- How I came to Excalidraw
- Lipis on why he dedicates so much of his time to Excalidraw
- Excalidraw in action
- Working with files
  - Opening files
  - Saving files
    - The "save as" feature
- Drag and drop
- Sharing files
- Lipis on the deprecated Electron version
- File handling
- Clipboard integration
- Collaborating with others
  - Sharing a session URL
  - Live collaboration
  - Seeing collaborator statuses
- Lipis on what is holding back Excalidraw
- Tabbed application mode
- Closing


This is a write-up of my Google I/O talk. If you prefer watching it, see the video below.

https://www.youtube.com/watch?v=EK1AkxgQwro

# How I came to Excalidraw

I want to start with a story. On January 1st, 2020, Christopher Chedeau, a software engineer at Facebook, tweeted about a small drawing app he had started to work on. With this tool, you could

draw boxes and arrows that feel cartoony and hand-drawn. The next day, you could also draw ellipses and text, as well as select objects and move them around. On January 3, the app had gotten its name, Excalidraw, and, like with every good side project, buying the domain name was one of Christopher's first acts. By now, you could use colors and export the whole drawing as a PNG.

*Screenshot of the Excalidraw prototype application showing that it supported rectangles, arrows, ellipses, and text.*

On January 15, Christopher put out a blog post that drew a lot of attention on Twitter, including mine. The post started with some impressive stats:

- 12K unique active users
- 1.5K stars on GitHub
- 26 contributors

For a project that started a mere two weeks ago, that's not bad at all. But the thing that truly spiked my interest was further down in the post. Christopher wrote that he tried something new this time: *giving everyone who landed a pull request unconditional commit access.* The same day of reading the blog post, I had a pull request up that added File System Access API support to Excalidraw, fixing a feature request that someone had filed.

*Screenshot of the tweet where I announce my PR.*

My pull request was merged a day later and from thereon, I had full commit access. Needless to say, I didn't abuse my power. And nor did anybody else from the 149 contributors so far.

Today, Excalidraw is a full-fledged installable progressive web app with offline support, a stunning dark mode, and yes, the ability to open and save files thanks to the File System Access API.

*Screenshot of the Excalidraw PWA in today's state.*

# Lipis on why he dedicates so much of his time to Excalidraw

So this marks the end of my "how I came to Excalidraw" story, but before I dive into some of Excalidraw's amazing features, I have the pleasure to introduce Panayiotis. Panayiotis Lipiridis, on the Internet simply known as lipis, is the most prolific contributor to Excalidraw. I asked lipis what motivates him to dedicate so much of his time to Excalidraw:

> Like everyone else I learned about this project from Christopher's tweet. My first contribution was to add the Open Color library, the colors that are still part of Excalidraw today. As the project grew and we had quite many requests, my next big contribution was to build a backend for storing drawings so users could share them. But what really drives me to contribute is that whoever tried Excalidraw is looking to find excuses to use it again.

I fully agree with lipis. Whoever tried Excalidraw is looking to find excuses to use it again.

# Excalidraw in action

I want to show you now how you can use Excalidraw in practice. I'm not a great artist, but the Google I/O logo is simple enough, so let me give it a try. A box is the "i", a line can be the slash, and the "o" is a circle. I hold down `shift`, so I get a perfect circle. Let me move the slash a little, so it looks better. Now some color for the "i" and the "o". Blue is good. Maybe a different fill style? All solid, or cross-hatch? Nah, hachure looks great. It's not perfect, but that's the idea of Excalidraw, so let me save it.

I click the save icon and enter a file name in the file save dialog. In Chrome, a browser that supports the File System Access API, this is not a download, but a true save operation, where I can choose the location and name of the file, and where, if I make edits, I can just save them to the same file.

Let me change the logo and make the "i" red. If I now click save again, my modification is saved to the same file as before. As a proof, let me clear the canvas and reopen the file. As you can see, the modified red-blue logo is there again.

# Working with files

On browsers that currently don't support the File System Access API, each save operation is a download, so when I make changes, I end up with multiple files with an incrementing number in the filename that fill up my Downloads folder. But despite this downside, I can still save the file.

## Opening files

So what's the secret? How can opening and saving work on different browsers that may or may not support the File System Access API? Opening a file in Excalidraw happens in a function called `loadFromJSON)(`), which in turn calls a function called `fileOpen()`.

```
export const loadFromJSON = async (localAppState: AppState) => {
  const blob = await fileOpen({
    description: 'Excalidraw files',
    extensions: ['.json', '.excalidraw', '.png', '.svg'],
    mimeTypes: ['application/json', 'image/png', 'image/svg+xml'],
  });
  return loadFromBlob(blob, localAppState);
};
```

The `fileOpen()` function that comes from a small library I wrote called browser-fs-access that we use in Excalidraw. This library provides file system access through the File System Access API with a legacy fallback, so it can be used in any browser.

