Notes

> This file is a catch-all for ideas, problems, plans, and other miscellaneous notes. If something should instead be an issue, it should be made an issue.
> This started out as a file under source control, but I think it will be more useful as a wiki page.

Other tools/libs not used, but to keep an eye on

- Markdown-to-HTML:
  - evilstreak / markdown-js (There's currently a GFM dialect that seems almost finished. See issue #41.)
  - Pagedown
  - isaacs / github-flavored-markdown (A Showdown derivative)
  - Showdown
- HTML-to-Markdown
  - domchristie / to-markdown
  - leeoniya / reMarked.js
  - Dr. Sax
  - html.md
  - upndown
- CSSUtilities: "... is a JavaScript library that provides unique and indispensible methods for querying CSS style sheets!"
  - Could maybe be used for better make-styles-explicit.
- Rangy, a "cross-browser JavaScript range and selection library".

Extension development

MDN: Performance
MDN: Performance best practices in extensions
MDN: Appendix A: Add-on Performance

TODO: Notes about MDH performance goals and decisions. Like: - Passing big strings from contentscript to backgroundscript probably undesirable. - Loading big script files (like highlight.js) into each page probably undesirable. - I don't really have a good idea of how much JS loaded into a page is too much, etc.

Postbox

Postbox has some annoying deviations from the other platforms.

Postbox uses an annoying old version of Gecko (v7, based on `userAgent`), which means that it's not compatible with some JavaScript that can be safely used in Blink, Webkit, and Mozilla-Gecko. Conflicts found:

- According to the MDN documentation, the `whatToShow` and `filter` arguments of `createTreeWalker()` aren't used.

- There's a bug (I think) in Postbox's `Range.compareBoundaryPoints`. Details here.

Miscellaneous

- Update selection conversion screenshot to not be all about replies.
- Add a visual cue as to what action took place. Sometimes converts and reverts may be a little surprising if the user's selection is off. And sometimes their viewport won't show the entirety of what change occurred.
- Briefly highlight rendered and reverted blocks/ranges.
  - Probably use CSS transitions.
  - I started this in the `transitions` branch, but wasn't thrilled with how it worked. Might come back to it, though...
- Add telemetry/usage statistics gathering. Optional, of course. It'd be nice to know how people use MDH.
  - Hotkey vs. button vs. context menu
  - Custom CSS?
  - Custom hotkey?
- Maybe rendered output should sometimes use `<br>` instead of `<p>` elements? I don't have any solid examples, but it seems to me that some clients might choke if presented with `<p>`-centric HTML when they only edit `<br>` content. Now that we count `<br>`-vs-`<p>` during HTML-to-text conversion, maybe we could use that information to alter the output to match.
  - But it's hard to justify that kind of change with no evidence of benefit.

The limitations of email.

- Mailchimp's CSS guide
- Gmail's HTML Tag Whitelist
- CSS support in mail clients
- HTML Email Boilerplate
- MUI's email CSS

Better HTML-to-Text

[Need to flesh out these notes.]

`jsHtmlToText.js` isn't very smart. New `mdh-html-to-text.js` has access to the DOM and could check element styles and whatnot to do a better HTML-to-text conversion.

Thunderbird's nsIEditor::outputToString does a really great job with HTML-to-text. Too bad it's all C++. Links to the code anyway:

- nsIEditor::outputToString: https://developer.mozilla.org/en-US/docs/XPCOM_Interface_Reference/NsI
- http://hg.mozilla.org/mozilla-central/file/153aebb30387/content/base/src/nsHTMLContentSerializer.cpp
- http://hg.mozilla.org/mozilla-central/file/153aebb30387/content/base/src/nsXHTMLContentSerializer.cp
- http://hg.mozilla.org/mozilla-central/file/153aebb30387/editor/libeditor/text/nsPlaintextEditor.cpp#l124
- http://hg.mozilla.org/mozilla-central/file/153aebb30387/content/base/src/nsDocumentEncoder.cpp

New renderers and render targets

Description

Right now Markdown Here takes content from a rich-edit element, turns it into plaintext, renders it from Markdown into HTML, and then replaces the content of the rich-edit element with the HTML. This flow is baked into the code.

There are at least two aspects of that flow that could be customizable.

The first is the source markup language, which is currently Github-flavored Markdown. There are many other excellent choices: Textile, ReST, LaTeX, and so on. There are also other flavors of Markdown, such as MultiMarkdown.

The second aspect is the "target", by which I mean the language that the source markup is rendered into as well as the element that the rendered value is put into. The target language is currently HTML, and the target element is the `innerHTML` of a rich-compose (`contenteditable`, `designmode`) element. But there have been requests (issues 36 and 43) for those aspects to be customizable: users would like to be able to render to BBCode or HTML and have the rendered value inserted as plaintext into a `textarea`.

Work

It will require a considerable refactor to implement this, to say the least.

It'll also require some UI/UX thought and work to present this to the user in a coherent way. This includes options changes and in-app commands.

There might need to be some CSS-specific work as well, kind of how there's different CSS for syntax highlighting. Some markup languages will be special-purpose (like math stuff) and will need special/specific styles.

Maybe there should be separate projects for some of the structurally distinct components? Maybe other people could leverage, say, an email mutator extension where they just need to drop in the mutation code.

Links

- MathJax is a JS LaTeX and MathML renderer.
- Fountain is a screenplay markup language.
- StrictDown is a new Markdown variant (forked from Marked).

