Web Apps without Web Servers

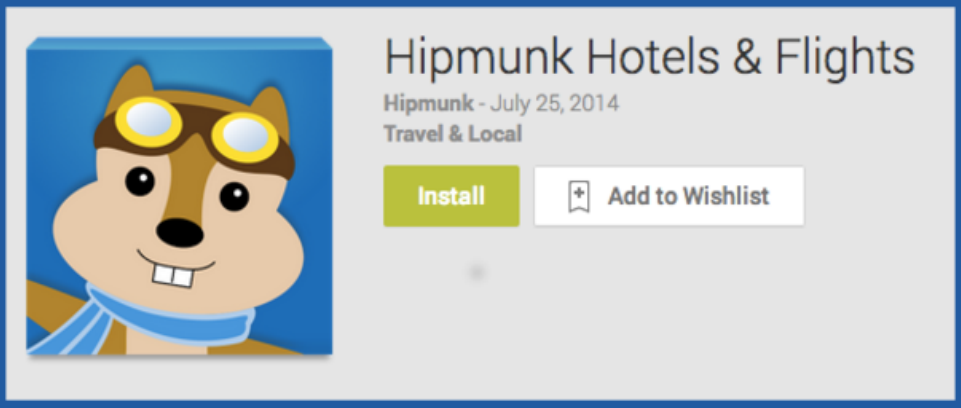
Richard Feldman

Presented at Strange Loop 2014.

Video: http://www.youtube.com/watch?v=WqV5kqaFRDU

When booking a flight, how many people use a browser compared to a native app? Ask around and you'll find that most people reach for a browser first.

Why is that? It's not price, since both are free to use, so it must to some extent be user experience. But the in-app user experiences are fairly equivalent: you enter your dates and airports, select a flight from a list, and pay.



What about the user experience that comes before you begin booking? The part between when you decide to book a flight and actually begin booking it?

Let's compare the experience of using hipmunk.com to book a flight to the same experience using the Hipmunk native app. We'll start with the native app.

Step 1: Search

As with any native app, Step 1 is to search for it on an app store.

This is much better than it used to be! Remember back before app stores, when you had to drive to a physical store to buy a disc, then take it home and put it in a drive? App stores have made this process much more seamless.

Step 1: Search
Step 2: Install

Step 2 hasn't changed much from the days before app stores; you still need to install your app.

Step 1: Search
Step 2: Install
Step 3: Run

Step 3 is to actually run the app, at which point you're off and running, booking your flight.

Step 1: Enter Name

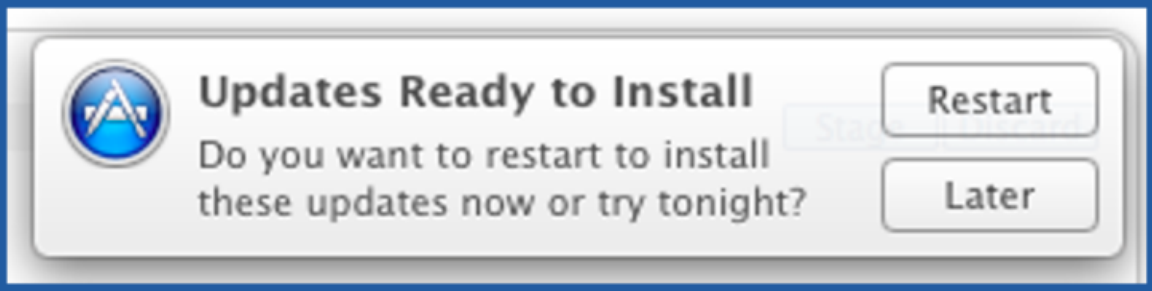
With a web app, Step 1 is that you enter the name of the web app you want to use. In this case, hipmunk.com.

Step 1: Enter Name

There is no Step 2. Once you've entered the name of the app you want to use, that's it...the next thing that happens is that you're using it.

Suppose you wanted to streamline this process for a web app. How would you do it? The only possible step to eliminate is typing in the name.