Let me first show you the implementation for when the API is supported. After negotiating the accepted MIME types and file extensions, the central piece is calling the File System Access API's function `showOpenFilePicker()`. This function returns an array of files or a single file, dependent on whether multiple files are selected. All that's left then is to put the file handle on the file object, so it can be retrieved again.

```
export default async (options = {}) => {
  const accept = {};
  // Not shown: deal with extensions and MIME types.
  const handleOrHandles = await window.showOpenFilePicker({
    types: [
      {
        description: options.description || '',
        accept: accept,
      },
    ],
    multiple: options.multiple || false,
  });
  const files = await Promise.all(handleOrHandles.map(getFileWithHandle));
  if (options.multiple) return files;
  return files[0];
  const getFileWithHandle = async (handle) => {
    const file = await handle.getFile();
    file.handle = handle;
    return file;
  };
};
```

The fallback implementation relies on an `input` element of type `"file"`. After the negotiation of the to-be-accepted MIME types and extensions, the next step is to programmatically click the input element so the file open dialog shows. On change, that is, when the user has selected one or multiple files, the promise resolves.

```
export default async (options = {}) => {
  return new Promise((resolve) => {
    const input = document.createElement('input');
    input.type = 'file';
    const accept = [
      ...(options.mimeTypes ? options.mimeTypes : []),
      options.extensions ? options.extensions : [],
    ].join();
    input.multiple = options.multiple || false;
    input.accept = accept || '*/*';
    input.addEventListener('change', () => {
      resolve(input.multiple ? Array.from(input.files) : input.files[0]);
    });
    input.click();
  });
};
```

# Saving files

Now to saving. In Excalidraw, saving happens in a function called `saveAsJSON()`. It first serializes the Excalidraw elements array to JSON, converts the JSON to a blob, and then calls a function called `fileSave()`. This function is likewise provided by the browser-fs-access library.

```
export const saveAsJSON = async (
  elements: readonly ExcalidrawElement[],
  appState: AppState,
) => {
  const serialized = serializeAsJSON(elements, appState);
  const blob = new Blob([serialized], {
    type: 'application/vnd.excalidraw+json',
  });
  const fileHandle = await fileSave(
    blob,
    {
      fileName: appState.name,
```

```
      description: 'Excalidraw file',
      extensions: ['.excalidraw'],
    },
    appState.fileHandle,
  );
  return { fileHandle };
};
```

Again let me first look at the implementation for browsers with File System Access API support. The first couple of lines look a little involved, but all they do is negotiate the MIME types and file extensions. When I have saved before and already have a file handle, no save dialog needs to be shown. But if this is the first save, a file dialog gets displayed and the app gets a file handle back for future use. The rest is then just writing to the file, which happens through a writable stream.

```
export default async (blob, options = {}, handle = null) => {
  options.fileName = options.fileName || 'Untitled';
  const accept = {};
  // Not shown: deal with extensions and MIME types.
  handle =
    handle ||
    (await window.showSaveFilePicker({
      suggestedName: options.fileName,
      types: [
        {
          description: options.description || '',
          accept: accept,
        },
      ],
    }));
  const writable = await handle.createWritable();
  await writable.write(blob);
  await writable.close();
  return handle;
};
```

## The "save as" feature

If I decide to ignore an already existing file handle, I can implement a "save as" feature to create a new file based on an existing file. To show this, let me open an existing file, make some modification, and then not overwrite the existing file, but create a new file by using the save-as feature. This leaves the original file intact.

The implementation for browsers that don't support the File System Access API is short, since all it does is create an anchor element with a `download` attribute whose value is the desired filename and a blob URL as its `href` attribute value.

```
export default async (blob, options = {}) => {
  const a = document.createElement('a');
  a.download = options.fileName || 'Untitled';
  a.href = URL.createObjectURL(blob);
  a.addEventListener('click', () => {
    setTimeout(() => URL.revokeObjectURL(a.href), 30 * 1000);
  });
  a.click();
};
```

The anchor element then gets clicked programmatically. To prevent memory leaks, the blob URL needs to be revoked after use. As this is just a download, no file save dialog gets shown ever, and all files land in the default `Downloads` folder.

# Drag and drop

One of my favorite system integrations on desktop is drag and drop. In Excalidraw, when I drop an `.excalidraw` file onto the application, it opens right away and I can start editing. On browsers that support the File System Access API, I can then even immediately save my changes. No need to go through a file save dialog since the required file handle has been obtained from the drag and drop operation.

