Work with files

Your browser extension may need to work with files to deliver its full functionality. This article looks at the five mechanisms you have for handling files:

- Downloading files to the user's selected download folder.
- Opening files using a file picker on a web page.
- Opening files using drag and drop onto a web page.
- Storing files or blobs locally with IndexedDB using the idb-file-storage library.
- Passing files to a native application on the user's computer.

For each of these mechanisms, we introduce their use with references to the relevant API documentation, guides, and any examples that show how to use the API.

Download files using the downloads API

This mechanism enables you to get a file from your website (or any location you can define as a URL) to the user's computer. The key method is `downloads.download()`, which in its simplest form accepts a URL and downloads the file from that URL to the user's default downloads folder:

JS
```js
browser.downloads.download({ url: "https://example.org/image.png" });
```

You can let the user download to a location of their choice by specifying the `saveAs` parameter.

> **Note:** Using URL.createObjectURL() you can also download files and blobs defined in your JavaScript, which can include local content retrieved from IndexedDB.

The downloads API also provides features to cancel, pause, resume, erase, and remove downloads; search for downloaded files in the download manager; show downloaded files in the computer's file manager; and open a file in an associated application.

To use this API, you need to have the `"downloads"` API permission specified in your `manifest.json` file.

Example: Latest download API reference: downloads API

Open files in an extension using a file picker

If you want to work with a file from the user's computer one option is to let the user select a file using the computer's file browser. Either create a new page or inject code into an existing page to use the `file` type of the HTML `input` element to offer the user a file picker. Once the user has picked a file or files, the script associated with the page can access the content of the file using the DOM File API, in the same way a web application does.

Example: Imagify Guide: Using files from web applications API references: HTML input element | DOM File API

Note: If you want to access or process all the files in a selected folder, you can do so using `<input type="file" webkitdirectory="true"/>` to select the folder and return all the files it contains.

Open files in an extension using drag and drop

The Web Drag and Drop API offers an alternative to using a file picker. To use this method, establish a 'drop zone' that fits with your UI, then add listeners for the `dragenter`, `dragover`, and `drop` events to the element. In the handler for the drop event, your code can access any file dropped by the user from the object offered by the `dataTransfer` property using `DataTransfer.files`. Your code can then access and manipulate the files using the DOM File API.

Example: Imagify Guides: Using files from web applications | File drag and drop API references: DOM File API

Store files data locally using the IndexedDB file storage library

If your extension needs to save files locally, the idb-file-storage library provides a simple Promise-based wrapper to the IndexedDB API to aid the storage and retrieval of files and blobs.

The key features of the library are:

getFileStorage

Returns an `IDBFileStorage` instance, creating the named storage if it does not exist.

IDBFileStorage

Provides the methods to save and retrieve files, such as:

- list to obtain an optionally filtered list of file in the database.
- put to add a file or blob to the database.
- get to retrieve a file or blob from the database.
- remove to delete a file or blob from the database.

The Store Collected Images example illustrates how to use most of these features.

The Store Collected Images example lets users add images to a collection using an option on the image context menu. Selected images are collected in a popup and can be saved to a named collection. A toolbar button (`browserAction`) opens a navigate collection page, on which the user can view and delete saved images, with a filter option to narrow choices. See the example in action .

The workings of the library can be understood by viewing image-store.js in /utils/:

Creating the store and saving the images

JS

```
async function saveCollectedBlobs(collectionName, collectedBlobs) {
  const storedImages = await getFileStorage({ name: "stored-images" });

  for (const item of collectedBlobs) {
    await storedImages.put(`${collectionName}/${item.uuid}`, item.blob);
  }
}
```

`saveCollectedBlobs` is called when the user clicks save in the popup and has provided a name for the image collection.

First, `getFileStorage` creates, if it does not exist already, or retrieves the IndexedDB database `"stored-images"` to the object `storedImages` . `storedImages.put()` then adds each collected image to the database, under the collection name, using the blob's unique id (the file name).

