navigator.clipboard API Proposal (public strawman doc)

started by: lgarron@
last updated: 2016-03-09
canonical link (for Googlers): bit.ly/navigator-clipboard
crbug: 593475

See Supporting navigator.clipboard in Chrome for Chrome-specific details.

Contents

Overall TODO

TODO

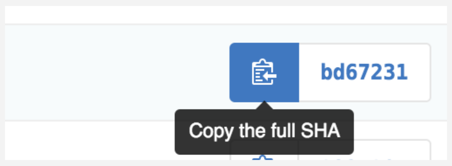
- Will `document.execCommand()` have to stick around?
- Interaction with existing baggage.
- Other MIME types (images, etc.)
- Sequence numbers
- Are there times you want to trigger a "real" paste but can't emulate it using clipboard reading?
- Talk about how you can almost shim the proposed API already (see lgarron/clipboard.js), although you still need to register a global listener for the "copy" event.
- Figure out how to read all messages in this thread.

- Interaction with the Permissions API?
- Requestion permission separately from using the API?
- Copying/pasting “files” to/from the OS (similar to drag/drop considerations?)
- If we implement the paste API, could Google Docs stop shipping as an extension and instead just ship with built-in clipboard permission?
- Frames should not have access to the clipboard.
- Can we avoid three permissions? (write, read, listen)
- Listen: only allow on desktop?
- Extensions

Disclaimer

As of right now, this is a very basic strawman by Lucas Garron, whose main relevance so far is that he is the the engineer in charge of the security UI review for the clipboard API in Google Chrome. It does **not** (yet?) reflect any intent to implement this in Chrome, and still needs lots of feedback from stakeholders before anyone should even consider it.

Motivation



Example from github.com

Web apps commonly offer a convenient “copy to clipboard” button. A few apps built on the web platform also have advanced use cases that benefit from more clipboard access. In particular:

- Some apps want to be able to copy a selection without requiring specific user interactions to copy/select the content.
- Some apps (e.g. rich document editors) want to read the clipboard without requiring the user to initiate an OS “paste” action every time.
- Some apps (e.g. remote desktop) want to receive updates about clipboard changes.

History

Since the introduction of `designMode`, there has been a web specification that supports clipboard access: `document.execCommand()`. This API was originally not available to ordinary web pages in most browsers, due to concerns about clobbering the clipboard and sniffing clipboard content[1].

Thus, the only reliable way to implement this feature across browsers until 2015 was through Flash (e.g. ZeroClipboard). Anecdotally, for some sites this was the last remaining use of Flash that did not have an open web API alternative[2].

However, as of 2015 all major browsers support `document.execCommand("copy")`:

- Internet Explorer 9 (2011-04-14)
- Chrome 42 (2015-04-14; email thread, bug)
- Opera 29 (2015-04-28)
- Firefox 41 (2015-09-22)
- Safari Technology Preview (2016-03-30; announcement)

Internet Explorer allows `"paste"` as well as `"copy"` for plain text, but shows a permission prompt to the user for both.
All other browsers allow copying `"text/plain"` and/or `"text/html"` upon user gesture (without a user prompt).
Also, `"cut"` is currently supported everywhere `"copy"` is supported.

There is work on a Clipboard API specification by Hallvord R. M. Steen of Mozilla. However, this API is strongly rooted in the assumption that it should be based on `document.execCommand()`, and Hallvord has even started a thread questioning whether this is appropriate in the long term.

Regarding potential for abuse, see Appendix C.

Benefits of the proposed API

Here are the main benefits over this proposal over the existing `document.execCommand("copy")` technique:

- **Asynchronous**: navigator.clipboard can use the same Promise-style API as other powerful web features. This:
  - Allows user agents to show a **permission prompt** without blocking the page in the same way as other APIs.
  - Allows **sanitizing security-sensitive data types** without blocking the main page.

[1] Wording taken from this Blink intent.
[2] citation needed

        - In particular, one main reason Chrome does not support images using the existing API is because of a desire to transcode any images written to/from the clipboard in order to guard against exploits in external parsers.
- **Easy to use**: `document.execCommand()` has a lot of baggage from its `designMode` origins, and is difficult to use correctly (see Appendix B for more).
- `The navigator` object is available from workers, which means this API is not artificially restricted to the UI thread.

