Async Clipboard, pt III

Images and Delayed Data Generation (cont.)

Status: Draft (5 March 2019)

{garykac, huangdarwin}@chromium/google

This is a follow-up to the discussions in part II.
Simple mock implementation: jsfiddle

Background

For general background about the new Async Clipboard API, see the previous Async Clipboard API explainer.

Goals & Non-goals

Goals:

- Determine how to represent data on clipboard
  - What is the correct API "shape"
  - Streamline the common case
  - Support the complex cases (e.g., delayed)
- Image support
  - How is image data specified
  - How can we do this safely/securely: sanitizing
- Still bitmap images only
  - Initially PNG, JPG. No animated or video format.
- Determine how to support delayed generation of clipboard content
  - Important because this will likely affect the API method signatures

Non-Goals:

- Custom data types and non-still-bitmap-images
  - *(Longer term, we believe this can be done safely/securely via pickling)*
- Raw image support:
  - Copy image to system clipboard without sanitization
  - *(Longer term, we believe this can be done safely/securely via pickling)*
- Anything unrelated to Clipboard

Common use cases

- Read/write images from clipboard
  - Scenario: A PNG image is copied onto the clipboard. The UA transcodes the image before copying it to the system clipboard.
- Delayed Generation: Write a placeholder to the clipboard with a callback that is called only if that clipboard item is read.
  - Scenario: CAD program wants to put data onto the clipboard in N common formats, but they are expensive to calculate. Placeholders are placed on the clipboard so that the format is computed only if the user requests it.
- Write data to clipboard that should be pasted as an attachment.
  - Scenario: A chunk of HTML is placed on the clipboard. If this is pasted into an email message, it should be added as an attachment rather than pasted inline.

General Clipboard Architecture

Clipboard

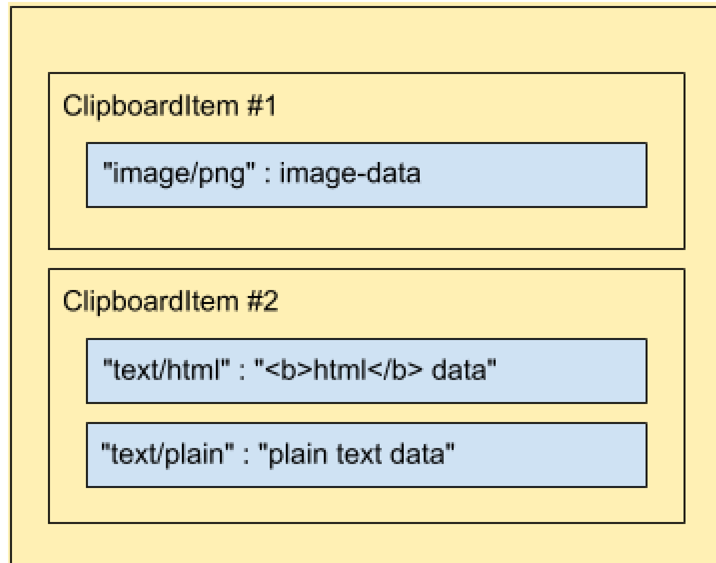
```
[SecureContext, Exposed=Window]
partial interface Clipboard : EventTarget {
  Promise<sequence<ClipboardItem>> read();
  Promise<void> write(sequence<ClipboardItem> data);
};
```

ClipboardItem

A clipboard is represented as a sequence of **`ClipboardItem`**s.

Each **`ClipboardItem`** has a dictionary of *representations*. In this dictionary, each item is keyed off the mimetype and the value is a **`ClipboardItemData`**.

