Making Fullscreen Experiences

We have the ability to easily make immersive fullscreen websites and applications, but like anything on the web there are a couple of ways to do it. This is especially important now that more browsers are supporting an "installed web app" experience which launch fullscreen.

Getting your app or site fullscreen

There are several ways that a user or developer can get a web app fullscreen.

- Request the browser go fullscreen in response to a user gesture.
- Install the app to the home screen.
- Fake it: auto-hide the address bar.

Request the browser go fullscreen in response to a user gesture

Not all platforms are equal. iOS Safari doesn't have a fullscreen API, but we do on Chrome on Android, Firefox, and IE 11+. Most applications you build will use a combination of the JS API and the CSS selectors provided by the fullscreen specification. The main JS API's that you need to care about when building a fullscreen experience are:

- `element.requestFullscreen()` (currently prefixed in Chrome, Firefox, and IE) displays the element in fullscreen mode.
- `document.exitFullscreen()` (currently prefixed in Chrome, Firefox and IE. Firefox uses `cancelFullScreen()` instead) cancels fullscreen mode.
- `document.fullscreenElement` (currently prefixed in Chrome, Firefox, and IE) returns true if any of the elements are in fullscreen mode.

Note: You will notice that in the prefixed versions there is a lot of inconsistency between the casing of the 'S' in screen. This is awkward, but this is the problem with specs that are in flight.

When your app is fullscreen you no longer have the browser's UI controls available to you. This changes the way that users interact with your experience. They don't have the standard navigation controls such as Forwards and Backwards; they don't have their escape hatch that is the Refresh button. It's important to cater for this scenario. You can use some CSS selectors to help you change the style and presentation of your site when the browser enters fullscreen mode.

```
<button id="goFS">Go fullscreen</button>
<script>
  var goFS = document.getElementById("goFS");
  goFS.addEventListener("click", function() {
      document.body.requestFullscreen();
  }, false);
</script>
```

The above example is a little contrived; I've hidden all the complexity around the use of vendor prefixes.

Note: Damn you, vendor prefixes!

The actual code is a lot more complex. Mozilla has created a very useful script that you can use to toggle fullscreen. As you can see, the vendor prefix situation it is complex and cumbersome compared to the specified API. Even with the slightly simplified code below, it is still complex.

```
function toggleFullScreen() {
  var doc = window.document;
  var docEl = doc.documentElement;

  var requestFullScreen = docEl.requestFullscreen || docEl.mozRequestFullScreen |
  var cancelFullScreen = doc.exitFullscreen || doc.mozCancelFullScreen || doc.web

  if(!doc.fullscreenElement && !doc.mozFullScreenElement && !doc.webkitFullscreen
    requestFullScreen.call(docEl);
  }
  else {
    cancelFullScreen.call(doc);
  }
}
```

We web developers hate complexity. A nice high-level abstract API you can use is Sindre Sorhus' Screenfull.js module which unifies the two slightly different JS API's and vendor prefixes into one consistent API.

