Geolocation in the Browser

From Google Gears to Geolocation Sensors

Thomas Steiner
Google LLC
20354 Hamburg, Germany
tomac@google.com

Anssi Kostiainen
Intel Corporation
02160 Espoo, Finland
anssi.kostiainen@intel.com

Marijn Kruisselbrink
Google LLC
San Francisco, CA 94105, USA
mek@google.com

ABSTRACT

Geolocation is arguably one of the most powerful capabilities of smartphones, and a lot of attention has been paid to native applications that make use of it. The discontinued Google Gears plugin was the first approach to access exact location data on the Web as well, apart from coarse location lookups based on Internet Protocol (IP) addresses; and the plugin led directly to the now widely implemented Geolocation API. The World Wide Web Consortium (W3C) Geolocation API specification defines a standard for accessing location services in the browser via JavaScript. Since the intent to deprecate the use of powerful features over insecure connections and a general demand for increased user privacy, the Geolocation API now requires a secure origin to work. For a long time, developers have also demanded more advanced features like background geolocation tracking and geofencing. The W3C Geolocation and the Devices and Sensors Working Groups, as well as the Web Incubator Community Group (WICG), have addressed these demands with the no longer maintained Geofencing API specification for the former, and, with now (early 2019) resumed efforts, the in-flight Geolocation Sensors specification for the latter two groups. This paper first provides a quick overview of the historical development of geolocation in the browser, and then gives an outlook on current and future efforts, challenges, and use cases.

1 HISTORY OF BROWSER GEOLOCATION

Geolocation has been available to developers implicitly through the mapping of Internet Protocol (IP) addresses or address blocks to known locations. There are numerous paid subscription and free geolocation databases with varying claims of accuracy that range from country level to state or city level, sometimes including ZIP/post code level. A popular free way to obtain IP-based location data was through Google's AJAX API Loader library [2], which provided an approximate, region-level estimate of a user's location through its `google.loader.ClientLocation` property. This API did not require users to install any client-side software.

1.1 Google Gears Geolocation API

More accurate location data is available through Wi-Fi access point or cell tower data, as well as through Global Positioning Service (GPS) data. This data is accessible to native applications, however, initially, not to the Web. The Google Gears browser plugin [6] for the first time exposed exact device-based geolocation data to the browser. Its Geolocation API had two JavaScript methods: `getCurrentPosition()` made a single, one-off attempt to get a position fix, while `watchPosition()` watched the user's position over time, and provided an update whenever the position changed. Both methods allowed the developer to configure which sources of location information would be used. As a simple way to get an approximate position fix with low cost in terms of both network and battery resources, Gears also kept track of the best position fix obtained from these calls and made it available as the `lastPosition` property. To a JavaScript developer of today that may have never used the plugin, the code in Listing 1 still looks very familiar, even more than ten years later.

```
var geo = google.gears.factory
    .create('beta.geolocation');

function updatePosition(position) {
  alert('Current lat/lon is: ' + position.latitude + ','
      + position.longitude);
}

function handleError(positionError) {
  alert('Attempt to get location failed: ' +
      positionError.message);
}

geo.getCurrentPosition(updatePosition, handleError);
```

Listing 1: Google Gears API (2008)

From the start, user privacy was a major concern, and the plugin authors wrote in the Wiki [7]:

> "It must be clear to users when an application is using the Geolocation API. We could implement one or both of the following UI elements:
> (1) A separate dialog from the Gears security dialog to enable the Geolocation API. If the general-purpose dialog gave access to position data, it would be easy for users to forget they allowed access to Gears, or to fail to realize enabling Gears also exposes their position data.
> (2) Some persistent UI that indicates the Geolocation API is being used. For example, there could be a bar across the bottom of the browser with an icon of a globe or map. Perhaps this UI should be 'active' somehow, indicating that something is happening, so that the user cannot forget it is being used."

In the end, they decided on approach (1), which is still how location access is granted today, despite different underlying mechanics.

WWW '19 Companion, May 13–17, 2019, San Francisco, United States of America

The similarity of the code in Listing 1 with current code is no surprise. Gears implemented the at the time current editor's draft of the W3C Geolocation specification, which some of the authors have helped to define in collaboration with Microsoft, Mozilla, and others. The goal for Gears was to advance browser capabilities and eventually make itself a relict of time, which is what has happened.

