Unblocking Clipboard Access

By Jason Miller

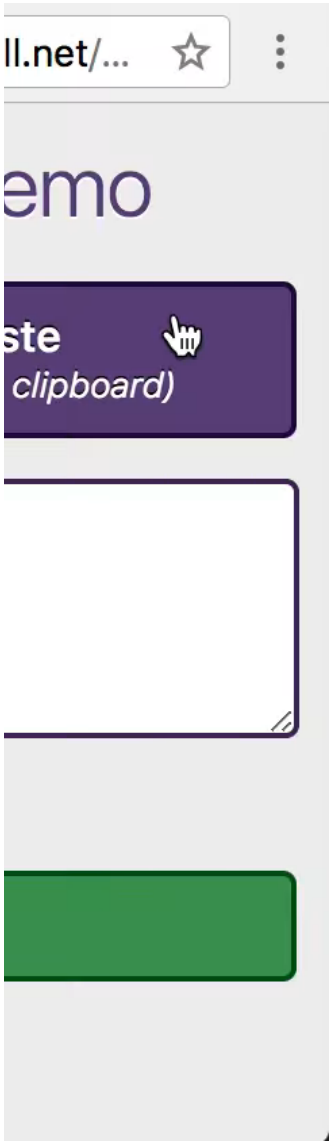
Jason is a Web DevRel

Over the past few years, browsers have converged on using `document.execCommand` for clipboard interactions. It's great to have a single widely-supported way to integrate copy and paste into web apps, but this came at a cost: clipboard access is synchronous, and can only read & write to the DOM.

Synchronous copy & paste might seem fine for small bits of text, but there are a number of cases where blocking the page for clipboard transfer leads to a poor experience. Time consuming sanitization or decoding might be needed before content can be safely pasted. The browser may need to load linked resources from a pasted document - that would block the page while waiting on the disk or network. Imagine adding permissions into the mix, requiring that the browser block the page while asking the user if an app can access the clipboard.

At the same time, the permissions put in place around `document.execCommand` for clipboard interaction are loosely defined and vary between browsers. So, what might a dedicated clipboard API look like if we wanted to address blocking and permissions problems?

That's the new Async Clipboard API, the text-focused portion of which we're shipping in Chrome 66. It's a replacement for execCommand-based copy & paste that has a well-defined permissions model and doesn't block the page. This new API also Promises (see what I did there?) to simplify clipboard events and align them with the Drag & Drop API.



0:00 / 0:32

Copy: Writing Text to the Clipboard

Text can be copied to the clipboard by calling **`writeText()`**. Since this API is asynchronous, the **`writeText()`** function returns a Promise that will be resolved or rejected depending on whether the text we passed is copied successfully:

```
navigator.clipboard.writeText('Text to be copied')
  .then(() => {
    console.log('Text copied to clipboard');
  })
  .catch(err => {
    // This can happen if the user denies clipboard permissions:
    console.error('Could not copy text: ', err);
  });
```

Similarly, we can write this as an async function, then await the return of `writeText()`:

```
async function copyPageUrl() {
  try {
    await navigator.clipboard.writeText(location.href);
    console.log('Page URL copied to clipboard');
  } catch (err) {
    console.error('Failed to copy: ', err);
  }
}
```

Paste: Reading Text from the Clipboard

Much like copy, text can be read from the clipboard by calling `readText()` and waiting for the returned Promise to resolve with the text:

```
navigator.clipboard.readText()
  .then(text => {
    console.log('Pasted content: ', text);
  })
  .catch(err => {
    console.error('Failed to read clipboard contents: ', err);
  });
```

For consistency, here's the equivalent async function:

```
async function getClipboardContents() {
  try {
    const text = await navigator.clipboard.readText();
    console.log('Pasted content: ', text);
  } catch (err) {
    console.error('Failed to read clipboard contents: ', err);
  }
}
```

Handling Paste Events

There are plans to introduce a new event for detecting clipboard changes, but for now it's best to use the "paste" event. It works nicely with the new asynchronous methods for reading clipboard text:

```
document.addEventListener('paste', event => {
  event.preventDefault();
  navigator.clipboard.readText().then(text => {
    console.log('Pasted text: ', text);
  });
});
```

