Revisiting Bibliotype

I've been meaning to update Bibliotype to more modern technologies and document the process.

The original

In 2011 Craig Mod published Bibliotype, a demo and an article in a List Apart.

The article explore the nature of physical pages and how they may translate into the digital world, making references to epub3 and iBooks (an encrypted wrapper around the WebKit rendering engine).

Craig also talks about how the notion of a page changes on the web. The content as large as the content in it.

> On the other hand, the physicality of these devices doesn't represent the full potential of content space. The screen becomes a small portal to an infinite content plane, or "infinite canvas," as so well illustrated by Scott McCloud.
>
> Craig Mod – A Simpler Page

Bibliotype is one of the first attempts at taking advantage of that innfinite canvas to publish content.

I love epub… I've created content for it and I keep in touch with people who've done massive amounts of work in the community but the fact that publishers are still creating epub2 content (as opposed to the current epub3 spec) and new specifications at W3C seem to have stalled.

But the web gives you so many more possibilities and so many ways to innovate the reading experience. That's one of the things I want to explore with Bibliotype 2: How much can we leverage new(er) web technologies to enrich the user experience without becoming burdensome to develop.

The new ideas

These are some of the ideas that I have for the new iteration:

- Clean up the structure of the file
    - Recreate the menu so it doesn't depend on positioning to make things look good
- Use Pointer Events
    - PEP as an alternative
- Store settings in the user's browser using Local Storage
- Change the CSS to use variables and a `:root` element
    - This should make the menu easier to handle
    - Use variable fonts as defaults
        - Source Sans for sans serif
        - Source Serif for serif
    - Decide on what fallbacks to provide for each
        - For browsers that don't support Variable Fonts
        - For browsers that don't support woff2
- Adding Metadata
    - Adding JSON-LD for SEO and discoverability
- Adding PWA features
    - Create a Web Manifest
    - Create a Service Worker
- Looking forward to what's next
    - Background Sync
    - Push Notifications

Clean up the structure of the content file

Rather than take the existing Bibliotype file I've decided to start from scratch and create a brand new structure that will allow for experimentation and additional work as we start working on the project.

The structure of the project looks like this (content removed for brevity)

```
<!DOCTYPE html>
```

```
<html lang="en"<html lang="en">a charset="UTF-8">
  <meta name="viewport" content="width=device-wid<meta name="viewport" co
  <title>Document</title>
  <link rel="stylesheet" href="css/master.css">
  <script defer src="js/bibliotype.js"><title></script>
</head>

<body>
  <button id="menuButton">Menu</button>
  <div id="menu">
    <h1>Menu</h1>

    <p class="desc">Size</p>
    <button id="bed">bed</button>
    <button id="knee">knee</button>
    <button id="breakfast">breakfast</button>

    <p class="desc">Justification</p>
    <button id="rag">Ragged</button>
    <button id="just">Justified</button>

    <p class="desc">Font Family</p>
    <button id="roboto">Roboto</button>
    <button id="source">Source Serif</button>

    <p class="desc">Contrast</p>
    <button class="lowc">Low</button>
    <button class="highc">High</button>

  </div>


  <div class="content">

    <div class="centered">
      <h1>Monsters of Mars</h1>
      <h2>A Complete Novelette</h2>
      <h3><em>By Edmond Hamemlton</em></h3>
```

```
    </div>

    <p>Rest of the content goes here</p>

  </div>

</body>
</html>
```

Remove dependency on jQuery and jQueryDoubleTap

In the first version of the project I'm more concerned with removing the jQUery dependecy. It would probably be easier to work with jQuery but part of the challenge is to make this work with as few dependencies as possible.

A portion of the Javascript file looks like this:

```
function setRootVar(name, value) {
  let rootStyles = document.styleSheets[0].cssRules[3].style;
  rootStyles.setProperty('--' + name, value);
};

 document.addEventListener('DOMContentLoaded', function(e) {
   const menuButton = document.getElementById('menuButton');
   menuButton.addEventListener('click', function(e) {
     setRootVar('menu-visibility', 'show');
   }, false);

   const roboto = document.getElementById('roboto');
   roboto.addEventListener('click', function(e) {
     setRootVar('body-font', 'roboto');
     setRootVar('body-backup', 'Open Sans');
     setRootVar('body-default', 'sans-serif');
   });
```

```
  const source = document.getElementById('source');
  source.addEventListener('click', function(e) {
    setRootVar('body-font', 'Source Serif');
    setRootVar('body-backup', 'Georgia');
    setRootVar('body-default', 'serif');
  });
});
```

Because we're working with CSS Variables I created a little function that will insert them into the correct rule in the stylesheet. `document.styleSheets[0].cssRules[3].style;` represents the style of the 3rd rule in the first stylesheet of the document. We'll look at that third rule in the SASS/CSS section.