1.2 Mozilla Geode

Mozilla Geode [1] was an experimental add-on to explore geolocation in Firefox 3 ahead of the implementation of geolocation in a future product release. Geode provided an early implementation of the W3C Geolocation specification [11] so that developers could begin experimenting with enabling location-aware experiences. It included a single experimental geolocation service provider called Skyhook to map WiFi signals to locations, so that any computer with WiFi could get accurate positioning data. An interesting historical fact is that the ultimate plan for Firefox was that service providers and geolocation methods would be pluggable and user-selectable to provide users with as many choices and privacy options as possible.

1.3 W3C Geolocation API

The W3C Geolocation API specification [11] defines a high-level interface to location information associated only with the device hosting the implementation, such as latitude and longitude. The API itself is agnostic of the underlying location information sources, and no guarantee is given that it returns the device's actual location. The design enables both "one-shot" position requests as well as repeated position updates, and also includes the ability to explicitly query the cached positions. As outlined above, the work is based on prior art in the industry in the form of Google Gears, as well as the work of Turner, Sarver, and Raskin in the context of locationaware.org and geolocation in the Firefox browser [12].

Similar to its predecessors Google Gears and Mozilla Geode, the specification defines three methods: The `getCurrentPosition()` method asynchronously attempts to obtain the current location of the device. If the attempt is successful, the `successCallback` must be invoked with a new `Position` object, reflecting the current location of the device. If the attempt fails, the `errorCallback` must be invoked with a new `PositionError` object, reflecting the reason for the failure. The `watchPosition()` method returns an identifier that uniquely identifies a watch operation and then asynchronously starts that watch operation. This operation must first attempt to obtain the current location of the device. If the attempt is successful, the `successCallback` must be invoked with a new `Position` object, reflecting the current location of the device. The watch operation then must continue to monitor the position of the device and invoke the appropriate callback every time this position changes. Error handling is the same as before. The watch operation must continue until the `clearWatch` method is called with the corresponding identifier, which immediately stops the watch operation identified by the identifier. Listing 2 shows all methods in operation (see Listing 1 for comparison). This API is almost universally supported in browsers, Figure 1 shows the support situation at the time of writing.

```
function showMap(pos) {
  // Show a map centered at
  // (pos.coords.latitude, pos.coords.longitude)
}

function scrollMap(pos) {
  // Scrolls the map so that it is centered at
  // (pos.coords.latitude, pos.coords.longitude)
}

function handleError(error) {
  // Handles the error gracefully.
}

// One-shot position request
navigator.geolocation.getCurrentPosition(showMap,
     handleError);

// Request repeated updates
var watchId = navigator.geolocation.watchPosition(
    scrollMap, handleError);

function clearPositionWatch() {
  // Cancel the updates when the user clicks a button
  navigator.geolocation.clearWatch(watchId);
}
```

Listing 2: Geolocation API

1.4 W3C Geofencing API

The W3C Geofencing API [9] specification defined an API that should let Web applications set up geographic boundaries around specific locations and then receive notifications when the hosting device entered or left those areas, also referred to as geofences. The Geolocation Working Group was chartered to define a secure and privacy-sensitive interface for using client-side location information in location-aware Web applications and published an update to the Geolocation API [11] specification as a Recommendation without the geofencing feature that was worked on as a separate Geofencing API specification, but did not complete. While it would be possible to implement something similar using the Geolocation API (as long as the application is in the foreground), there were a few differences that made the proposed API look like a better choice:

> "(1) Because of the limited API surface of the Geofencing API, user agents can implement the API in a more (power) efficient way than could be achieved by regularly checking the current geographic position with the Geolocation API.
> (2) The Geofencing API is built around Service Workers. This makes it possible for a webapp to receive notifications from the Geofencing API even after the user has closed the webapp."

It was not generally agreed on[1] that Service Workers were the right technology choice, as they may be killed by the user agent at nearly any time, which means one cannot expect `watchPosition()` to keep watching for extended periods, and likewise the method `getCurrentPosition()` could only work if the position was already available, but if it has to wait for a GPS update, the Service Worker is likely to be gone by the time an answer comes back.