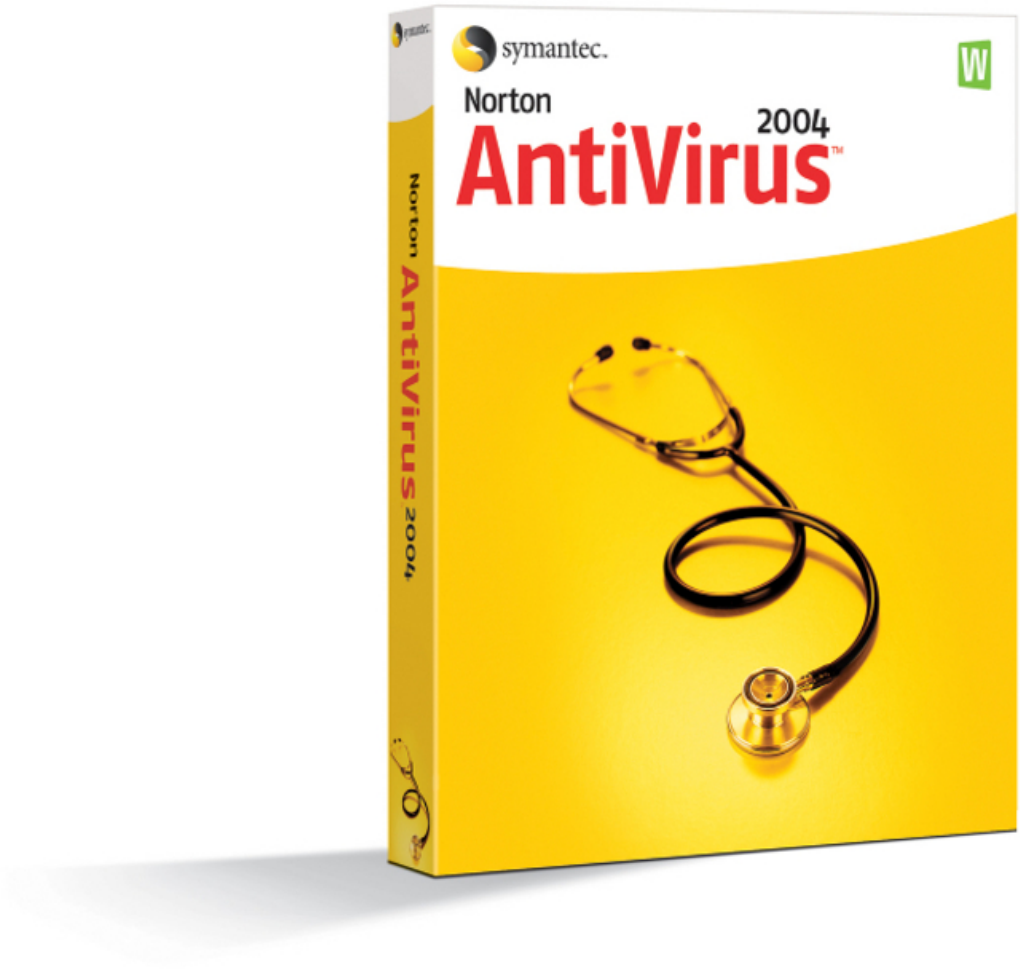
To improve it, the computer would have to determine what you wanted to use without being told. To do this accurately would essentially require reading your mind; to improve the delivery process web apps have today essentially requires science fiction.