In a future iteration I'm thinking about replacing the click events with Pointer Events to provide a unified interface for both touch events like those in tablets and the mouse events from our desktop environments.

Change the CSS to use variables and a :root element

Rather than change each individual item for each individual combination of attributes I have set up a :root pseudo element with sets of different variables.

The first set (starting with `--roboto`) define Open Type Features of the Roboto font. Combined with CSS classes (not shown) we can enable open type features as needed.

I got the CSS by running the variable font through Wakamaifondue.

```
:root {
  --roboto-c2sc: "c2sc" off;
  --roboto-dlig: "dlig" off;
  --roboto-dnom: "dnom" off;
  --roboto-frac: "frac" off;
  --roboto-lnum: "lnum" off;
  --roboto-numr: "numr" off;
```

```
  --roboto-onum: "onum" off;
  --roboto-pnum: "pnum" off;
  --roboto-salt: "salt" off;
  --roboto-smcp: "smcp" off;
  --roboto-ss01: "ss01" off;
  --roboto-ss02: "ss02" off;
  --roboto-ss03: "ss03" off;
  --roboto-ss04: "ss04" off;
  --roboto-ss05: "ss05" off;
  --roboto-ss06: "ss06" off;
  --roboto-ss07: "ss07" off;
  --roboto-tnum: "tnum" off;
  --roboto-unic: "unic" off;
  --roboto-cpsp: "cpsp" off;
```

The second block contains all the elements that we can manipulate through Javascript, either individually or in groups.

```
// Default Colors
--bright-bg-color: rgba(255, 255, 255, 1);
--bright-txt-color: rgba(0, 0, 0, 1);
// Font and Backup description
--body-font: "Roboto";
--body-backup: "Open Sans";
--body-default: sans-serif;
"Roboto"// Font and line-height related
--font-size: 100%;
--line-height: 1.25;
--body-font-size: 1rem;
// Font stylingweight: "wght" 400;
--font-width: "wdth" 100;
--font-italics: "slnt" 0;
// Width"wght"// Width container
--body-width: 40rem;
// Menu Visibil3ty
--menu-visibility: hidden;
// Justify?
```

```
// in left to right languages, start is
// equivalent to left
--content-justify: start;
}
```

So how do we use the CSS variables we defined?

If you look at the body declaration below you'll see how we use the variables to set their values as the attributes.

The syntax is `var(--name-of-variable)` the two dashes (`--`) are required by the specification as a way to avoid conflict with built in property declarations.

Font-familly uses three properties defined earlier to control the font stack. One of the advantages of CSS variables that is not part of SASS or other pre-processors is that changing the value of the variable will immediately cascade down and change the value everywhere it appears in the document.

This is the full declaration in the SCSS file:

```
body {
  background-color: rgba(255, 255, 255, 1);
  color: rgba(0, 0, 0, 1);
  font-size: 1rem;
  line-height: 1.25;
  font-family: Roboto, 'Open Sans', sans-serif;
  text-align: left;
  -webkit-hyphens: manual;
  -ms-hyphens: manual;
  hyphens: manual;
  transition: all linear 0.2s;
  background-color: var(--background-color);
  color: var(--text-color);
  font-family:  var(--body-font),
                var(--body-backup),
                var(--body-default);
  font-size: 1rem;
  line-height: var(--line-height);
```

```
  text-align: var(--content-justify);
  -webkit-hyphens: var(--content-hyphenate);
  -ms-hyphens: var(--content-hyphenate);
  hyphens: var(--content-hyphenate);
}
```

I know it looks scary but it's simply the values hardcoded to default values followed by the same declarations using variables where appropriate. There are a couple items I want to highlight.

