the desktop. On the `dragenter` and `dragleave` events, the class “dropping” is toggled to give the user a visual indication a drop can occur.

Example 4-7. Importing files using drag and drop from the desktop

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Drag and drop files</title>
```

```
<style>
.dropping {
  background: -webkit-repeating-radial-gradient(white, #fc0 5px);
  background: -moz-repeating-radial-gradient(white, #fc0 5px);
  background: -ms-repeating-radial-gradient(white, #fc0 5px);
  background: -o-repeating-radial-gradient(white, #fc0 5px);
  background: repeating-radial-gradient(white, #fc0 5px);
}
</style>
</head>
<body>

<p>Drag files in from your desktop. They will be added to the filesystem.</p>

<script>
window.requestFileSystem = window.requestFileSystem ||
                           window.webkiRequestFileSystem;

/**
 * Class to handle drag and drop events on an element.
 *
 * @param {string} selector A CSS selector for an element to attach drag and
 *     drop events to.
 * @param {function(FileList)} onDropCallback A callback passed the list of
 *     files that were dropped.
 * @constructor
 */
function DnDFileController(selector, onDropCallback) {
  var el_ = document.querySelector(selector);

  this.dragenter = function(e) {
    e.stopPropagation();
    e.preventDefault();

    // Give a visual indication this element is a drop target.
    el_.classList.add('dropping');
  };

  this.dragover = function(e) {
    e.stopPropagation();
    e.preventDefault();
  };

  this.dragleave = function(e) {
    e.stopPropagation();
    e.preventDefault();
    el_.classList.remove('dropping');
  };

  this.drop = function(e) {
    e.stopPropagation();
    e.preventDefault();

    el_.classList.remove('dropping');

    onDropCallback(e.dataTransfer.files);
  };

  el_.addEventListener('dragenter', this.dragenter, false);
  el_.addEventListener('dragover', this.dragover, false);
  el_.addEventListener('dragleave', this.dragleave, false);
  el_.addEventListener('drop', this.drop, false);
};

var FS = null; // Cache the FileSystem object for later use.

// Allow dropping onto the entire page.
var controller = new DnDFileController('body', function(files) {
  [].forEach.call(files, function(file, i) {
    // See Example 4-5 for the defintion of writeFile().
    writeFile(FS.root, file);
  });
});
```

```
(function openFS() {
  window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, function(fs) {
    FS = fs;
  }, onError);
})();
</script>
</body>
</html>
```

Using XMLHttpRequest

A third way to import data is to use `XMLHttpRequest` to fetch remote files. The difference between this method and the first two methods is that this option requires data to already exist somewhere in the cloud. In most cases that’s not a problem. As web developers, we have learned to deal with remote data and encounter it all the time.

Many of the enhancements put into XMLHttpRequest Level 2 are designed for better interoperability with binary data, blobs, and files. This is really good news for web developers. It means we can put an end to crazy string manipulation and error-prone character code hacks in our applications. As an example of what I mean, here is one well-known trick to fetch an image as a binary string.

Example 4-8. Old way to fetch a binary file

```
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); // or String.fromCharCode()
      var byte = c & 0xff; // byte at offset i
      ...
    }
  }
};