[1] https://www.w3.org/2014/10/27-28-geolocation-minutes.html#day21

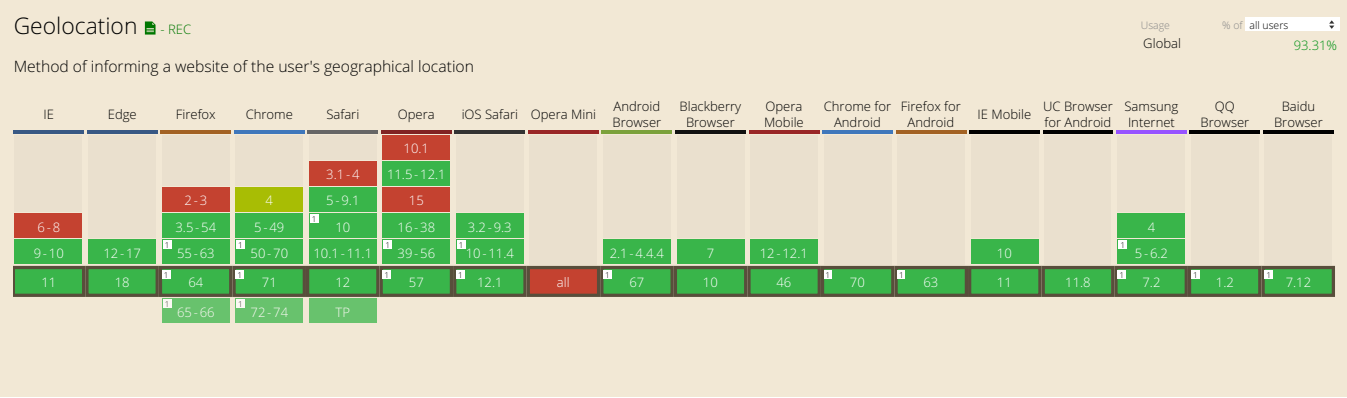

Figure 1: Browser support data for the Geolocation API, data via https://caniuse.com/#feat=geolocation

```
// On main page
navigator.serviceWorker
  .register('serviceworker.js')
  .then((swRegistration) => {
    let region = new CircularGeofenceRegion({
      name: 'myfence',
      latitude: 37.421999,
      longitude: -122.084015,
      radius: 1000
    });
    let options = {
      includePosition: true
    };
    swRegistration.geofencing.add(region, options)
      .then(
        // If more than just a name needs to be stored
        // with a geofence, now would be the time to
        // store this in some storage
        (geofence) => console.log(geofence.id),
        (error) => console.log(error)
      );
});

// In serviceworker.js
self.ongeofenceenter = (event) => {
  console.log(event.geofence.id);
  console.log(event.geofence.region.name);

  // If this is not a geofence of interest anymore,
  // remove it now
  if (event.geofence.region.name !== "myfence") {
    event.waitUntil(event.geofence.remove());
  }
};
```

Listing 3: Geofencing API (conceived example)

2 CURRENT GEOLOCATION EFFORTS

After having covered the significant milestones in the history of geolocation in the browser, the second part of the paper introduces some of the current efforts, challenges, and use cases in the field.

2.1 Background Geolocation with Wake Locks

When a device that has either of the two Geolocation API methods `watchPosition()` or `getCurrentPosition()` running goes into stand-by mode (colloquially "goes to sleep"), or when the browser window is backgrounded, location reporting stops soon thereafter. This renders certain use cases like fitness run trackers or maps navigation that all require background geolocation entirely impossible. Wake Locks, initially introduced with the since discontinued Firefox OS, provided a way[2] to prevent the system from sleeping and to keep certain services running when the developer wanted to keep an invisible application continuing to use GPS. This could be done by requesting a `MozWakeLock` of the type `"gps"` and using it together with `watchPosition()`. App developers needed to be responsible and think carefully about whether they needed to keep the geolocation service on or not. The risk of claiming the lock was that users may forget to close the app when they were done using it, which could result in significantly increased battery use, apart from the privacy implications of potentially inadvertently continuing to share one's location.

Work on the concept of Wake Locks is resumed in the W3C Wake Lock specification [3]. With the at the time of writing implemented version, Wake Locks of the type `"system"` (there is no type `"gps"`) can be used with `watchPosition()`. Listing 4 shows the relevant code excerpts of a Web application called "Where Am I", depicted in Figure 2, that implements this.

2.2 Geolocation Sensor

The W3C Geolocation Sensor specification [8] extends the `Sensor` interface defined in the W3C Generic Sensor API [15] to in turn define the new `GeolocationSensor` interface for obtaining the geolocation of the hosting device.