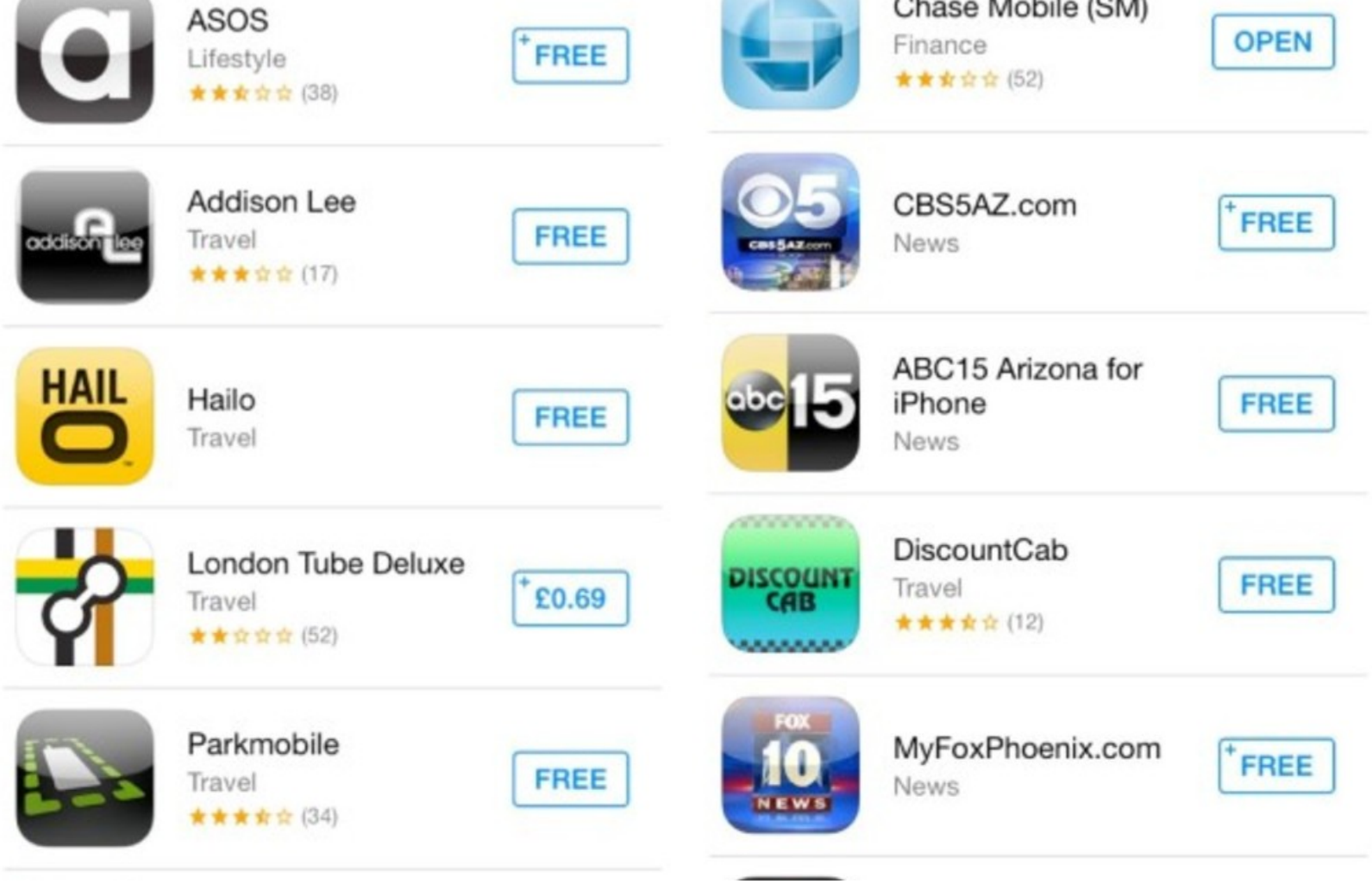
Besides delivery, web apps and native apps also offer different user experiences when it comes to updates.

In the browser, updates happen invisibly. You are never pestered to approve permissions for a web app so that it can update, or to restart a web app after it has updated.

When you enter the name of the web app you want to use, not only are you using it right away, you're also using the latest version right away.



Web apps also tend to handle security differently. They prompt for permissions on an as-needed basis rather than up front, so you can actually do a sanity check as to whether it makes sense for them to be requesting those permissions.



Despite this, native apps are more popular than ever. App stores are exploding. Why is this, if - all other things being equal - web apps have a better user experience?

Part of it is that app stores make it easier to charge for a product, but free apps with no paid equivalent have also been exploding in popularity.

Past Reputation

Slow and clunky, with constant page refreshes

One reason is that web apps have a reputation for being good at certain things and bad at others.

Suppose someone tells you, "I just made a really great photo editor. It's completely browser-based." How does the fact that it's completely browser-based affect your expectations? In a positive, negative, or neutral way?

Reputation comes from history. Historically, web apps were crude because their interactivity was tied to re-rendering HTML on the server, one request at a time. These web apps became popular because they were still more convenient than driving to the store to purchase a disk (before the days of app stores), but their reputation for clunkiness has outlasted the reality of modern web app architecture.

Present Potential

What can we do now that we couldn't in the 1990s?

Check out http://dreamwriter.io for a real-world example of what we can do now that we couldn't do in the 1990s. We've come a long way!

Dreamwriter

No web server
100% static files on Amazon S3
Syncs with Dropbox

In contrast to 1990s-era web apps, Dreamwriter's architecture is not at all dependent on web servers. Despite this, it can still provide the rich functionality one would expect from a modern editor, including syncing with third-party services.

The App Cache

```
<html manifest="http://example.com/my-app.appcache">
```

Making a web app that works completely offline starts with the application cache. Here's how you enable one.

“Application Cache is a douchebag.”

Jake Archibald, A List Apart

The Application Cache is a minefield of gotchas. Jake Archibald wrote an excellent article on it, entitled “Application Cache is a douchebag,” on A List Apart: http://alistapart.com/article/application-cache-is...