Proposal

Introduce a new object: `navigator.clipboard`

For now, this proposal:
- ignores the existing `document.execCommand()/document.addEventListener("copy")` approach,
- ignores that the `Clipboard.clipboardData` type exists, and
- attempts to suggest an API "as if we were introducing it from scratch".

TODO
- Be pragmatic and use `Clipboard.clipboardData` in the APIs.

Type Definitions

```
/*
 * @enum {string}
 */
MIMETypeString = ["text/plain", "text/html", ...]


/** @typedef {Object<MIMETypeString, *>} */
MIMETypeObject;
```

TODO
- Instead of `MIMETypeObject`, we should actually use DataTransferItems (see Hallvord's example) or ClipboardEvent.clipboardData. However, the basic goal remains to map MIME types to values.

`navigator.clipboard.write()`

JSDoc:

```
/**
 * @param {MIMETypeObject} content
 * @returns {!Promise<>}
 */
function navigator.clipboard.write(content) {}
```

Examples:

```
// Markup with specified MIME type.
navigator.clipboard.write({
  “text/html”: “<b>Howdy</b>, partner!”
});

// Multiple MIME types.
navigator.clipboard.write({
  “text/plain”: “Howdy, partner!”,
  “text/html”: “<b>Howdy</b>, partner!”
});

// Use the Promise outcome to perform an action.
navigator.clipboard.write(“text”).then(function() {
  console.log(“Copied successfully!”);
}, function() {
  console.error(“Unable to write. :-(”);
});
```

TODO

- Be pragmatic and use `Clipboard.clipboardData` in the APIs.
- Also provide convenience functions like `navigator.clipboard.writeText(/*string*/)`, `navigator.clipboard.writeElement(/*DOMElement*/)`, or a way to write the selection?

`navigator.clipboard.read()`

JSDoc:

```
/**
 * @returns {!Promise<MIMETypeObject>}
 */
function navigator.clipboard.read() {}

/** @typedef {Object<string, *>} */
MIMETypeObject;
```

TODO

- Is it possible to make the argument type simpler for the “paste plain text” case?
- What if the only MIME type is `text/html` and you want plain text? Should the UA convert to plain text? Which ones already do?

Examples:

```
// Common use case: pasting a string
navigator.clipboard.read().then(function(clipboardData) {
  if (“text/plain” in clipboardData) {
    console.log(“Your string:”, clipboardData[“text/plain”])
  } else {
    console.error(“No string for you!”);
  }
})

// General case: pasting a string
navigator.clipboard.read().then(function(clipboardData) {
  // Do stuff with clipboardData
})
```

`navigator.clipboard.addEventListener(“change”)`

Usage:

```
/**
 * @param {ClipboardEvent}
 */
function listener(clipboardEvent) {
  // Do stuff with clipboardEvent.clipboardData
}

navigator.clipboard.addEventListener(“change”, listener);
```

TODO

- Attach `clipboardChange` to document instead?

Appendix A: “Simple” code to copy a string

Suppose you want a simple `copyToClipboard(stringMessage)` function without manipulating the DOM or modifying the document selection. This is fairly minimal “safe” implementation that works in today’s browsers.

```
var copyToClipboard = (function() {
  var _dataString = null;
  document.addEventListener("copy", function(e){
    if (_dataString !== null) {
      try {
        e.clipboardData.setData("text/plain", _dataString);
        e.preventDefault();
      } finally {
        _dataString = null;
      }
    }
  });
  return function(data) {
    _dataString = data;
    document.execCommand("copy");
  };
})();
```

TODO

- Add the simplest way using DOM modification.

Appendix B: Why not use document.execCommand()?

Although the web thrives by building on existing features (even if they are imperfect), `document.execCommand()` has a lot of historical baggage.

Here is how to use `"copy"`:

```
document.addEventListener("copy", function(event) {
  // Approach 1: set individual values
  event.clipboardData.setData(“text/plain”, “some text”);

  // Approach 2: modify the document selection

  // Bookkeeping
  event.preventDefault();
}

document.execCommand("copy")
```

Paste:

```
document.addEventListener("paste", function(event) {
  // read event.clipboardData
}
document.execCommand("paste")
```

Points:

- `document.execCommand()` is **synchronous**. This makes it tricky for user agents to ask for user permission to execute a command.
    - The event listener synchronously fires before control returns to the callsite of `document.execCommand()`, but this is not completely obvious until you try to work with them.
    - Image encoding/decoding issues (via @dcheng):
        - When pasting, we transcode the image from bitmap to PNG. This takes time, but the way the API is written forces us to do this synchronously and block the main thread. The DataTransferItem interface has a getAsFile() member, but it's not possible to construct a File object of unknown size. Thus we have to block on the image transcode before we can return the File object. This is extremely noticeable when pasting large images.
        - When copying, we need to decode the image as well. We could offload the decode to another thread and unblock the main thread but...
        - If you immediately execCommand("paste") after execCommand("copy"), you would expect the image you just copied to the clipboard to be there. But it might not be if the image decode isn't done. So we'd end up blocking the main thread on image decode on image copy too (!!!)
- In order to modify the clipboard, use of `document.execCommand(“copy”)` also requires **intercepting a “copy” event** on the document (or the relevant part of the DOM) that you have to register ahead of time using `document.addEventListener("copy”)`.
    - You have to call one function (`execCommand`) to trigger copy event, but you have to use **another function (the listener) to modify the clipboard**. If you want to be able to copy a dynamic value, the listener needs to be able to reference it dynamically. Even the common case of trying to copy a string “safely” becomes complicated. See below.

    - Since the event **also fires for user-initiated copy events**, this makes it easy for a web developer to introduce unintended side effects.
    - If you use multiple libraries that try to do anything with the clipboard (not unreasonable in the near future, given client-side packaging trends), you can end up with **conflicting listeners**.
    - Similar for paste.
- In practice, `document.execCommand("copy")` **requires a library** to use "properly" (see lgarron/clipboard.js and zenorocha/clipboard.js). It is better to give modern web apps the right tools out of the box.
    - zenorocha/clipboard.js is less than 5 months old and has 11,000 stars on GitHub. The library also provides some declarative conveniences, but its popularity is a strong indicator that developers want more a more convenient way to use the clipboard API.
- In order to copy part of the DOM using `document.execCommand("copy")`, you have to **change the current selection**.
- `document.execCommand("copy")` was introduced as an incidental part of `designMode`, even though none of the browsers that support `document.execCommand("copy")` require any part of the document to be in `designMode` in order to call it. Therefore, the use of `document.execCommand("copy")` has **compromises based on a 10-year old API** designed for editing the DOM. It is natural to edit the DOM/selection with some copy operations, but this is not necessary for most use cases.
- `document.execCommand()` **has many commands unrelated to copying**, with various interoperability bugs, and using it to control the clipboard is awkward (more below). This doesn't necessarily impact the usability of the copy command, but it would be simpler if a web developer only needed to reference a straightforward API dedicated to what they need.
- Conceptually, the clipboard is not a property of the document. It makes more sense to consider it a property of the `navigator`.
    - `window` might make sense, but workers don't have a `window` global. Regardless of whether it seems wise at the moment to give workers access to the clipboard, this problem has hit in the past with window.crypto.
- The "Clipboard API" spec is still a draft (strongly based on `designMode` roots). Note that contains the definition of the `Clipboard.clipboardData` type.
- Unless we want `document.execCommand()` to stay the recommended API forever, the best time to introduce a change is **before starting supporting paste on the open web**.
- There is currently no way to register a listener for clipboard changes. In order to support this, we would require a spec change anyhow.
- The latest execCommand spec has the following strongly worded **warning**:

"

WARNING

This spec is incomplete and it is not expected that it will advance beyond draft status. Authors should not use most of these features directly, but instead use JavaScript editing libraries. The features described in this document are not implemented consistently or fully by user agents, and it is not expected that this will change in the foreseeable future. There is currently no alternative to some execCommand actions related to clipboard content and contentEditable=true is often used to draw the caret and move the caret in the block direction as well as a few minor subpoints. This spec is to meant to help implementations in standardizing these existing features. It is predicted that in the future both specs will be replaced by Content Editable Events and Input Events.

"

Appendix C: Abuse

Even though the existing API on the open web easily allows any site on the web to overwrite the clipboard, there does not appear to be significant abuse.

Also note that **even without `document.execCommand("copy")`** it has long been possible to intercept a user-initiated copy command and modify the content before it reaches the clipboard:

```
document.addEventListener("copy", function(e) {
  // modify the document selection or call e.clipboardData.setData()
}
```

Anecdotally, this is also not abused much. The main "arguably user-hostile" use of this technique is to insert links and copyright notices into text that the user copies from a page[3].

[3] citation needed