xhr.send(null);
```

While this technique 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. Any time you resort to character code hacks and string manipulation for coercing data into a desirable format, that’s a problem. Instead, `XMLHttpRequest` now exposes `responseType` and `response` properties to inform the browser what format to return data in:

`xhr.responseType`

> After opening a new request but before sending it, set `xhr.responseType` to “text”, “arraybuffer”, “blob”, or “document”, depending on your data needs. Setting `xhr.responseType=''` or omitting altogether defaults the `response` to “text” (i.e., `xhr.responseText === xhr.response`).

`xhr.response`

> After a successful request, the `xhr.response` contains the requested data as a `DOMString`, `ArrayBuffer`, `Blob`, or `Document`, according to what `xhr.responseType` was set to.

With these new tools, we can clean up the previous example by reworking how the data is fetched. This time, the image is downloaded as an `ArrayBuffer` instead of a binary string, then handed over to the `BlobBuilder` API to create a `Blob`.

Example 4-9. Fetch an image file as a blob and write it to the filesystem

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Fetch + write an image to the HTML5 filesystem</title>
</head>
```

```
<body>
<script>
// Take care of vendor prefixes.
window.BlobBuilder = window.BlobBuilder || window.WebKitBlobBuilder ||
                     window.MozBlobBuilder;
window.requestFileSystem = window.requestFileSystem ||
                           window.webkitRequestFileSystem;

var onError = function(e) {
  console.log('There was an error', e);
};

/**
 * Writes a Blob to the filesystem.
 *
 * @param {DirectoryEntry} dir The directory to write the blob into.
 * @param {Blob} blob The data to write.
 * @param {string} fileName A name for the file.
 * @param {function(ProgressEvent)} opt_callback An optional callback.
 *     Invoked when the write completes.
 */

var writeBlob = function(dir, blob, fileName, opt_callback) {
  dir.getFile(fileName, {create: true, exclusive: true}, function(fileEntry) {

    fileEntry.createWriter(function(writer) {
      if (opt_callback) {
        writer.onwrite = opt_callback;
      }
      writer.write(blob);
    }, onError);

  }, onError);
};

/**
 * Fetches a file by URL and writes it to the filesystem.
 *
 * @param {string} url The url the resource resides under.
 * @param {string} mimeType The content type of the file.
 */
var downloadImage = function(url, mimeType) {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', url, true);
  xhr.responseType = 'arraybuffer';

  xhr.onload = function(e) {
    if (this.status == 200) {
      var bb = new BlobBuilder();
      bb.append(xhr.response); // Note: not xhr.responseText

      var parts = url.split('/');
      var fileName = parts[parts.length - 1];

      window.requestFileSystem(TEMPORARY, 1024*1024*5 /*5MB*/, function(fs) {
        var onWrite = function(evt) {
          console.log('Write completed.');
        };

        // Write file to the root directory.
        writeBlob(fs.root, bb.getBlob(mimeType), fileName, onWrite);
      }, onError);
    }
  };

  xhr.send(null);
};

if (window.requestFileSystem && window.BlobBuilder) {
  downloadImage('/path/to/image.png', 'image/png');
}
</script>
</body>
</html>
```

Using Copy and Paste

A final way to import files involves pasting files in your application. This is done by setting up and `onpaste` handler on the document body and iterating through the event's `clipboardData` items. Each item has a "kind" and "type" property. Checking the "kind" property can be used to verify whether or not a pasted item is a file. If `item.kind == "file"`, then the item is indeed a file.

The following example sets up an `onpaste` listener on the page, allowing users to paste in *.png*s. The images/items are then read as Blobs using `getAsFile()`, and written into the filesystem.

Example 4-10. Pasting a file into an application and saving it to the filesystem

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Pasting a file into an application and saving it to the filesystem</title>
</head>
<body>
<p>
  Copy an image from the Web (right-click > Copy Image), click in this window,
  and paste it in.
</p>
<script>
// Take care of vendor prefixes.
window.requestFileSystem = window.requestFileSystem ||
                           window.webkiRequestFileSystem;
window.URL = window.URL || window.webkitURL;

var onError = function(e) {
  console.log('There was an error', e);
};

/**
 * Writes a Blob to the filesystem.
 *
 * @param {DirectoryEntry} dir The directory to write the blob into.
 * @param {Blob} blob The data to write.
 * @param {string} fileName A name for the file.
 * @param {function(ProgressEvent)} opt_callback An optional callback.
 *     Invoked when the write completes.
 */
var writeBlob = function(dir, blob, fileName, opt_callback) {
  dir.getFile(fileName, {create: true}, function(fileEntry) {

    fileEntry.createWriter(function(writer) {
      if (opt_callback) {
        writer.onwrite = opt_callback;
      }
      writer.write(blob);
    }, onError);

  }, onError);
};


// Setup onpaste handler to catch dropped .png files.
document.body.onpaste = function(e) {
  var items = e.clipboardData.items;
  for (var i = 0; i < items.length; ++i) {
    if (items[i].kind == 'file' && items[i].type == 'image/png') {
      var blob = items[i].getAsFile();

      writeBlob(FS.root, blob, 'MyPastedImage', function(e) {
        console.log('Write completed.');
      });
    }
  }
};

va FS; // cache the FileSystem object for later.
window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, function(fs) {
  FS = fs;
}, onError);
</script>
</body>
</html>
```

Removing Files

To remove a file from the filesystem, call `entry.remove()`. The first argument to this method is a zero-parameter callback function, which is called when the file is successfully deleted. The second is an optional error callback if any errors occur.

Example 4-11. Removing a file by name

```
window.requestFileSystem(TEMPORARY, 1024*1024 /*1MB*/, function(fs) {
  fs.root.getFile('log.txt', {}, function(fileEntry) {

    fileEntry.remove(function() {
      console.log('File removed.');
    }, onError);

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