Advanced styling

My original intention with Markdown Here was make structurally complex emails easy to write and look pretty good, like I was getting with my `README.md` files on Github.

After seeing user Casey Watts's examples of how he uses (and styles) Markdown Here, I realized that there's another axis of functionality that MDH begins to address but could do much, much more towards: making *stylistically* complex emails easy to write, look great, and be consistent (over time and across people).

You still just write Markdown. While writing you might have your final styling in mind, but you have to make no extra effort.

As can be seen from Casey's examples, quite a lot of cool custom styling can be done – and I don't think he has pushed it nearly as far as it can be pushed. There are limits, since this still has to be accepted as an email, but I think there's a lot of room to grow.

There's also an opportunity for users to share styles and themes. Casey does a rudimentary form of this by sharing his CSS via Github with the people he works with, so they can all send consistent looking email, but there's no feature in MDH at the present that facilitates this – users are forced to copy and paste back and forth.

A small-ish point, but: There should also be the ability to fairly easily switch between themes (style sets). Users send email in different contexts and need them to look different ways.

The styling could also go beyond straight CSS: * Supporting LESS/SASS would be obviously good (I've wished for that already, while working on the default styles). * It would also be cool to provide the ability to JS pre/post-processing of the rendered output. For example, Casey wants to style paragraphs that follow a `H1` differently from ones that follow a `H2`, but I don't think there's a way to do it without JS (although I suggested a horrible hack workaround to him).

I don't have a good sense for how widespread the appeal of this might be, but I suspect that it's pretty significant. (Although likely not so much among the coder crowd that I believe makes up the current user base.)

Lots of work.

Clipboard Interaction

In a Google Docs support issue (#28), a user suggested using paste to get the rendered Markdown into a GDoc. (This might also be useful for Evernote, Wordpress, and other places where MDH doesn't quite work right.) This seemed interesting, so I did some investigation.

- Paste does work fairly well to get rendered Markdown into a Google Doc. (Although Paste vs. Paste and Match Style give different results.) Some

massaging of the Markdown (`<br>` vs. `<p>`, maybe) might improve results.

- Googling and experimentation suggests that pasting cannot be triggered from a Chrome extension. (Not to be confused with reading from the clipboard, which can be done in a background script.)
- Pasting probably *can* be triggered in an old-style (non-Jetpack) Firefox/Thunderbird extension. But I suspect it can't be done in a Jetpack extension (as with Chrome).
- (Probably also can't be done in a Safari extension.)
- A common way for websites (including Github) to support copy-to-clipboard is to use Flash. The most popular seems to be ZeroClipboard – but it doesn't magically provide pasting (even if I could figure out how to use it in an extension).
- When pasting, the `markdown-here-wrapper` is lost. This means that reverting may not be possible (at the very least, it'll require another approach).
- Copying arbitrary HTML to the clipboard is probably doable on all platforms (it is on Chrome, at least).

So, it seem that, at best, MDH could put the rendered Markdown HTML into the clipboard and then the user will have to manually paste it. This is not great, but is perhaps better than nothing.

Another really big outstanding question: Can MDH detect what's selected in the GDoc? Can it get the text? Can it change the selection? Can it delete or replace the selection? (Some simple `getSelection()` tests are *not* hopeful.) Forcing the user to first copy the target text to the clipboard might be the best/only approach.

Sample code

Chrome In `manifest.json`:

```
-  "permissions": ["contextMenus", "storage"],
+  "permissions": ["contextMenus", "storage", "clipboardRead", "clipboardWrite"],
```

In `backgroundscript.js`:

```
function writeToClipboard(str) {
  var sandbox = document.createElement('div');
  sandbox.contentEditable = true;

  // Potentially unsafe, and Mozilla review will disallow it. Use Utils.saferSetInnerHTML.
  sandbox.innerHTML = str;

  document.body.appendChild(sandbox);
```

```
  var selection = document.getSelection();
  selection.removeAllRanges();
  var range = document.createRange();
  range.selectNodeContents(sandbox);
  selection.addRange(range);

  document.execCommand('copy');

  document.body.removeChild(sandbox);
}

// `plain` is a boolean indicating whether a plaintext version of the text
// should be read from the clipboard.
function readFromClipboard(plain) {
  var sandbox = document.createElement('div');
  sandbox.contentEditable = true;
  document.body.appendChild(sandbox);

  var selection = document.getSelection();
  selection.removeAllRanges();
  var range = document.createRange();
  range.selectNodeContents(sandbox);
  selection.addRange(range);
  range.collapse();

  var result = null;
  if (document.execCommand('paste')) {
      result = plain ? sandbox.innerText : sandbox.innerHTML;
  }

  document.body.removeChild(sandbox);

  return result;
}
```

Then in the background page console you can do stuff like this (after manually copying):

```
var prefs;
// async, so do it separtely in the console.
OptionsStore.get(function(opts) {prefs=opts;});

var cb = readFromClipboard();
markdownRender(prefs, htmlToText, marked, hljs.highlight, cb, document, null);
writeToClipboard(cb);
```

```
// And then paste back into GDocs.

// Alternatively, use `readFromClipboard(true)` to get a maybe-nicer plaintext
// version of the Markdown, if `htmlToText` is misbehaving.
```

(Bug: It seems that spaces are being lost from the HTML written back into the clipboard.)