I've chosen to use a fixed `1rem` size for the body font so I can keep the size of the menu consistent. If I ue the `--font-body-size` variable the menu size will grow along with the text.

If changes are going to be noticeable to the user I've used a short transition to make them less jarring. That's the `transition: all linear 0.2s;` item in the body declaration.

Media Queries, Font Sizes and Content Width

One of the few things I didn't like about Bibliotype is that it made the assumption that font size is directly related to reading distance.

I don't think this is the case.

In iBooks, for example, I like to set the font larger regardless of the distance I hold the device from my face as I read.

So how do we handle the combination of width and screen sizes?

The best solution I can come with is to split the problem into two areas. Content width and font sizes.

The content width part is easy and it involves setting the value of the `--body-width` to the desired value and hiding the menu, for now, so the text is easier to read. We will handle this via Javascript.

Media queries will handle device orientation. The basic queris look like this:

```
@media screen and (orientation: landscape) {
  // Content for landscape mode
}

@media screen and (orientation: portrait) {
  // Content for portrait mode
}
```

Inside the queries, particularly the landscape query, we can make any adjustments we need to make to keep the reading experience a pleasant one.

The Gravy (V2)

The rest of the project deals with additional “nice to have” items that will make this into a full fledged PWA.

Remaining issues with (S)CSS and Javascript

The function that makes the font size smaller doesn’t check if the font has reached a minimal size. It can be made small enough to be unreadable. We should put a check to make sure that the font size doesn’t get below a certain value.

My first pass at a fix modifies the smaller event listener to restrict the minimum font size. The idea is that if the value is smaller than a value we set (0.75 in this case) we force the value to be the value we set.

This will prevent the font from growing too small to read.

```
smaller.addEventListener('click', function(e) {
  const style = window.getComputedStyle(document.documentElement);
  const fontSize = style.getPropertyValue('--body-font-size');
  const calcFont = parseFloat(fontSize);
  if (fontSize > 0.75) {
    setRootVar('body-font-size', calcFont - 0.1);
  } else {
    setRootVar('body-font-size', 0.75);
  }
```

```
});
```

Right now the menu button is setup to show the menu and do nothing if it's already showing. We need a toggle to show and hide the menu accordingly.

```
const menuButton = document.getElementById('menuButton');
  menuButton.addEventListener('click', function(e) {
  const style = window.getComputedStyle(document.documentElement);
  const menuStatus = style.getPropertyValue('--menu-visibility');
  if (menuStatus == 'hidden') {
    setRootVar('menu-visibility', 'show');
  } else {
    setRootVar('menu-visibility', 'hidden');
  }
});
```

Store settings in the user's browser

Rather than force the user to set their settings on every visit is to store the settings on the user's browsers using Local Storage. It will save the last item clicked on each category and will then store those settings in the user's browser for retrieval on the next visit.

Getting this to work is a two-step proocess.

The first step is to modify our event listeners to create/set a key/value pair in our local storage box.

Taking the justify click event handler definition, it now looks like this.

```
  justify.addEventListener('click', function(e) {
    setRootVar('content-justify', 'justify');
    localStorage.setItem('content-justify', 'justify');
  });
```

We've added the call to `localStorage.setItem` and pass the names of the key and value as string.

The second step is more tedious. For each of the event listeners we created and each key/value pair we stored in local storage we do the following:

1. Test if the value exists by making sure it's not null. The spec says that if a we use `localStorage.getItem` in a non-existing value, the return value is null
2. Set the CSS variable using `setRootVar` with the name of the property and the return of `localStorage.getItem`

We then execute the function to make sure that it runs as soon as possible.

The example below has been stripped down to only show `content-justify` as an example, the same event listener that we looked at earlier in the section.

```
function loadSettings() {
  // content removed for brevity

  if (localStorage.getItem('content-justify') !== null) {
    setRootVar('content-jus'content-justify'lStorage.getItem('content-just
  }

  'content-justify'// More content removed
}

loadSettings();
```

What I like about this method of setting and saving preferences is the flexibility it gives us.

We don't have to modify all the settings between visits, whatever items changed since your last visit will be reflected in localStorage and will be reflected in subsequent visits, the items that have not changed will fail their respective test and not change.

One last thing that we may need is a reset button to acommodate the need for all settings to be wiped out from the user's browser.