Fullscreen API Tips

Making the document fullscreen

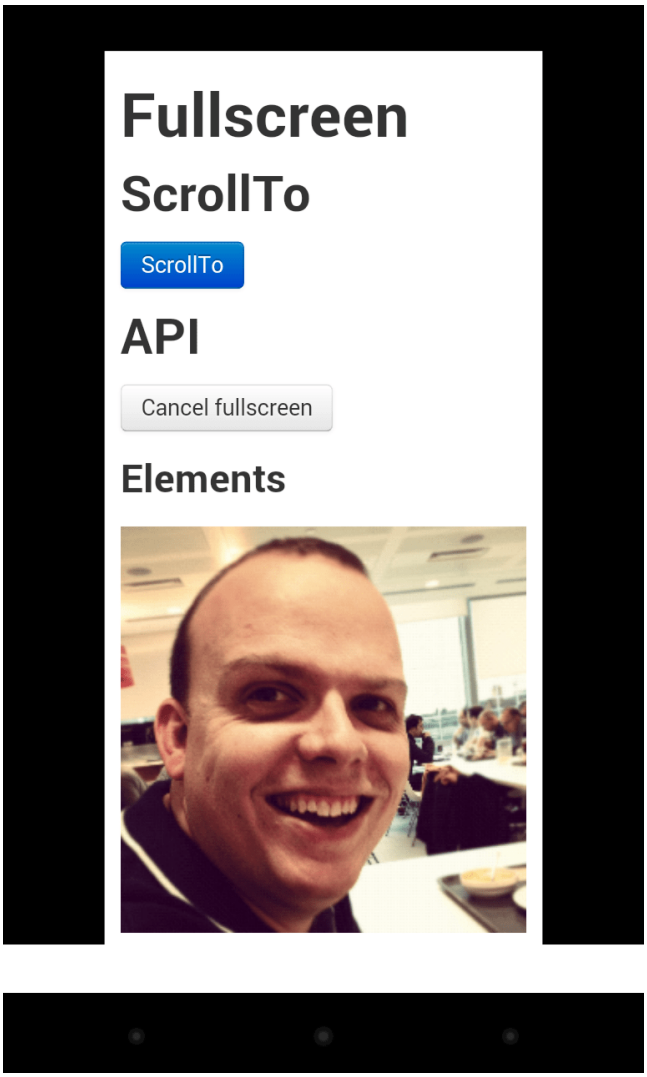

Figure 1: Fullscreen on the body element.

It is natural to think that you take the body element fullscreen, but if you are on a WebKit or Blink based rendering engine you will see it has an odd effect of shrinking the body width to the smallest possible size that will contain all the content. (Mozilla Gecko is fine.)

Fullscreen

ScrollTo

ScrollTo

API

Cancel fullscreen

Elements

Figure 2: Fullscreen on the document element.

To fix this, use the document element instead of the body element:

```
document.documentElement.requestFullscreen();
```

Making a video element fullscreen

To make a video element fullscreen is exactly the same as making any other element fullscreen. You call the **`requestFullscreen`** method on the video element.

```
<video id=videoElement></video>
<button id="goFS">Go Fullscreen</button>
<script>
  var goFS = document.getElementById("goFS");
  goFS.addEventListener("click", function() {
      var videoElement = document.getElementById("videoElement");
```

```
      videoElement.requestFullscreen();
  }, false);
</script>
```

If your **<video>** element doesn't have the controls attribute defined, there's no way for the user to control the video once they are fullscreen. The recommended way to do this is to have a basic container that wraps the video and the controls that you want the user to see.

```
<div id="container">
  <video></video>
  <div>
    <button>Play</button>
    <button>Stop</button>
    <button id="goFS">Go fullscreen</button>
  </div>
</div>
<script>
  var goFS = document.getElementById("goFS");
  goFS.addEventListener("click", function() {
      var container = document.getElementById("container");
      container.requestFullscreen();
  }, false);
</script>
```

This gives you a lot more flexibility because you can combine the container object with the CSS pseudo selector (for example to hide the "goFS" button.)

```
<style>
  #goFS:-webkit-full-screen #goFS {
    display: none;
  }
  #goFS:-moz-full-screen #goFS {
    display: none;
  }
  #goFS:-ms-fullscreen #goFS {
    display: none;
  }
  #goFS:fullscreen #goFS {
    display: none;
  }
</style>
```

Using these patterns, you can detect when fullscreen is running and adapt your user interface appropriately, for example:

- By providing a link back to the start page

- By Providing a mechanism to close dialogs or travel backwards

Launching a page fullscreen from home screen

Launching a fullscreen web page when the user navigates to it is not possible. Browser vendors are very aware that a fullscreen experience on every page load is a huge annoyance, therefore a user gesture is required to enter fullscreen. Vendors do allow users to "install" apps though, and the act of installing is a signal to the operating system that the user wants to launch as an app on the platform.