Individual representations are not exposed to the user as an object type.




```
typedef (DOMString or Blob) ClipboardItemDataType;

typedef Promise<ClipboardItemDataType> ClipboardItemData;

callback ClipboardItemDelayedCallback = ClipboardItemData ();

[Constructor(record<DOMstring, ClipboardItemData> items,
             optional ClipboardItemOptions options),
  Exposed=Window]

interface ClipboardItem {
  static ClipboardItem createDelayed(
      record<DOMString, ClipboardItemDelayedCallback> items,
      optional ClipboardItemOptions options);

  readonly attribute PresentationStyle presentationStyle;
  readonly attribute long long lastModified;
  readonly attribute boolean delayed;

  readonly attribute FrozenArray<DOMString> types;

  Promise<Blob> getType(DOMString type);
};

enum PresentationStyle { "unspecified", "inline", "attachment" };

dictionary ClipboardItemOptions {
```

```
  PresentationStyle presentationStyle = "unspecified";
};
```

Reading

The read() method returns a sequence of **`ClipboardItem`**.

Code:

```
const items = await navigator.clipboard.read();
for (const item of items) {
  var types = item.getTypes();
  if (types.includes("text/plain")) {
    const textBlob = await item.getType(text/plain");
    var text = await (new Response(textBlob)).text();
    ...
  }
}
```

Writing

Code (general case):

```
const item1 = new ClipboardItem({
    "text/plain;charset=utf-8": "data",
    "text/html": "<b>data</b>",
  }
);
const imageData = new ArrayBuffer(4096);
// ... Fill |imageData| with image data.
const item2 = new ClipboardItem({
    "image/png": new Blob([imageData], {type: "image/png"}),
    "text/plain": "An image of a blob",
  }
);
await navigator.clipboard.write([item1, item2]);
```

Different ways of writing data

Writing immediate data:

```
new ClipboardItem({"text/plain": "data"})
```

Writing with a Promise:

```
new ClipboardItem({
  "text/plain":
    fetch("https://somewhere.example/").then(res => res.blob())
} )
```

Promise is triggered immediately, but the data will be returned async.

Writing delayed data

For items that are expensive to calculate, it can be useful to delay the actual generation of the data until it is requested. This allows the cost of the data generation to be avoided if the item is never read from the clipboard.

There is a static method for creating delayed data items.

```
ClipboardItem.createDelayed({
  "text/plain":
    () => { return(new Blob(["delayed data"], {type: "text/plain"})); }
})
```

The function will not be called until the data is requested by the user. It will be returned in a Promise at that time.

Specifying the item presentation style

Some clipboards can distinguish between attachments and inline data.

For example, when HTML is pasted into an email message this can be done as either an attachment ("File"), or the HTML can be pasted directly into the message.

To support this, there is a `presentationStyle` attribute on `ClipboardItem`.

Writing:

```
new ClipboardItem({
    "text/html": new Blob(["<b>data</b>"], {type: "text/html"})
  }, {
    presentationStyle: "attachment"
  } )
```

Reading:

```
const items = await navigator.clipboard.read();
const presentation = items[0].presentationStyle;
```

Blobs

Blobs are a convenient way to specify data, and are necessary to support more general data types being placed on the clipboard.

Issues:

- Must verify that the blob type matches the clipboard item's type.

Text Encoding

A `ClipboardItem` may have only one "text/plain" representation. More generally, there can be only one representation for each type (mimetype params are ignored for this comparison).

For `ClipboardItem.types`, the `type` should only be the type/subtype -- no params (like "charset") should be present. To get the full mimetype with params, you need to look at the mimetype of the Blob.

- TODO: Return an error if "text/plain" is specified with a charset param? Or strip the params as long as the type matches the Blob's type?

`ClipboardItem.types` will return only the type/subtype for each representation. No params will be included. This is necessary so that scanning the types for "text/plain" works as expected.

The Blob that is created by `writeText()` should have a mimetype of "text/plain;charset=utf-8".

Issues:

- Should `readText()` and `writeText()` use USVString instead of DOMString?

Supported image types

Initially, only "image/png", "image/jpg", and "image/jpeg".

Image Sanitizing

When copying an image onto the system clipboard, a User Agent may choose to sanitize the image. Note that, while UAs should make a best effort to preserve the image integrity, this sanitization process may result in the pixel data and/or the metadata being changed.