The Generic Sensor API is a set of interfaces that expose sensor devices to the Web platform. It consists of the base `Sensor` interface and a set of concrete sensor classes (like `GeolocationSensor`, `AmbientLightSensor`, `Accelerometer`,...) built on top. Having a base interface simplifies the implementation and specification process for the concrete sensor classes. The core functionality is specified by the base interface, and `GeolocationSensor` merely extends it with a tiny set of additional attributes and methods.

[2] https://developer.mozilla.org/en-US/docs/Archive/B2G_OS/API/Wake_Lock_API/Keeping_the_geolocation_on_when_the_application_is_invisible

```
try {
  wakeLockObj = await navigator.getWakeLock('system');
  wakeLockObj.addEventListener('activechange', () => {
    wakelock.textContent =
        `The ${wakeLockObj.type} wake lock is ${
        wakeLockObj.active ? 'active' : 'not active'}.`;
  });
} catch (err) {
  console.error('Could not obtain wake lock', err);
}

const toggleWakeLock = () => {
  if (wakeLockRequest) {
    wakeLockRequest.cancel();
    wakeLockRequest = null;
    return;
  }
  wakeLockRequest = wakeLockObj.createRequest();
};

const startTracking = () => {
  id = navigator.geolocation.watchPosition(success,
      error, opt);
};

trackButton.addEventListener('click', () => {
  toggleWakeLock();
  if (id) {
    stopTracking();
  } else {
    startTracking();
  }
});
```

Listing 4: Using the Wake Lock API for background geolocation tracking

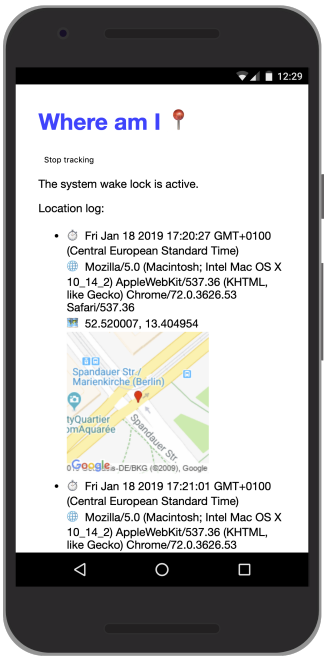

Figure 2: Web app "Where Am I" with Wake Lock for background location tracking (https://whereami.glitch.me/)

The feature set of `GeolocationSensor` is similar to that of the Geolocation API [12], but it is surfaced through a modern API that

```
// Get a new geolocation reading every second
const geo = new GeolocationSensor({frequency: 1});
geo.start();

geo.onreading = () => console.log(
    `lat: ${geo.latitude}, long: ${geo.longitude}`);

geo.onerror = event => console.error(
    event.error.name, event.error.message);

// Get a one-shot geolocation reading
GeolocationSensor.read()
  .then(geo => console.log(
      `lat: ${geo.latitude}, long: ${geo.longitude}`))
  .catch(error => console.error(error.name));
```

Listing 5: Geolocation Sensor API

is consistent across contemporary Sensor APIs, improves security and privacy, and is extensible. The API aims to be polyfillable[3] on top of the existing Geolocation API. As all recent APIs, instead of callback functions, Geolocation Sensor uses promises for one-shot or one-and-done operations.[4]

Unlike with the previous Geolocation API, the Geolocation Sensor API does not have dedicated callbacks for one-shot versus continuous location requests,[5] but instead fires an event according to a frequency (specified in Hertz) that is used to calculate the requested sampling frequency for the associated platform sensor and to define the upper bound of the reporting frequency for the `GeolocationSensor` object. Listing 5 shows the API in practice.

3 GEOLOCATION PRIVACY

3.1 Insecure Origin Usage of Geolocation

As the Web platform is extended to enable more useful and powerful applications, it becomes increasingly important to ensure that the features which enable those applications are enabled only in contexts which meet a minimum security level. The Secure Contexts specification [16] describes threat models for feature abuse on the Web as follows:

> "Granting permissions to unauthenticated origins is, in the presence of a network attacker, equivalent to granting the permissions to any origin. The state of the Internet is such that we must indeed assume that a network attacker is present. Generally, network attackers fall into two classes: passive and active.
> **1. Passive Network Attacker:** A 'Passive Network Attacker' is a party who is able to observe traffic flows but who lacks the ability or chooses not to modify traffic at the layers which this specification is concerned with. [...]
> **2. Active Network Attacker:** An 'Active Network Attacker' has all the capabilities of a 'Passive Network Attacker' and is additionally able to modify, block or replay any data transiting the network. These capabilities are available to potential adversaries at many levels of capability, from compromised devices offering

[3] https://github.com/w3c/sensors/blob/master/polyfills/geolocation.js
[4] https://www.w3.org/2001/tag/doc/promises-guide#one-and-done
[5] `GeolocationSensor` does have a dedicate static operation `read()` for one-shots.

