Estimating Available Storage Space

By Jeff Posnick
Web DevRel @ Google

tl;dr

Chrome 61, with more browsers to follow, now exposes an estimate of how much storage a web app is using and how much is available via:

```
if ('storage' in navigator && 'estimate' in navigator.storage) {
  navigator.storage.estimate().then(({usage, quota}) => {
    console.log(`Using ${usage} out of ${quota} bytes.`);
  });
}
```

Modern web apps and data storage

When you think about the storage needs of a modern web application, it helps to break *what's* being stored into two categories: the core data needed to load the web application, and the data needed for meaningful user interaction once the application's loaded.

The first type of data, what's needed to load your web app, consists of HTML, JavaScript, CSS, and perhaps some images. Service workers, along with the Cache Storage API, provide the needed infrastructure for saving those core resources and then using them later to quickly load your web app, ideally bypassing the network entirely. (Tools that integrate with your web app's build process, like the new Workbox libraries or the older `sw-precache`, can fully automate the process of storing, updating, and using this type of data.)

But what about the other type of data? These are resources that aren't needed to load your web app, but which might play a crucial role in your overall user experience. If you're writing an image editing web app, for instance, you may want to save one or more local copies of an image, allowing users to switch between revisions and undo their work. Or if you're developing an offline media playback experience, saving audio or video files locally would be a critical feature. Every web app that can be personalized ends up needing to save some sort of state information. How do you know how much space is available for this type of runtime storage, and what happens when you run out of room?

The past: window.webkitStorageInfo and navigator.webkitTemporaryStorage

Browsers have historically supported this type of introspection via prefixed interfaces, like the very old (and deprecated) `window.webkitStorageInfo`, and the not-quite-as-old, but still non-standard `navigator.webkitTemporaryStorage`. While these interfaces provided useful information, they don't have have a future as web standards.

That's where the WHATWG Storage Standard enters the picture.

The future: navigator.storage

As part of the ongoing work on the Storage Living Standard, a couple of useful APIs have made it to the `StorageManager` interface, which is exposed to browsers as `navigator.storage`. Like many other newer web APIs, `navigator.storage` is only available on secure (served via HTTPS, or localhost) origins.

Last year, we introduced the `navigator.storage.persist()` method, which allows your web application to request that its storage be exempted from automatic cleanup.

It's now joined by the `navigator.storage.estimate()` method, which serves as a modern replacement for `navigator.webkitTemporaryStorage.queryUsageAndQuota()`. `estimate()` returns similar information, but it exposes a promise-based interface, which is in keeping with other modern asynchronous APIs. The promise that `estimate()` returns resolves with an object containing two properties: `usage`, representing the number of bytes currently used, and `quota`, representing the maximum bytes that can be stored by the current origin. (Like everything else related to storage, quota is applied across an entire origin.)

If a web application attempts to store—using, for example, IndexedDB or the Cache Storage API—data that's large enough to bring a given origin over its available quota, the request will fail with a `QuotaExceededError` exception.

Storage estimates in action

Exactly how you use `estimate()` depends on the type of data your app needs to store. For example, you could update a control in your interface letting users know how much space is being used after each storage operation is complete. You'd ideally then provide an interface allowing users to manually clean up data that's no longer needed. You might write code along the lines of:

```
// For a primer on async/await, see
// https://developers.google.com/web/fundamentals/getting-started/primers/async-f
async function storeDataAndUpdateUI(dataUrl) {
  // Pro-tip: The Cache Storage API is available outside of service workers!
  // See https://googlechrome.github.io/samples/service-worker/window-caches/
  const cache = await caches.open('data-cache');
  await cache.add(dataUrl);

  if ('storage' in navigator && 'estimate' in navigator.storage) {
    const {usage, quota} = await navigator.storage.estimate();
    const percentUsed = Math.round(usage / quota * 100);
    const usageInMib = Math.round(usage / (1024 * 1024));
    const quotaInMib = Math.round(quota / (1024 * 1024));

    const details = `${usageInMib} out of ${quotaInMib} MiB used (${percentUsed}%

    // This assumes there's a <span id="storageEstimate"> or similar on the page.
    document.querySelector('#storageEstimate').innerText = details;
  }
}
```

How accurate is the estimate?

It's hard to miss the fact that the data you get back from the function is just an estimate of the space an origin is using. It's right there in the function name! Neither the **usage** nor the **quota** values are intended to be stable, so it's recommended that you take the following into account:

- **usage** reflects how many bytes a given origin is effectively using for same-origin data, which in turn can be impacted by internal compression techniques, fixed-size allocation blocks that might include unused space, and the presence of "tombstone" records that might be created temporarily following a deletion. To prevent the leakage of exact size information, cross-origin, opaque resources saved locally may contribute additional padding bytes to the overall **usage** value.
- **quota** reflects the amount of space currently reserved for an origin. The value depends on some constant factors like the overall storage size, but also a number of potentially volatile factors, including the amount of storage space that's currently unused. So as other applications on a device write or delete data, the amount of space that the browser is willing to devote to your web app's origin will likely change.

The present: feature detection and fallbacks

`estimate()` is enabled by default starting in Chrome 61. Firefox is experimenting with `navigator.storage`, but, as of August 2017, it's not turned on by default. You need to enable the `dom.storageManager.enabled` preference in order to test it.

When working with functionality that isn't yet supported in all browsers, feature detection is a must. You can combine feature detection along with a promise-based wrapper on top of the older `navigator.webkitTemporaryStorage` methods to provide a consistent interface along the lines of:

```
function storageEstimateWrapper() {
  if ('storage' in navigator && 'estimate' in navigator.storage) {
    // We've got the real thing! Return its response.
    return navigator.storage.estimate();
  }

  if ('webkitTemporaryStorage' in navigator &&
      'queryUsageAndQuota' in navigator.webkitTemporaryStorage) {
    // Return a promise-based wrapper that will follow the expected interface.
    return new Promise(function(resolve, reject) {
      navigator.webkitTemporaryStorage.queryUsageAndQuota(
        function(usage, quota) {resolve({usage: usage, quota: quota})},
        reject
      );
    });
  }

  // If we can't estimate the values, return a Promise that resolves with NaN.
  return Promise.resolve({usage: NaN, quota: NaN});
}
```

Last updated July 2, 2018.