If the image being stored has the same name as one already in the database, it is overwritten. If you want to avoid this, query the database first using `imagesStore.list()` with a filter for the file name; and, if the list returns a file, add a suitable suffix to the name of the new image to store a separate item.

Retrieving stored images for display

JS

```
export async function loadStoredImages(filter) {
  const imagesStore = await getFileStorage({ name: "stored-images" });
  let listOptions = filter ? { includes: filter } : undefined;
  const imagesList = await imagesStore.list(listOptions);
  let storedImages = [];
  for (const storedName of imagesList) {
    const blob = await imagesStore.get(storedName);
    storedImages.push({ storedName, blobUrl: URL.createObjectURL(blob) });
  }
  return storedImages;
}
```

`loadStoredImages()` is called when the user clicks view or reload in the navigate collection page. `getFileStorage()` opens the `"stored-images"` database, then `imagesStore.list()` gets a filtered list of the stored images. This list is then used to retrieve images with `imagesStore.get()` and build a list to return to the UI.

Note the use of `URL.createObjectURL(blob)` to create a URL that references the image blob. This URL is then used in the UI (navigate-collection.js) to display the image.

Delete collected images

JS

```
async function removeStoredImages(storedImages) {
  const imagesStore = await getFileStorage({ name: "stored-images" });
  for (const storedImage of storedImages) {
    URL.revokeObjectURL(storedImage.blobUrl);
    await imagesStore.remove(storedImage.storedName);
  }
}
```

`removeStoredImages()` is called when the user clicks delete in the navigate collection page. Again, `getFileStorage()` opens the `"stored-images"` database then `imagesStore.remove()` removes each image from the filtered list of images.

Note the use of `URL.revokeObjectURL()` to explicitly revoke the blob URL. This enables the garbage collector to free the memory allocated to the URL. If this is not done, the memory will not get returned until the page on which it was created is closed. If the URL was created in an extension's background page, this is not unloaded until the extension is disabled, uninstalled, or reloaded, so holding this memory unnecessarily could affect browser performance. If the URL is created in an extension's page (new tab, popup, or sidebar) the memory is released when the page is closed, but it is still a good practice to revoke the URL when it is no longer needed.

Once the blob URL has been revoked, any attempt to load it will result in an error. For example, if the blob URL was used as the `SRC` attribute of an `IMG` tag, the image will not load and will not be visible. It is therefore good practice to remove any revoked blob URLs from generated HTML elements when the blob URL is revoked.

Example: Store Collected Images API References: idb-file-storage library

Note: You can also use the full Web IndexedDB API to store data from your extension. This can be useful where you need to store data that isn't handled well by the simple key/value pairs offered by the DOM Storage API.

Process files in a local app

Where you have a native app or want to deliver additional native features for file processing, use native messaging to pass a file to a native app for processing.

You have two options:

Connection-based messaging

Here you trigger the process with `runtime.connectNative()` , which returns a `runtime.Port` object. You can then pass a JSON message to the native application using the `postMessage()` function of `Port` . Using the `onMessage.addListener()` function of `Port` you can listen for messages from the native application. The native application is opened if it is not running when `runtime.connectNative()` is called and the application remains running until the extension calls `Port.disconnect()` or the page that connected to it is closed.

Connectionless messaging

Here you use `runtime.sendNativeMessage()` to send a JSON message to a new, temporary instance of the native application. The browser closes the native application after receiving any message back from the native application.

To add the file or blob you want the native application to process use `JSON.stringify()` .

To use this method the extension must request the `"nativeMessaging"` permission or optional permission in its `manifest.json` file. Where optional permission is used, remember to check that permission has being granted and where necessary request permission from the user with the `permissions` API. Reciprocally, the native application must grant permission for the extension by including its ID in the `"allowed_extensions"` field of the app manifest.

Example: Native Messaging (illustrates simple messaging only) Guides: Native messaging API references: runtime API