> or simply participating in public wireless networks, to Internet Service Providers indirectly introducing security and privacy vulnerabilities [...], to parties with direct intent to compromise security or privacy who are able to target individual users, organizations or even entire populations."

In a rare case of breaking backwards compatibility and after a thorough risk analysis, the concepts in [16] were applied to features that had already shipped in browsers and which did not meet the—new, not present at the time—requirements. Specifically, geolocation support was removed on insecure origins, motivated by the large privacy risk for end users of even passive attackers sniffing geolocation obtained from this API.

3.2 Feature Policy Integration

Feature Policy [4] defines a mechanism that allows developers to selectively enable and disable use of various browser features and APIs. The Feature Policy integration in Generic Sensor API is used to control access to sensors data for a frame. By default the `Sensor` objects (and therefore the `GeolocationSensor`) can be created only within a main frame or same-origin subframes, thus preventing cross-origin iframes from unsanctioned reading of sensor data. This default behavior can be modified by explicitly enabling or disabling of the corresponding policy-controlled features. The snippet below illustrates granting geolocation data access to a cross-origin iframe, meaning that now `GeolocationSensor` objects can be created there. Listing 6 shows how this looks like in practice.

```
<iframe src="https://3rd-party.com" allow="geolocation">
```

Listing 6: Allowing an iframe to use `GeolocationSensor`

3.3 Focused Area and Visibility State

Sensor readings are only available for active documents whose origin is same origin-domain with the currently focused area document and whose visibility state is `"visible"`. This is done in order to mitigate the risk of a skimming attack against the browsing context containing an element which has gained focus, for example when the user carries out an in-game purchase using a third party payment service from within an iframe.

Similar to Subsection 2.1, these constraints limit use cases like fitness run trackers or maps navigation that require sensor readings in potentially non-active, non-visible documents, for example while the user chooses a music track to accompany their run or drive.

3.4 Permissions

Access to sensor readings are controlled by the W3C Permissions API [10]. User agents use a number of criteria to grant access to the readings. Note that while access to some sensors can be granted without prompting the user, `GeolocationSensor` always requires a prompt. Listing 7 shows the permissions flow. In contrast, with the legacy Geolocation API, the user agent prompts the user automatically upon the first time they try to use either of `watchPosition` or `getCurrentPosition()`.

```
navigator.permissions.query({name: "geolocation"})
  .then(({state}) => {
    switch (state) {
      case "granted":
        showLocalNewsWithGeolocation();
        break;
      case "prompt":
        showButtonToEnableLocalNews();
        break;
      default:
        // Don't do anything if the permission was denied
        break;
    }
  });
```

Listing 7: Asking for permission to use `GeolocationSensor`

The current permission prompt has the options to allow or to block the location access request (or for what it is worth, to ignore by closing the dialog). Interestingly, Raskin envisioned [12] a far more detailed permissions dialog, depicted in Figure 3, which had granular levels of permitted access, for example, to limit the granularity to city or state level, however, this was never implemented.

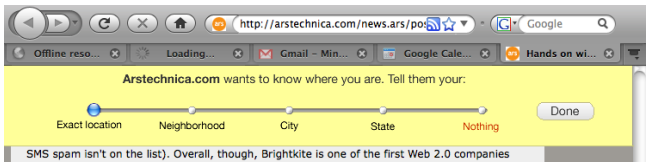

Figure 3: Aza Raskin's security prompt mock-up [12]

4 FUTURE WORK AND CONCLUSION

Future work will happen in several areas and touch upon both technological as well as user privacy related aspects. In the following we present some of them, but note that especially the permission aspects reach further than geolocation.

We will continue our experiments[6] with Wake Locks and how they can interact with Geolocation Sensor, with a special focus on sensor readings in non-focused and non-visible states (Subsection 3.3). Further, after having explored use cases around background geolocation and geofencing [8] in more detail, we will resume our work on Geofencing [9] and investigate integration of this feature with Geolocation Sensor [8], possibly via the extensibility mechanism defined in Service Workers [14] that makes them extensible from other specifications through a functional event by extending the `ExtendableEvent` interface.