Across the major mobile platforms it is pretty easy to implement using either meta tags, or manifest files as follows.

iOS

Since the launch of the iPhone, users have been able to install Web Apps to the home screen and have them launch as full-screen web apps.

```
<meta name="apple-mobile-web-app-capable" content="yes">
```

> If content is set to yes, the web application runs in full-screen mode; otherwise, it does not. The default behavior is to use Safari to display web content. You can determine whether a webpage is displayed in full-screen mode using the window.navigator.standalone read-only Boolean JavaScript property. Apple

Chrome for Android

The Chrome team has recently implemented a feature that tells the browser to launch the page fullscreen when the user has added it to the home screen. It is similar to the iOS Safari model.

```
<meta name="mobile-web-app-capable" content="yes">
```

> You can set up your web app to have an application shortcut icon added to a device's home screen, and have the app launch in full-screen "app mode" using Chrome for Android's "Add to Home screen" menu item. Google Chrome

A better option is to use the Web App Manifest.

Web App Manifest (Chrome, Opera, Firefox, Samsung)

The Manifest for Web applications is a simple JSON file that gives you, the developer, the ability to control how your app appears to the user in the areas that they would expect to see apps (for example the mobile home screen), direct what the user can launch and, more importantly, how they can launch it. In the future the manifest will give you even more control over your app, but right now we are just focusing on how your app can be launched. Specifically:

1. Telling the browser about your manifest
2. Describing how to launch

Once you have the manifest created and it is hosted on your site, all you need to do is add a link tag from all your pages that encompass your app, as follows:

```
<link rel="manifest" href="/manifest.json">
```

Chrome has supported Manifests since version 38 for Android (October 2014) and it gives you the control over how your web app appears when it is installed to the home screen (via the **`short_name`**, **`name`** and **`icons`** properties) and how it should be launched when the user clicks on the launch icon (via **`start_url`**, **`display`** and **`orientation`**).

An example manifest is shown below. It doesn't show everything that can be in a manifest.

```
{
  "short_name": "Kinlan's Amaze App",
  "name": "Kinlan's Amazing Application ++",
  "icons": [
    {
      "src": "launcher-icon-4x.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ],
  "start_url": "/index.html",
  "display": "standalone",
  "orientation": "landscape"
}
```

This feature is entirely progressive and allows you create better, more integrated experiences for users of a browser that supports the feature.

When a user adds your site or app to the home screen, there is an intent by the user to treat it like an app. This means you should aim to direct the user to the functionality of your app

rather than a product landing page. For example, if the user is required to sign-in to your app, then that is a good page to launch.

Utility apps

The majority of utility apps will benefit from this immediately. For those apps you'll likely want them launched standalone just alike every other app on a mobile platform. To tell an app to launch standalone, add this the Web App Manifest:

```
"display": "standalone"
```

Games

The majority of games will benefit from a manifest immediately. The vast majority of games will want to launch full-screen and forced a specific orientation.

If you are developing a vertical scroller or a game like Flappy Birds then you will most likely want your game to always be in portrait mode.

```
"display": "fullscreen",
"orientation": "portrait"
```

If on the other hand you are building a puzzler or a game like X-Com, then you will probably want the game to always use the landscape orientation.

```
"display": "fullscreen",
"orientation": "landscape"
```

News sites

News sites in most cases are pure content-based experiences. Most developers naturally wouldn't think of adding a manifest to a news site. The manifest will let you define what to launch (the front page of your news site) and how to launch it (fullscreen or as a normal browser tab).

The choice is up to you and how you think your users will like to access your experience. If you want your site to have all the browser chrome that you would expect a site to have, you can set the display to `browser`.

```
"display": "browser"
```

If you want your news site to feel like the majority of news-centric apps treat their experiences as apps and remove all web-like chrome from the UI, you can do this by setting display to **`standalone`**.

```
"display": "standalone"
```

Fake it: auto-hide the address bar

You can "fake fullscreen" by auto-hiding the address bar as follows:

```
window.scrollTo(0,1);
```

Caution: I am telling you this as a friend. It exists. It is a thing, but it is a hack. Please don't use it. — Paul

This is a pretty simple method, the page loads and the browser bar is told to get out of the way. Unfortunately it is not standardized and not well supported. You also have to work around a bunch of quirks.

For example browsers often restore the position on the page when the user navigates back to it. Using **`window.scrollTo`** overrides this, which annoys the user. To work around this you have to store the last position in localStorage, and deal with the edge cases (for example, if the user has the page open in multiple windows).

UX guidelines

When you are building a site that takes advantage of full screen there are a number of potential user experience changes that you need to be aware of to be able to build a service your users will love.