Adding Metadata

The first thing in this section is not strictly related to PWAs but it has more to do

with search engines and discoverability.

We'll mark up the code with [JSON-LD(https://json-ld.org/) based on the article and example

```
<script type="application/ld+json"> {
  "@context": "http://schema.org",
  "@type": "Article",
  "mainEntityOfPage": {
    "@type": "WebPage",
    "@id": "https://caraya.github.io/bibliotype2/"
  },
  "headline": "Monsters of Mars",
  "alternativeHeadline": "TA Complete Novelette robots and stuff",
  "image": "https://www.gutenberg.org/files/30452/30452-h/images/image_003
  "author": {
    "@type": "Person",
    "name": "Edmond Hamilton"
  },
  "editor": {
    "@type": "Person",
    "name": "John Campbell"
  },
  "genre": "science Fiction",
  "keywords": "scifi astounding campbell",
  "wordcount": "13155",
  "publisher": {
    "@type": "Organization",
    "name": "Readers Guild, Inc",
    "logo": {
      "@type": "ImageObject",
      "url": "http://central.gutenberg.org/img/ProjectGutenberg.jpg"
    }
  },
  "url": "https://caraya.github.io/bibliotype2/",
  "datePublished": "1931-04-01",
  "dateCreated": "2018-08-25",
  "dateModified": "2018-08-25",
  "description": "Story from the April 1931 issue of Astounding Stories"
```

```
  }
  </script>
```

When the Google crawler (and any other crawler that understands JSON-LD) crawls the document it will get the additional information listed in the JSON-LD script and provide additional features for the page in the results page.

The Gravy (V3): Making a PWA Out of You

The final active development stage is to generate assets to turn the application into a PWA

Create a Web Manifest

Using a manifest generator I was able tp generate the JSON manifest itself as well as the icons necessary for the manifest.

> The JSON file is simple, it's listed below.

```
{
  "name": "Monsters "Monsters from Mars""short_name": "monsters",
  "theme_color": "#2196f3",
  "background_color": "#2196f3",
  "display": "standalone",
  "Scope": "/",
  "start_url": "/",
  "icons"  {
      "src": "ima": "icons/icon-72x72.png",
      "sizes": "": ""sizes""sizes"e": "image/png"
 ": ""type"{
      "src": "images/icons/icon": ""src"g",
      "sizes": "96x96",
      "type": ""sizes": "96x96",
      "type""src": "images/icons/icon-128x128": ""src"    "sizes": "128x12
      "type": "im": ""sizes": "128x128",
```

```
      "type": "image/png"
    }
  ],
  "splash_pages": null
}
```

Create a Service Worker

A service worker will provide a better experience for users. The idea behind our service worker is simple:

It will precache the minimum set of assets (HTML, CSS and Javascript) to display the content.

It will then cache additional assets like fonts and images using a cache first strategy; check if the resource is in the cache and if it is serve it from there but, if it's not, then fetch it from the network.

The first step is to register the service worker. This step is the same regardless of how you create the worker. It is important to notice that the script you reference here must live at the root of your app/site for it to work effectively.

```
// Check that service workers are registered
if ('serviceWorker' in navigator) {
  'serviceWorker'// Use the window load event to keep the page load perfo
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('sw.js');
  });
}
```

For the rest of the steps we'll use workbox.js, a set of libraries from Google that make it easiier to work with service workers.

Install the packge globally to make the commands available on your terminal.

```
npm i -g workbox-build
```

I've chosen the longer route that incolves my creating a pre-caching service workers with additional routes to handle special cases.

Based on the tutorials on the workbox.js site I created the followng service worker:

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/3.4.1/w

  // point where we'll mount the precache manifest files
  workbox.precaching.precacheAndRoute([]);

  workbox.routing.registerRoute(
    // Cache CSS files
    /.*\.css/,
    // Use cache but update in the background ASAP
    workbox.strategies.staleWhileRevalidate({
      // Use a custom cache name
      cacheName: 'css-cache',
    })
  );

  workbox.routing.registerRoute(
    /.*\.(?:otf|ttf|woff|woff2)/,
    workbox.strategies.cacheFirst({
      cacheName: 'fonts-cache',
      plugins: [
        new /.*\.(?:otf|ttf|woff|woff2)/workbox.expiration.Plugin({
          maxEntries: 4,
        }),
      ],
    })
  );

  workbox.routing.registerRoute(
    'css-cache'// Cache image files
    /.*\.(?:png|jpg|jpeg|svg|gif)/,
    // Use the cache if it's available
    workbox.strategies.cacheFirst({
      // Use a custom cache nameeName: 'image-cache',
```

