

OS Integration



- [Working with the file system](#)
 - [Opening a file](#)
 - [Saving changes](#)
 - [File handling](#)
 - [URL handling](#)
 - [URL protocol handling](#)
 - [Calling other apps](#)
 - [Web Share](#)
 - [Web Share Target](#)
 - [Contact Picker](#)
 - [Resources](#)
-

Web apps have a big reach. They run on multiple platforms. They are easy to share via links. But traditionally they lacked integration with the operating system. Not long ago they were not even installable. Luckily that has changed and now we can take advantage of that integration to add useful features to our PWAs. Let's explore some of those options.

Working with the file system

A typical user workflow using files looks like this:

- Pick a file or folder from the device and open it directly.
- Make changes to those files or folders, and save the changes back directly.
- Make new files and folders.

Before the [File System Access API](#), web apps couldn't do this. Opening files required a file upload, saving changes required users to download them, and the web had no access at all to make new files and folders in the user's filesystem.

Opening a file

To open a file we use the `window.showOpenFilePicker()` method. Note that this method requires a user gesture, such as a button click. Here is the rest of the setup for opening a file:

1. Capture the [file handle](#) from the file system access's file picker API. This gives you basic information about the file.
2. Using the handle's `getFile()` method, you'll get a special kind of [Blob](#) called a [File](#) that includes additional read-only properties (such as name and last modified date) about the file. Because it's a Blob, Blob methods can be called on it, such as `text()`, to get its content.

```
// Have the user select a file.
const [ handle ] = await window.showOpenFilePicker();
// Get the File object from the handle.
const file = await handle.getFile();
// Get the file content.
// Also available, slice(), stream(), arrayBuffer()
const content = await file.text();
```

Saving changes

To save changes to a file, you also need a user gesture; then:

1. Use the file handle to create a [FileSystemWritableFileStream](#).
2. Make changes to the stream. This won't update the file in place; instead, a temporary file is typically created.
3. Finally, when you've finished making changes, you close the stream, which moves the changes from temporary to permanent.

Let's see this in code:

```
// Make a writable stream from the handle.
const writable = await handle.createWritable();
// Write the contents of the file to the stream.
await writable.write(contents);
// Close the file and write the contents to disk.
await writable.close();
```

{% Glitch 'mlearn-pwa-os-integration-file' %}

{% Aside 'caution' %} Safari implements the File System Access API, not for the public system, but as an [origin-private sandbox](#). You can only read and write files in that virtual file system using this

File handling

The File System Access API lets you open files from within your app, but what about the other way around? Users want to set their favorite app as their default to open files with. The [file handling API](#) is an experimental API that lets installed PWAs: Register as a file handler on a user's device, specifying the MIME type and file extension that your PWA supports in your web app manifest. You can specify custom file icons for your supported extensions.

Once registered, your installed PWA will show up as an option from the user's file system, allowing them to open the file directly into it. Here is an example of the manifest setup for a PWA to read text files:

```
...
"file_handlers": [
  {
    "action": "/open-file",
    "accept": {
      "text/*": [".txt"]
    }
  }
]
...
```

URL handling

With URL handling, your PWA can capture links that are part of its scope from the operating system and render them within a PWA window, instead of the default browser's tab. For example, if you receive a message linking to the PWA, or click on a deep link (a URL that points to a specific piece of content) in your PWA, the content will open in a standalone window.

This behavior is automatically available on Android when WebAPK is used, such as when users install a PWA with Chrome. It's impossible to capture URLs on PWAs installed on iOS and iPadOS from Safari.

For desktop browsers, the web browser community created a new spec. This spec is currently [experimental](#); it adds a new manifest file member: `url_handlers`. This property expects an array of origins that the PWA wants to capture. The origin of your PWA will be granted automatically, and

each other origin must accept that handling operating through a file named `web-app-origin-association`. For example, if your PWA's manifest is hosted on the web.dev, and you want to add the app.web.dev origin, it would look like this:

```
"url_handlers": [  
  {"origin": "https://app.web.dev"},  
]
```

In this case, the browser will check if a file exists at `app.web.dev/.well-known/web-app-origin-association`, accepting the URL handling from the PWA scope URL. The developer has to create this file. In the following example, the file looks like this:

```
{  
  "web_apps": [  
    {  
      "manifest": "/mypwa/app.webmanifest",  
      "details": {  
        "paths": [ "/" ]  
      }  
    }  
  ]  
}
```

{% Aside 'gotchas' %} On Android with WebAPKs, PWAs are automatically registered as handlers for the manifest's scope with an Android intent filter, and it is not possible to add more origins or scopes to it. For iOS and iPadOS, you can only handle URLs with PWAs published in the AppStore, such as those created with [PWABuilder](#). {% endAside %}

URL protocol handling

URL handling works with standard `https` protocol URLs, but it is possible to use custom URI-schemes, such as `pwa://`. In several operating systems, installed apps gain this ability by apps registering their schemes.

For PWA, this capability is enabled using the [URL protocol handler API](#), available only on desktop devices. You can only allow custom protocols for mobile devices by distributing your PWA on app stores.

{% Aside 'caution' %} For PWAs using the URL protocol handler API, the protocol name has to be one of the [safelisted schemes](#), or it must use a `web+` prefix. So, in our example, a link should point to `web+pwa://` to open our PWA. {% endAside %}

To register, you can use the [registerProtocolHandler\(\) method](#), or use the `protocol_handlers` member in your manifest, with the desired scheme and the URL you want to load in your PWA's context, such as:

```
...
{
  "protocol_handlers": [
    {
      "protocol": "web+pwa",
      "url": "/from-protocol?value=%s"
    },
  ]
}
...
```

You can route the URL `from-protocol` to the correct handler and get the query string `value` in your PWA. The `%s` is a placeholder for the escaped URL that triggered the operation, so if you have a link somewhere such as `<a href="web+pwa://testing">`, your PWA will open `/from-protocol?value=testing`.

