Read files in JavaScript

How to select files, read file metadata and content, and monitor read progress.

Jun 18, 2010 • Updated May 8, 2020

Kayce Basques
Twitter · GitHub · Glitch

Pete LePage
Twitter · GitHub · Glitch · Blog

Being able to select and interact with files on the user's local device is one of the most commonly used features of the web. It allows users to select files and upload them to a server, for example, uploading photos, or submitting tax documents, etc. But, it also allows sites to read and manipulate them without ever having to transfer the data across the network.

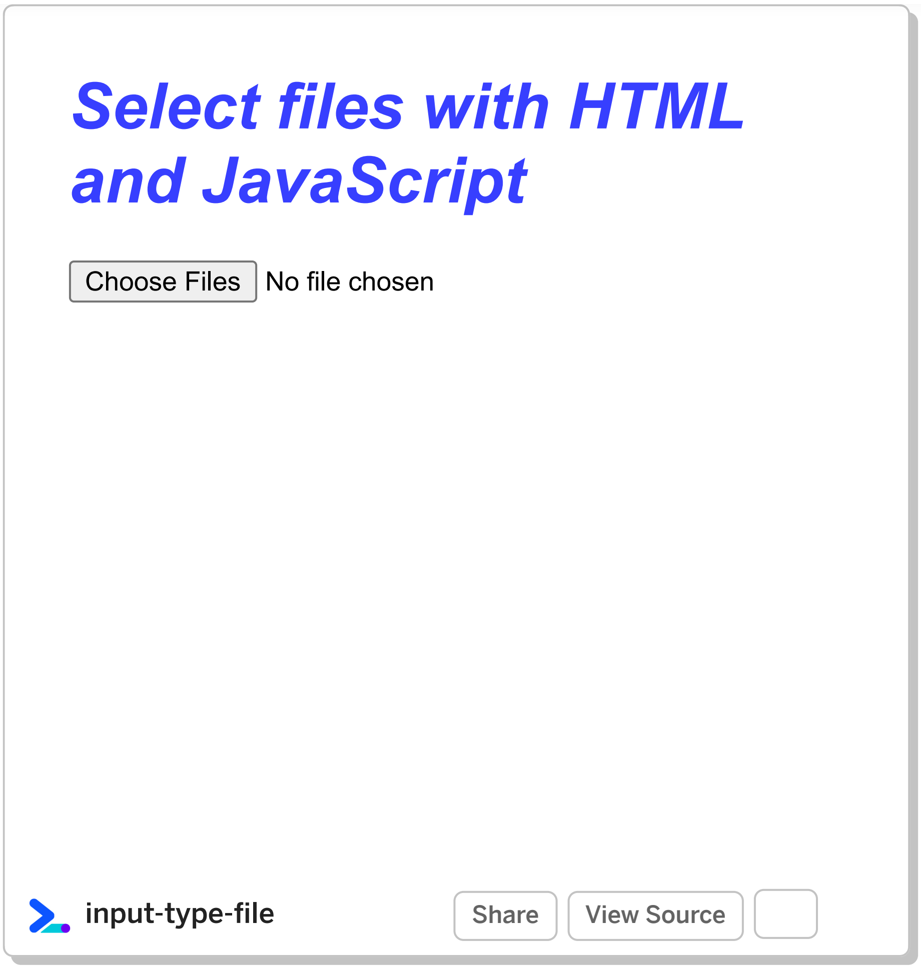
This guide shows you how to:

- Select files
  - Using the HTML input element
  - Using a drag-and-drop zone
- Read file metadata
- Read a file's content

The easiest way to allow users to select files is using the `<input type="file">` element, which is supported in every major browser. When clicked, it lets a user select a file, or multiple files if the `multiple` attribute is included, using their operating system's built-in file selection UI. When the user finishes selecting a file or files, the element's `change` event is fired. You can access the list of files from `event.target.files`, which is a `FileList` object. Each item in the `FileList` is a `File` object.

```html
<!-- The `multiple` attribute lets users select multiple files. -->
<input type="file" id="file-selector" multiple>
<script>
  const fileSelector = document.getElementById('file-selector');
  fileSelector.addEventListener('change', (event) => {
    const fileList = event.target.files;
    console.log(fileList);
  });
</script>
```

This example lets a user select multiple files using their operating system's built-in file selection UI and then logs each selected file to the console.