The secret for making this happen is by calling `getAsFileSystemHandle()` on the data transfer item when the File System Access API is supported. I then pass this file handle to `loadFromBlob()`, which you may remember from a couple paragraphs above. So many things you can do with files: opening, saving, over-saving, dragging, dropping. My colleague Pete and I have documented all these tricks and more in our article so you can catch up in case all this went a little too fast.

```
const file = event.dataTransfer?.files[0];
if (file?.type === 'application/json' || file?.name.endsWith('.excalidraw')) {
  this.setState({ isLoading: true });
  // Provided by browser-fs-access.
  if (supported) {
    try {
      const item = event.dataTransfer.items[0];
```

```
      file as any.handle = await item as any
        .getAsFileSystemHandle();
    } catch (error) {
      console.warn(error.name, error.message);
    }
  }
  loadFromBlob(file, this.state).then(({ elements, appState }) =>
    // Load from blob
  ).catch((error) => {
    this.setState({ isLoading: false, errorMessage: error.message });
  });
}
```

# Sharing files

Another system integration currently on Android, ChromeOS, and Windows is through the Web Share Target API. Here I am in the Files app in my `Downloads` folder. I can see two files, one of them with the non-descript name `untitled` and a timestamp. To check what it contains, I click on the three dots, then share, and one of the options that appears is Excalidraw. When I tap the icon, I can then see that the file just contains the I/O logo again.

# Lipis on the deprecated Electron version

One thing you can do with files that I haven't talked about yet is doubleclick them. What typically happens when you doubleclick a file is that the app that's associated with the file's MIME type opens. For example for `.docx` this would be Microsoft Word.

Excalidraw used to have an Electron version of the app that supported such file type associations, so when you double-clicked an `.excalidraw` file, the Excalidraw Electron app would open. Lipis, whom you have already met before, was both the creator and the deprecator of Excalidraw Electron. I asked him why he felt it was possible to deprecate the Electron version:

> People have been asking for an Electron app since the beginning, mainly because they wanted to open files by double-clicking. We also intended to put the app in app stores. In parallel, someone suggested creating a PWA instead, so we just did both. Luckily we were introduced to Project Fugu APIs like file system access, clipboard access, file handling, and more. With a sole click you can install the app on your desktop or mobile, without the extra weight of Electron. It was an easy decision to deprecate the Electron version, concentrate just on the web app, and make it the best-possible PWA. On top, we're now able to publish PWAs to the Play Store and the Microsoft Store! That's huge!

One could say Excalidraw for Electron was not deprecated because Electron is bad, not at all, but because the web has become good enough. I like this!

# File handling

When I say "the web has become good enough", it's because of features like the upcoming File Handling.

This is a regular macOS Big Sur installation. Now check out what happens when I right-click an Excalidraw file. I can choose to open it with Excalidraw, the installed PWA. Of course double-clicking would work, too, it's just less dramatic to demonstrate in a screencast.

So how does this work? The first step is to make the file types my application can handle known to the operating system. I do this in a new field called `file_handlers` in the web app manifest. Its value is an array of objects with an action and an `accept` property. The action determines the URL path the operating system launches your app at and the accept object are key value pairs of MIME types and the associated file extensions.

```
{
  "name": "Excalidraw",
  "description": "Excalidraw is a whiteboard tool...",
  "start_url": "/",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff",
  "file_handlers": [
    {
      "action": "/",
      "accept": {
        "application/vnd.excalidraw+json": [".excalidraw"]
```

```
      }
    }
  ]
}
```

The next step is to handle the file when the application launches. This happens in the `launchQueue` interface where I need to set a consumer by calling, well, `setConsumer()`. The parameter to this function is an asynchronous function that receives the `launchParams`. This `launchParams` object has a field called files that gets me an array of file handles to work with. I only care for the first one and from this file handle I get a blob that I then pass to our old friend `loadFromBlob()`.

```
if ('launchQueue' in window && 'LaunchParams' in window) {
  window as any.launchQueue
    .setConsumer(async (launchParams: { files: any[] }) => {
      if (!launchParams.files.length) return;
      const fileHandle = launchParams.files[0];
      const blob: Blob = await fileHandle.getFile();
      blob.handle = fileHandle;
      loadFromBlob(blob, this.state).then(({ elements, appState }) =>
        // Initialize app state.
      ).catch((error) => {
        this.setState({ isLoading: false, errorMessage: error.message });
      });
    });
}
```

Again, if this went too fast, you can read more about the File Handling API in my article. You can enable file handling by setting the experimental web platform features flag. It's scheduled to land in Chrome later this year.