The actual details of the sanitization are left up to the User Agent, but a common approach is to *transcode* the image by decoding and then re-encoding the image data.

Motivation:
The primary motivation for sanitization is to prevent malicious images (ie, ones that can exploit bugs in the native OS image decoder) from being placed on the system clipboard. See, for example, this recent example from Feb 2019:
https://source.android.com/security/bulletin/2019-02-01.html

Issues:
- Is it even possible to sanitize other media types: svg, video, audio, …?

Privacy / Security

See original explainer for discussion.

Concerns WRT images are security-related and are mitigated by sanitizing the image before copying to system clipboard.

Threading

The Async Clipboard should not block the main thread and potentially cause jank. Therefore, its code should be run on the kUserInteraction task runner, and optional decoding should be run on background task runners.

Alternate Designs Considered

DataTransfer

Initially, we considered using DataTransfer since (a) it is used by the existing Event-based clipboard API, and (b) it already exists and we'd rather not add unnecessary types unless necessary.

DataTransfer was created originally for Drag and Drop and then subsequently used for the Event-based Clipboard API - primarily because it provided a basic dictionary of { mimetype -> data }. DataTransfer also includes a set of DnD-specific attributes (e.g., dropEffect, effectAllowed, dragImage) that are not used by Clipboard.

We determined that DataTransfer is not a good choice because:
- Doesn't support required features: e.g., delayed data and representationStyle
- Lacks modern accoutrements:
    - Lack of async Promise support
    - With DataTransferItemList, it curiously defines its own Array interface rather than relying on standard data structures.

- More complex construction
    - Using a standard dictionary is more concise to construct inline
    - Cf. DataTransferItemList.add()

Updating DataTransfer was considered but it has the following risks:

- Risks breaking Drag and Drop
    - And/or, pulls updating DnD into scope for fixing Clipboard

Other than the dictionary { mimetype -> data }, DnD and Clipboard are very different APIs. DnD is primarily a set of UI events whereas Clipboard accesses a shared system resource (with all the permission/privacy/security issues that come along with that).

Our primary goal is to have a clean, modern Clipboard API. While fixing DnD would be convenient, it's worth noting that the current DnD API basically works (even though there could be improvements) whereas a proper Clipboard API is currently non-existent (and is one of the most requested features from users/developers).

Alternate API shape

To allow options on each item representation, we considered variants of the following:

```
const r1 = new ClipboardItemRepresentation("text/plain", "text data");
const r2 = new ClipboardItemRepresentation("text/html", "<b>data</b>");
const item1 = new ClipboardItem([r1, r2]);
await navigator.clipboard.write([item1]);
```

Because that would permit us to add an optional `options` argument where `delayed` or `presentationStyle` attributes could be specified at the representation level. The downside of this approach is that it greatly complicates the simple use case since you need to build up these Representations.

After considering this further (and getting comments from interested parties), the "cost" of the more cumbersome API is not worth the flexibility of being able to specify these attributes at the Representation level (for comparison, the proposed API allows these values to be specified at the ClipboardItem level). The only functionality lost is the ability to specify some representations as delayed while others are immediate (or async), which is a good trade-off for a much simpler API.

Functions

We also considered using data that is specified as a `Function` to implicitly indicate delayed content generation:

```
const item = new ClipboardItem({
    "text/plain": () => {
```

```
            return("Delayed data");
        }
    });
```

But a preference was expressed to make this explicit:

```
    const item = new ClipboardItem({
        "text/plain": () => {
            return("Delayed data");
        }
    }, {
        "delayed": true,
    });
```

Which raises the question of how we should handle `Functions` if delayed = false.

In addition, there is an additional concern with how supporting `Functions` (alongside `DOMString` and `Blob`) complicates the WebIDL for the API .

In the end, it was suggested that, since this is an "advanced" feature, we could have a static method to handle delayed items:

```
    const item = ClipboardItem.createDelayed({ … })
```

where the data must be specified as a Function.