Select files with HTML and JavaScript

Limit the types of files user can select

In some cases, you may want to limit the types of files users can select. For example, an image editing app should only accept images, not text files. To do that, you can add an `accept` attribute to the input element to specify which files are accepted.

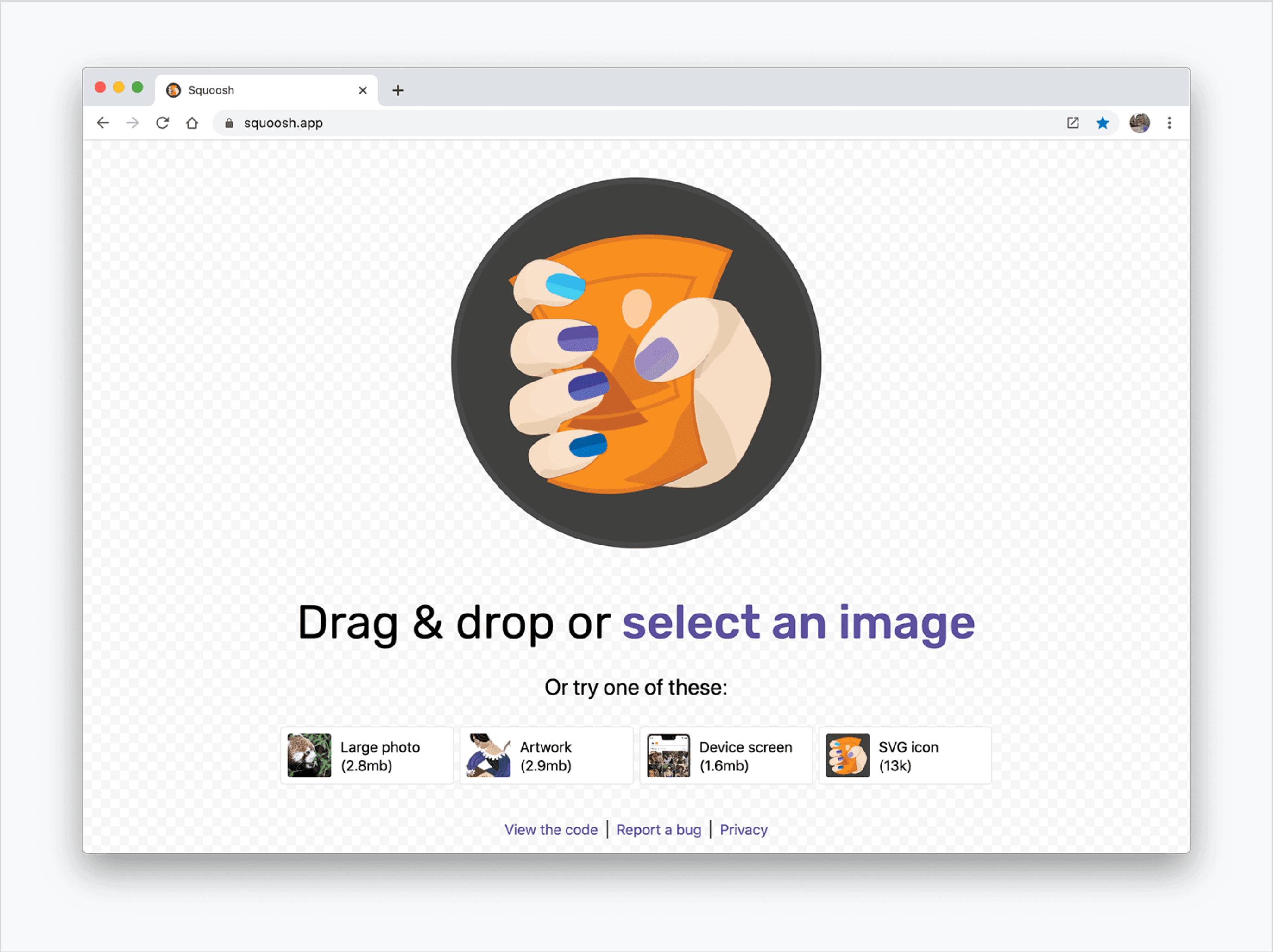
```
<input type="file" id="file-selector" accept=".jpg, .jpeg, .png">
```

Custom drag-and-drop

In some browsers, the `<input type="file">` element is also a drop target, allowing users to drag-and-drop files into your app. But, the drop target is small, and can be hard to use. Instead, once you've provided the core functionality using

Choose your drop zone

Your drop surface will depend on the design of your application. You may only want part of the window to be a drop surface, or potentially the entire window.