# Clipboard integration

Another cool feature of Excalidraw is the clipboard integration. I can copy my entire drawing or just parts of it into the clipboard, maybe adding a watermark if I feel like, and then paste it into another app. This is a web version of the Windows 95 Paint app by the way.

The way this works is surprisingly simple. All I need is the canvas as a blob, which I then write onto the clipboard by passing a one-element array with a `ClipboardItem` with the blob to the `navigator.clipboard.write()` function. For more information on what you can do with the clipboard API, See Jason's and my article.

```
export const copyCanvasToClipboardAsPng = async (canvas: HTMLCanvasElement) => {
  const blob = await canvasToBlob(canvas);
  await navigator.clipboard.write([
    new window.ClipboardItem({
      'image/png': blob,
    }),
  ]);
};

export const canvasToBlob = async (canvas: HTMLCanvasElement): Promise<Blob> => {
  return new Promise((resolve, reject) => {
    try {
      canvas.toBlob((blob) => {
        if (!blob) {
          return reject(new CanvasError(t('canvasError.canvasTooBig'),
'CANVAS_POSSIBLY_TOO_BIG'));
        }
        resolve(blob);
      });
    } catch (error) {
      reject(error);
    }
  });
};
```

# Collaborating with others

## Sharing a session URL

Did you know that Excalidraw also has a collaborative mode? Different people can work together on the same document. To start a new session, I click on the live collaboration button and then start a session. I can share the session URL with my collaborators easily thanks to the Web Share API that Excalidraw has integrated.

## Live collaboration

I have simulated a collaboration session locally by working on the Google I/O logo on my Pixelbook, my Pixel 3a phone, and my iPad Pro. You can see that changes I make on one device are reflected on all other devices.

I can even see all cursors move around. The Pixelbook's cursor moves steadily, since it's controlled by a trackpad, but the Pixel 3a phone's cursor and the iPad Pro's tablet cursor jump around, since I control these devices by tapping with my finger.

# Seeing collaborator statuses

To improve the realtime collaboration experience, there is even an idle detection system running. The cursor of the iPad Pro shows a green dot when I use it. The dot turns black when I switch to a different browser tab or app. And when I'm in the Excalidraw app, but just not doing anything, the cursor shows me as idle, symbolized by the three zZZs.

Avid readers of our publications might be inclined to think that idle detection is realized through the Idle Detection API, an early stage proposal that's been worked on in the context of Project Fugu. Spoiler alert: it's not. While we had an implementation based on this API in Excalidraw, in the end, we decided to go for a more traditional approach based on measuring pointer movement and page visibility.

*Screenshot of the Idle Detection feedback filed on the WICG Idle Detection repo.*

We filed feedback on why the Idle Detection API wasn't solving the use case we had. All Project Fugu APIs are being developed in the open, so everyone can chime in and have their voice heard!

# Lipis on what is holding back Excalidraw

Talking of which, I asked lipis one last question regarding what he thinks is missing from the web platform that holds back Excalidraw:

> The File System Access API is great, but you know what? Most files that I care about these days live in my Dropbox or Google Drive, not on my hard disk. I wish the File System Access API would include an abstraction layer for remote file systems providers like Dropbox or Google to integrate with and that developers could code against. Users could then relax and know their files are safe with the cloud provider they trust.

I fully agree with lipis, I live in the cloud, too. Here's hoping that this will be implemented soon.

# Tabbed application mode

Wow! We have seen a lot of really great API integrations in Excalidraw. File system, file handling, clipboard, web share, and web share target. But here is one more thing. Up until now, I could only ever edit one document at a given time. Not anymore. Please enjoy for the first time an early version of tabbed application mode in Excalidraw. This is how it looks.

I have an existing file open in the installed Excalidraw PWA that's running in standalone mode. Now I open a new tab in the standalone window. This is not a regular browser tab, but a PWA tab. In this new tab I can then open a secondary file, and work on them independently from the same app window.

Tabbed application mode is in its early stages and not everything is set in stone. If you're interested, be sure to read up on the current status of this feature in my article.

# Closing

To stay in the loop on this and other features, be sure to watch our Fugu API tracker. We're super excited to push the web forward and allow you to do more on the platform. Here's to an ever improving Excalidraw, and here's to all the amazing applications that you will build. Go start creating at excalidraw.com.

I can't wait to see some of the APIs that I have shown today pop up in your apps. My name is Tom, you can find me as @tomayac on Twitter and the internet in general. Thank you very much for watching, and enjoy the rest of Google I/O.

---

□□□ #3
□ □□□ □□□ 2026-10-31 21:52:17 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:08 UTC