Don't rely on navigation controls

iOS does not have a hardware back button or refresh gesture. Therefore you must ensure that users can navigate throughout the app without getting locked in.

You can detect if you are running in a fullscreen mode or an installed mode easily on all the major platforms.

iOS

On iOS you can use the **`navigator.standalone`** boolean to see if the user has launched from the home screen or not.

```
if(navigator.standalone == true) {
  // My app is installed and therefore fullscreen
}
```

Web App Manifest (Chrome, Opera, Samsung)

When launching as an installed app, Chrome is not running in true fullscreen experience so **`document.fullscreenElement`** returns null and the CSS selectors don't work.

When the user requests fullscreen via a gesture on your site, the standard fullscreen API's are available including the CSS pseudo selector that lets you adapt your UI to react to the fullscreen state like the following

```
selector:-webkit-full-screen {
  display: block; // displays the element only when in fullscreen
}

selector {
  display: none; // hides the element when not in fullscreen mode
}
```

If the users launches your site from the home screen the **`display-mode`** media query will be set to what was defined in the Web App Manifest. In the case of pure fullscreen it will be:

```
@media (display-mode: fullscreen) {

}
```

If the user launches the application in standalone mode, the **`display-mode`** media query will be **`standalone`**:

```
@media (display-mode: standalone) {

}
```

Firefox

When the user requests fullscreen via your site or the user launches the app in fullscreen mode all the standard fullscreen API's are available, including the CSS pseudo selector, which lets you adapt your UI to react to the fullscreen state like the following:

```
selector:-moz-full-screen {
  display: block; // hides the element when not in fullscreen mode
}

selector {
  display: none; // hides the element when not in fullscreen mode
}
```

Internet Explorer

In IE the CSS pseudo class lacks a hyphen, but otherwise works similarly to Chrome and Firefox.

```
selector:-ms-fullscreen {
  display: block;
}

selector {
  display: none; // hides the element when not in fullscreen mode
}
```

Specification

The spelling in the specification matches the syntax used by IE.

```
selector:fullscreen {
  display: block;
}

selector {
  display: none; // hides the element when not in fullscreen mode
}
```

Keep the user in the fullscreen experience

The fullscreen API can be a little finicky sometimes. Browser vendors don't want to lock users in a fullscreen page so they have developed mechanisms to break out of fullscreen as soon as they possibly can. This means you can't build a fullscreen website that spans multiple pages because:

- Changing the URL programmatically by using **`window.location = "http://example.com"`** breaks out of fullscreen.

- A user clicking on an external link inside your page will exit fullscreen.
- Changing the URL via the `navigator.pushState` API will also break out of the fullscreen experience.

You have two options if you want to keep the user in a fullscreen experience:

1. Use the installable web app mechanisms to go fullscreen.
2. Manage your UI and app state using the # fragment.

By using the #syntax to update the url (window.location = "#somestate"), and listening to the `window.onhashchange` event you can use the browser's own history stack to manage changes in the application state, allow the user to use their hardware back buttons, or offer a simple programmatic back button experience by using the history API as follows:

```
window.history.go(-1);
```

Let the user choose when to go fullscreen

There is nothing more annoying to the user than a website doing something unexpected. When a user navigates to your site don't try and trick them into fullscreen.

Don't intercept the first touch event and call `requestFullscreen()`.

1. It is annoying.
2. Browsers may decided to prompt the user at some point in the future about allowing the app to take up the fullscreen.

If you want to launch apps fullscreen think about using the install experiences for each platform.

Don't spam the user to install your app to a home screen

If you plan on offering a fullscreen experience via the installed app mechanisms be considerate to the user.

- Be discreet. Use a banner or footer to let them know they can install the app.
- If they dismiss the prompt, don't show it again.
- On a users first visit they are unlikely to want to install the app unless they are happy with your service. Consider prompting them to install after a positive interaction on your site.

- If a user visits your site regularly and they don't install the app, they are unlikely to install your app in the future. Don't keep spamming them.

Conclusion

While we don't have a fully standardized and implemented API, using some of the guidance presented in this article you can easily build experiences that take advantage of the user's entire screen, irrespective of the client.

Last updated October 9, 2017.