Security and Permissions

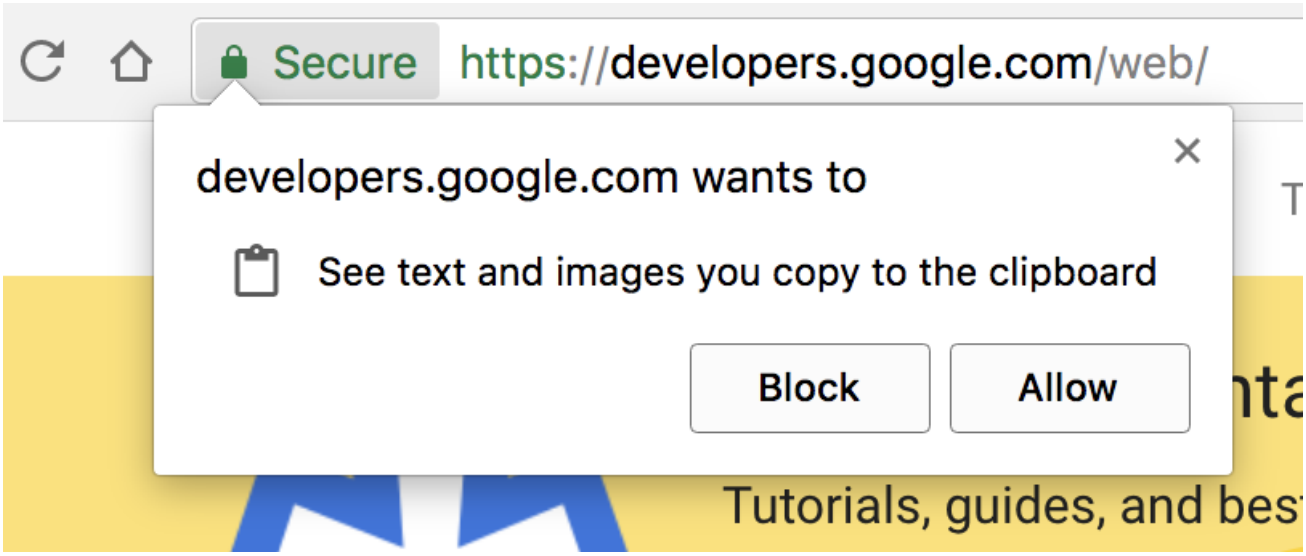
Clipboard access has always presented a security concern for browsers. Without proper permissions in place, a page could silently copy all manner of malicious content to a user's clipboard that would produce catastrophic results when pasted. Imagine a web page that silently copies `rm -rf /` or a decompression bomb image to your clipboard.

Giving web pages unfettered read access to the clipboard is even more troublesome. Users routinely copy sensitive information like passwords and personal details to the clipboard, which could then be read by any page without them ever knowing.

As with many new APIs, **`navigator.clipboard`** is only supported for pages served over HTTPS. To help prevent abuse, clipboard access is only allowed when a page is the active tab. Pages in active tabs can write to the clipboard without requesting permission, but reading from the clipboard always requires permission.

To make things easier, two new permissions for copy & paste have been added to the Permissions API. The **`clipboard-write`** permission is granted automatically to pages when they are the active tab. The **`clipboard-read`** permission must be requested, which you can do by trying to read data from the clipboard.

```
{ name: 'clipboard-read' }
{ name: 'clipboard-write' }
```



As with anything using the Permissions API, it's possible to check if your app has permission to interact with the clipboard:

```
navigator.permissions.query({
  name: 'clipboard-read'
}).then(permissionStatus => {
  // Will be 'granted', 'denied' or 'prompt':
  console.log(permissionStatus.state);

  // Listen for changes to the permission state
  permissionStatus.onchange = () => {
    console.log(permissionStatus.state);
  };
});
```

Here's where the "async" part of the Clipboard API really comes in handy though: attempting to read or write clipboard data will automatically prompt the user for permission if it hasn't already been granted. Since the API is promise-based this is completely transparent, and a user denying clipboard permission rejects the promise so the page can respond appropriately.

Since Chrome only allows clipboard access when a page is the current active tab, you'll find some of the examples here don't run quite right if pasted directly into DevTools, since DevTools itself is the active tab. There's a trick: we need to defer the clipboard access using setTimeout, then quickly click inside the page to focus it before the functions are called:

```
setTimeout(async () => {
  const text = await navigator.clipboard.readText();
  console.log(text);
}, 2000);
```

Looking Back

Prior to the introduction of the Async Clipboard API, we had a mix of different copy & paste implementations across web browsers.

In most browsers, the browser's own copy and paste can be triggered using `document.execCommand('copy')` and `document.execCommand('paste')`. If the text to be copied is a string not present in the DOM, we have to inject and select it:

```
button.addEventListener('click', e => {
  const input = document.createElement('input');
  document.body.appendChild(input);
  input.value = text;
  input.focus();
  input.select();
  const result = document.execCommand('copy');
  if (result === 'unsuccessful') {
    console.error('Failed to copy text.');
  }
})
```

Similarly, here's how you can handle pasted content in browsers that don't support the new Async Clipboard API:

```
document.addEventListener('paste', e => {
  const text = e.clipboardData.getData('text/plain');
  console.log('Got pasted text: ', text);
})
```

In Internet Explorer, we can also access the clipboard through window.clipboardData. If accessed within a user gesture such as a click event - part of asking permission responsibly - no permissions prompt is shown.

Detection and Fallback

It's a good idea to use feature detection to take advantage of Async Clipboard while still supporting all browsers. You can detect support for the Async Clipboard API by checking for the existence of `navigator.clipboard`:

```
document.addEventListener('paste', async e => {
  let text;
  if (navigator.clipboard) {
```

```
    text = await navigator.clipboard.readText()
  }
  else {
    text = e.clipboardData.getData('text/plain');
  }
  console.log('Got pasted text: ', text);
});
```

What's Next for the Async Clipboard API?

As you may have noticed, this post only covers the text part of **`navigator.clipboard`**. There are more generic **`read()`** and **`write()`** methods in the specification, but these come with additional implementation complexity and security concerns (remember those image bombs?). For now, Chrome is rolling out the simpler text parts of the API.

More Information

- Chrome Platform Status
- Example
- API
- Explainer
- Intent to Implement
- Discourse

Last updated July 2, 2018.