It's an extensive resource, and goes into much more depth than this talk reasonably can. It s essentially required reading for anyone who wants to dive into the world of offline-friendly web apps.

```
CACHE MANIFEST
# v2.1.7

CACHE:
favicon.ico
/default-photo-of-the-day.png

NETWORK:
*

FALLBACK:
/ /homepage-offline.html
photo-of-the-day/ /default-photo-of-the-day.png
```

The manifest file is split into three sections. The CACHE section lists which resources will explicitly be made available offline. Rather than micromanaging this section, it's generally best to have it automatically generated during a build, for example by reading all the filenames in a directory and outputting them here. This is especially true if using hashed filenames.

The NETWORK section is the opposite: it lists which resources require a connection. This can either be a whitelist of explicit paths (like the CACHE section), or an asterisk, indicating that all paths not specified in the CACHE section are assumed to require a connection. Without the asterisk, non-cached resources simply will not load

The FALLBACK section bridges the two. It specifies resources that should be fetched over the network if possible, but which can fall back to a specified cached resource if the network is unavailable. This can be applied to individual resources or entire directories, and does not perform a redirect; rather, it simply changes what content is served via the original URL.

The browser won't try to update the locally-stored versions of any cached files unless the manifest changes. Including a version string in a comment is an easy way to (technically) change the content of the manifest, causing the browser to check for updated versions of the files listed.

Reading Files

```
var reader = new FileReader();

reader.onload = function(error) {
  var arrayBuffer = reader.result;

  // ...do something with arrayBuffer
}

reader.readAsArrayBuffer(file);
```

This can be used to read gzipped files, such as .docx!

Downloading Data

```
var blob = new Blob(
  [someHtmlContent],
  "text/html;charset=" + document.characterSet);

saveAs(blob, "document.html");
```

FileSaver.js

Thanks to Blob and FileSaver.js, it's possible to make dynamic data downloadable by the user. Blob also supports binary data, so this can work just as well for image editing apps as it does for writing apps.

Offline Persistence

Cookies (synchronous, 4096 bytes per cookie)

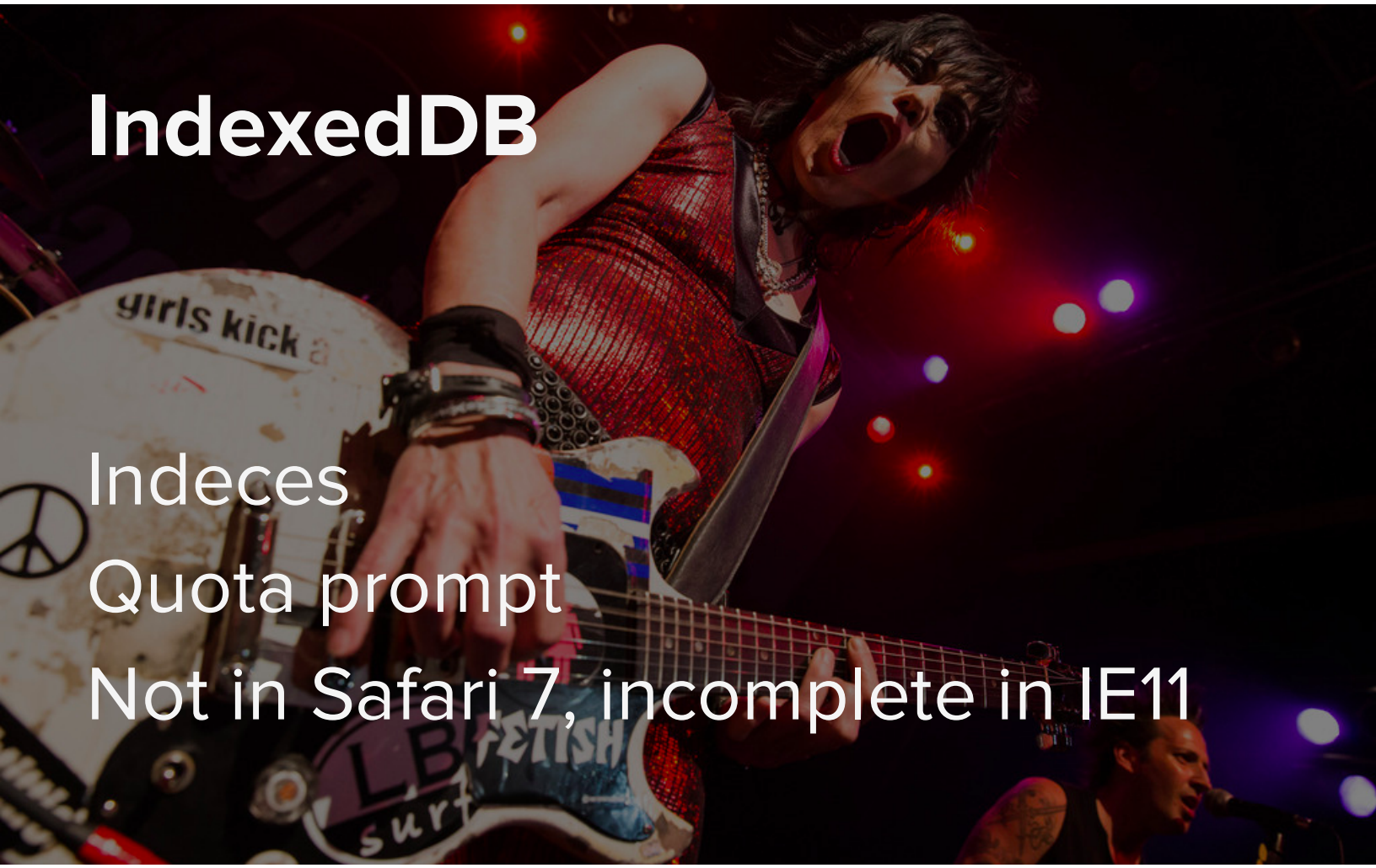
localStorage (synchronous, 5MB hard cap)