Drag & drop or select an image

Or try one of these:

Squoosh makes the entire window a drop zone.

Squoosh allows the user to drag-and-drop an image anywhere into the window, and clicking **select an image** invokes the `<input type="file">` element. Whatever you choose as your drop zone, make sure it's clear to the user that they can drag-and-drop files onto that surface.

Define the drop zone

To enable an element to be a drag-and-drop zone, you'll need to listen for two to the input element, you can access the list of files from `event.dataTransfer.files`, which is a [`FileList`] object. Each item in the `FileList` is a [`File`] object.

```
const dropArea = document.getElementById('drop-area');

dropArea.addEventListener('dragover', (event) => {
  event.stopPropagation();
  event.preventDefault();
  // Style the drag-and-drop as a "copy file" operation.
  event.dataTransfer.dropEffect = 'copy';
});

dropArea.addEventListener('drop', (event) => {
  event.stopPropagation();
  event.preventDefault();
  const fileList = event.dataTransfer.files;
  console.log(fileList);
});
```

`event.stopPropagation()` and `event.preventDefault()` stop the browser's default behavior from happening, and allow your code to run instead. Without them, the browser would otherwise navigate away from your page and open the files the user dropped into the browser window.

Check out Custom drag-and-drop for a live demonstration.

What about directories?

Unfortunately, today there isn't a good way to get access to a directory.

The `webkitdirectory` attribute on the `<input type="file">` element allows

If drag-and-drop is enabled, a user may try to drag a directory into the drop zone. When the drop event is fired, it will include a `File` object for the directory, but will be unable to access any of the files within the directory.

In the future, the File System Access API provides an easy way to both read and write to files and directories on the user's local system. It's currently under development and only available as an origin trial in Chrome. To learn more about it, see the File System Access API article.

Since the File System Access API is not compatible with all browsers yet, check out browser-nativefs, a helper library that uses the new API wherever it is available, but falls back to legacy approaches when it is not.

Read file metadata

The `File` object contains a number of metadata properties about the file. Most browsers provide the file name, the size of the file, and the MIME type, though depending on the platform, different browsers may provide different, or additional information.

```
function getMetadataForFileList(fileList) {
  for (const file of fileList) {
    // Not supported in Safari for iOS.
    const name = file.name ? file.name : 'NOT SUPPORTED';
    // Not supported in Firefox for Android or Opera for Android.
    const type = file.type ? file.type : 'NOT SUPPORTED';
    // Unknown cross-browser support.
    const size = file.size ? file.size : 'NOT SUPPORTED';
    console.log({file, name, type, size});
  }
```

You can see this in action in the `input-type-file` Glitch demo.

Read a file's content

To read a file, use `FileReader` , which enables you to read the content of a `File` object into memory. You can instruct `FileReader` to read a file as an array buffer, a data URL, or text.

```
function readImage(file) {
  // Check if the file is an image.
  if (file.type && file.type.indexOf('image') === -1) {
    console.log('File is not an image.', file.type, file);
    return;
  }

  const reader = new FileReader();
  reader.addEventListener('load', (event) => {
    img.src = event.target.result;
  });
  reader.readAsDataURL(file);
}
```

The example above reads a `File` provided by the user, then converts it to a data URL, and uses that data URL to display the image in an `img` element. Check out the `read-image-file` Glitch to see how to verify that the user has selected an image file.

Read an image file

Monitor the progress of a file read

When reading large files, it may be helpful to provide some UX to indicate how far the read has progressed. For that, use the `progress` event provided by `FileReader`. The `progress` event provides two properties, `loaded`, the amount read, and `total`, the total amount to read.

```
function readFile(file) {
  const reader = new FileReader();
  reader.addEventListener('load', (event) => {
    const result = event.target.result;
    // Do something with result
  });

  reader.addEventListener('progress', (event) => {
  });
  reader.readAsDataURL(file);
}
```

Hero image by Vincent Botta from Unsplash

Storage

Last updated: May 8, 2020 Improve article