Calling other apps

You can use URI schemes to connect to any other installed app (PWA or not) in users' devices on every platform. You just need to create a link or use `navigator.href` and point to the URI scheme you want, passing the arguments in URL-escaped form.

You can use well known standard schemes, such as `tel:` for phone calls, `mailto:` for email sending, or `sms:` for text messaging; or you can learn about other apps' URL schemes, for example from well known messaging, maps, navigation, online meetings, social networks, and app stores.

{% Aside %} Instead of using private URI schemes to communicate with messaging and social media apps, you can use the generic Web Share API {% endAside %}

Web Share

{% BrowserCompat 'api.Navigator.share' %}

With the [Web Share API](#), your PWA can send content to other installed apps in the device through the shared channel.

The API is only available on operating systems with a `share` mechanism, including Android, iOS, iPadOS, Windows, and ChromeOS. You can share an object containing:

- Text (`title` and `text` properties)
- A URL (`url` property)
- Files (`files` property).

To check if the current device can share, for simple data, like text, you check for the presence of the `navigator.share()` method, to share files you check for the presence of the `navigator.canShare()` method.

You request the share action by calling `navigator.share(objectToShare)`. That call returns a Promise that resolves with `undefined` or rejects with an exception.

```
{% Glitch 'mlearn-pwa-os-integration-web-share' %}
```

```
{% Img src="image/RK2djpBgopg9kzCyJbUSjhEGmnw1/wpB9v8QQTw4wB1iEGqvT.png",  
alt="Chrome on Android and Safari on iOS opening the Share Sheet thanks to Web Share.",  
width="800", height="661" %}
```

Web Share Target

[Web Share Target API](#) lets your PWA be a target of a share operation from another app on that device whether it is a PWA or not. Your PWA receives the data shared by another app.

It's currently available on Android with WebAPK and ChromeOS, and it works only after the user has installed your PWA. The browser registers the share target within the operating system when the app is installed.

You set up web share target in the manifest with the `share_target` member defined in the [Web Share Target draft spec](#). `share_target` is set to an object with some properties:

`action` : URL that will be loaded in a PWA window that is expected to receive the shared data.
`method` : HTTP verb method will be used for the action, such as `GET`, `POST`, or `PUT`. `enctype` : (Optional) Encoding type for the parameters, by default is `application/x-www-form-urlencoded`, but it can also be set as `multipart/form-data` for methods such as `POST`. `params` : An object that will map share data (from the keys: `title`, `text`, `url` and `files` from Web Share) to arguments that the browser will pass in the URL (on `method: 'GET'`) or in the body of the request using the selected encoding.

{% Aside %} You can parse a `POST` request server-side or within your service worker if you want to avoid a trip to the network. Check [this sample](#) for further details. {% endAside %}

For example, you can define for your PWA that you want to receive shared data (title and url only) by adding in your manifest:

```
...
"share_target": {
  "action": "/receive-share/",
  "method": "GET",
  "params": {
    "title": "shared_title",
    "url": "shared_url"
  }
}
...
```

From the previous sample, if any app in the system is sharing a URL with a title, and the user picks your PWA from the dialog, the browser will create a new navigation to your origin's `/receive-share/?shared_title=AAA&shared_url=BBB`, where AAA is the shared title, and BBB is the shared URL. You can use JavaScript to read that data from the `window.location` string by parsing it with the [URL constructor](#).

The browser will use the PWA name and icon from your manifest to feed the operating system's share entry. You can't pick a different set for that purpose.

{% Aside %} Make sure the page that handles the shared data is accessible offline. You don't want users choosing your app and finding an error because they don't have a connection at the moment. {% endAside %}

For more detailed examples and how to receive files, check [Receiving shared data with the Web Share Target API](#)

Contact Picker

{% BrowserCompat 'api.ContactsManager' %}

With the [Contact Picker API](#), you can request the device to render a native dialog with all the user's contacts so the user can choose one or more. Your PWA can then receive the data you want from those contacts.

The Contact Picker API is mainly available on mobile devices, and everything is done through the [navigator.contacts](#) interface on compatible platforms.

You can request the available properties to query with `navigator.contacts.getProperties()`, and request a single or multiple contact selection with a list of desired properties.

Some sample properties are `name`, `email`, `address`, and `tel`. When you ask the user to pick one or more contacts, you can call `navigator.contacts.select(properties)`, passing an array of properties you want to get in return.

{% Aside 'caution' %} The API doesn't guarantee that the returned objects will have all the requested properties, so you should recheck for their presence. {% endAside %}

The following sample will list the contacts received by the picker.

```
async function getContacts() {
  const properties = ['name', 'email', 'tel'];
  const options = { multiple: true };
  try {
    const contacts = await navigator.contacts.select(properties, options);
    console.log(contacts);
  } catch (ex) {
    // Handle any errors here.
  }
}
```

{% Glitch 'mlearn-pwa-os-integration-contacts' %}

Resources

- [The File System Access API: simplifying access to local files](#)
- [Let installed web applications be file handlers](#)
- [Handle files in Progressive Web Apps](#)
- [Integrate with the OS Sharing UI with the Web Share API](#)
- [Share content with other apps](#)
- [Receiving shared data with the Web Share Target API](#)
- [A contact picker for the web](#)

👤 #2

🕒 2026-10-31 22:55:46 UTC

🕒 2026-10-31 22:55:47 UTC