FileSystem API (RIP)

WebSQL (RIP)

IndexedDB (asynchronous, stores large data)

These are your options when it comes to offline persistence. IndexedDB is the most robust.



IndexedDB

Indeces

Quota prompt

Not in Safari 7, incomplete in IE11

Compared to other key-value stores like CouchDB and MongoDB, IndexedDB has fairly limited querying capabilities. Don't go in expecting a rich feature set; it's much better than localStorage, but very bare-bones compared to what you'll find on a server.

It's also not major available in the latest version of every browser yet. It will be in Safari 8 and iOS Safari 8, but as yet there is no version of Internet Explorer that completely implements a working version.

The quota prompt can be subtle and easily missed by users, yet it will block your IndexedDB transactions from going through until the user accepts it. A workaround is to display something on the screen to draw the user's attention to the prompt, but this requires detecting when the prompt is visible - and there's no easy, reliable way to do this cross-browser.

With Base64 encoding, you can translate binary data to and from string representations. By storing these encoded strings in IndexedDB, you can effectively (albeit with a performance penalty) cache binary content for offline use.

Third-Party Services

Having user data stuck on end user's machines is in some ways worse than the 1990s, because it can easily be wiped out in the case of hardware failure. Backing it up on a system with some redundancy, like Dropbox, solves this problem.

The Same-Origin Policy

CORS and easyXDM

Syncing with third-party services without using your own web server as an intermediary means you run afoul of the same-origin policy, which states that the browser cannot talk to other domains. For example, a page loaded from dreamwriter.io cannot send AJAX reuqests to dropbox.com - or even to api.dreamwriter.io, because that is on a different subdomain.

The preferred way to work around this is CORS, or Cross-Origin Resource Sharing. Once CORS headers are configured on the remote server (in this case Dropbox, via the Dropbox developer tools), they tell the browser to permit communication with the remote domain in question.

Older browsers do not support CORS, and not all third-party services do either. You can use easyXDM to work around this, so long as you have the ability to serve HTML files you provide from the remote server. (These files are loaded up in an iframe in the browser, since iframes can load content from other domains, and easyXDM provides a way to communicate via message-passing with the remote server through the iframe.)

Dropbox uses OAuth 2 to authenticate with its API. OAuth 2 uses HTTPS for security, meaning you don't have to do cryptography in the browser, but it also means you have to be able to serve your site over HTTPS.

This is trivial if you are running your own server, and possible if you are using S3 (as Amazon provides other services that sit in front of S3 and do things like serving custom SSL certificates), but if you are using a static hosting service like GitHub Pages, keep in mind that you may not be able to use your own domain name and also provide a custom SSL certificate.

Opening and downloading files
Local persistence via IndexedDB
Synchronization across tabs
3rd-party service integrations
...and much more from HTML5!

Writing to the clipboard still requires Flash. We cannot make arbitrary changes to the filesystem. Both of these contribute to web apps' reputation for built-in security, but limit what web apps can do.

Web apps currently cannot be first-class citizens in the operating system's task switcher, but...

The Web Manifest Specification enables one of the few things native apps have going for them: being first-class citizens in the operating system's task switching list. That's the future: being able to do what Fluid App currently works around, except trivially and with maximum power from the most up-to-date version of the browser.

This kind of thing blurs the lines even further between web apps and native apps, and I for one couldn't be more excited about what the future holds for software!

Thanks for listening.