```
      plugins: [
        new workbox.expiration.Plugin({
          // Cache on'image-cache'// Cache only 20 images
          maxEntries: 20,
          // Cache for a maximum of 30 days
          maxAgeSeconds: 30 * 24 * 60 * 60,
        }),
      ],
    })
  );
```

`workbox.precaching.precacheAndRoute([]);` is where we'll inject the assets we want to precache so it's ok that it's empty for now.

The other three routes are for additional resources matched by regular expressions.

The first one will match all CSS files that were not precached using the `/.*\.css/` regular expression. This route uses the stale while revalidate caching strategy: It's ok to use resources in the cache while the browser goes fetch an updated version in the background for the next visit.

The second route matches all major font formats (TTF, OTF, WOFF and WOFF2), puts them in a separate cache for fonts (`fonts-cache`) and restricts the number of items in the cache. This route uses a cache first strategy: Check if the resource is in the cache and, if it is, use it. If it's not in the cache then fetch it from the network and put it in the cache for later visits.

The last route matches images (PNG, JPG, SVG and GIF), puts them on an `image-cache`, restricts the number of items to 20 and expires the items after 30 days (measured in seconds). This cache also uses cache first as the images once added are not likely to change.

To generate the files to precache we run the command below from the terminal.

```
workbox wizard --injectManifest
```

The wizard will ask you questions about where you store the code that will run the

application and, based on your answer, what assets you want to precache.

For this particular version I told it I only wanted to precache CSS, Javascript and HTML. We need to be careful with what we precache and what we cache after the first load.

The command will generate a `workbox.config` file that we can tweak later. The file looks like this:

```
module.exports = {
  'globDirectory': '.',
  'globPatterns': [
    '**'.'/*.{css,html,js}',
  ],
  'globIgnores': [
    '**/node_modules/**/
    '**/sw.js',
    'workbox-config.js',
  ],
  'swDest': 'sw.js',
  'swSrc': 'js/sw.js',
};
'**/sw.js'
```

The last step is to have Workbox insert the files that match the manifest using the following command:

```
workbox injectManifest workbox-config.js
```

This will insert code like the one below into the service worker. This is the minimal setup to render the page's above the fold content (and in this case the entire page).

We cache fonts the first time the user changes them or interacts with them and they will be available locally without requiring network resources.

```
// point where we'll mount the precache manifest files
```

```
workbox.precaching.precacheAndRoute([
  {
    'url': 'css/master.css',
    'revision': '588aa989df5b9bb1f59aa6d22cba0cb6',
  },
  {
    'url': 'index.html',
    'revision': '708da8295998f6e46d9a4ecd13ae9dad',
  },
  {
    'url': 'js/bibliotype.js',
    'revision': '37efaf4d7f4bb690833bd11f0449c4af',
  },
]);
```

And that's it, we now have a service worker doing most of the work to provide better and faster user experiences for our users.

Looking forward to what's next

These are things that will take a while to work through either because the APIs are harder to work with or because I'm not certain of the status of the API and research plus implementation will take longer.

(Periodic) Sync: Serialized content

Periodic Sync will allow us to periodically sync to the application for new or updated information.

One case that quickly comes to mind is serialized content. Rather than upload an entire book or story we can create chunks and offer the first few when the user first uploads and the reminder over a period of time.

Push Notifications

At some point we'll have to use notifications to let users know changes or give them chances to opt in to new content or to new additions to the app as they are implemented.

Annotations

I've looked at Hypothes.is and AnnotatorJS but they present different sides of the same problem.

Hypothes.is offers a hosted solution that would be perfect but I'm not the one hosting it so there are privacy implications.

AnnotatorJS predates Hypothes.is but the backend situation is a little unclear from the documentation as to the storage requirements.

Once I figure out what the best backend storage solution is (maybe hosting the backed on an instance of Google App Engine and not worrying about the rest) will be a good solution.