User privacy is central to all sensor readings, and particularly to geolocation. As outlined in Subsection 1.1, from the start it was a major concern and especially with new background geolocation and geofencing use cases it remains an issue of high importance. In [13], Russell and Nattestad have explored permission dialog abuse, annoyance, and fatigue of users; and according to their opinion (which we support), "today's permissions model and API surface area should be heavily revised [...] to enable more flexible permission policies" in a more uniform Permissions API [10]. Some ideas

[6] https://medium.com/dev-channel/experimenting-with-the-wake-lock-api-b6f42e0a089f

include time-limiting and one-off requests and permission grants, but also extending Web Application Manifest [5] to allow sites to identify to the runtime a maximum set of permissions, and turn down permission requests not explicitly included in the Web Application Manifest.

Concluding, in this paper we have first provided an overview of the historical development of geolocation in the browser, and given an outlook on current and future efforts, challenges, and use cases in the field. The importance of user privacy is ever-increasing, and in our work on geolocation, background geolocation, and geofencing, we keep privacy as a top priority. While many privacy concerns such as unsanctioned third-party access and fingerprinting can now be mitigated by technical measures, we continue to work with the privacy research community to ensure the Web remains a both capable and safe platform today and in the future by carefully assessing every new proposed feature for possible privacy threats.

REFERENCES

[1] Chris Beard. *Introducing Geode.* Blog. Mozilla Labs, https://blog.mozilla.org/labs/2008/10/introducing-geode/.

[2] Steve Block. 2008. *Two new ways to location-enable your web apps.* Blog. Google Developers, https://developers.googleblog.com/2008/08/two-new-ways-to-location-enable-your.html.

[3] Ilya Bogdanovich, Andrey Logvinov, and Marcos Cáceres. 2017. *Wake Lock API.* Candidate Recommendation. w3c, https://www.w3.org/TR/wake-lock/.

[4] Ian Clelland. *Feature Policy.* Editor's Draft. w3c, https://w3c.github.io/webappsec-feature-policy/.

[5] Marcos Cáceres, Kenneth Rohde Christiansen, Mounir Lamouri, Anssi Kostiainen, Rob Dolin, and Matt Giuca. *Web App Manifest, Living Document.* Working Draft. w3c, https://www.w3.org/TR/appmanifest/.

[6] Google Gears. 2008. *Geolocation API.* Documentation. Google, https://web.archive.org/web/20080902073745/http://code.google.com:80/apis/gears/api_geolocation.html.

[7] Google Gears. 2008. *GeolocationAPI.* Wiki. Google, https://web.archive.org/web/20080901235406/http://code.google.com:80/p/gears/wiki/GeolocationAPI.

[8] Anssi Kostiainen, Thomas Steiner, and Marijn Kruisselbrink. 2018. *Geolocation Sensor.* Working Draft. w3c, https://www.w3.org/TR/geolocation-sensor/.

[9] Marijn Kruisselbrink. 2017. *Geofencing API.* Working Group Note. w3c, https://www.w3.org/TR/geofencing/.

[10] Mounir Lamouri, Marcos Cáceres, and Jeffrey Yasskin. *Permissions.* Working Draft. w3c, https://www.w3.org/TR/permissions/.

[11] Andrei Popescu. 2016. *Geolocation API Specification 2nd Edition.* Recommendation. w3c, https://www.w3.org/TR/geolocation-API/.

[12] Aza Raskin. 2010. *Geolocation in Firefox and Beyond.* Blog. Aza on Design, http://www.azarask.in/blog/post/geolocation-in-firefox-and-beyond/.

[13] Alex Russell and Thomas Nattestad. *Permissions Workshop.* Position Paper. w3c, https://www.w3.org/Privacy/permissions-ws-2018/papers/thomas-nattestad.pdf.

[14] Alex Russell, Jungkee Song, Jake Archibald, and Marijn Kruisselbrink. 2017. *Service Workers 1.* Working Draft. w3c, https://www.w3.org/TR/service-workers/.

[15] Rick Waldron, Mikhail Pozdnyakov, and Alexander Shalamov. 2018. *Generic Sensor API.* Candidate Recommendation. w3c, https://www.w3.org/TR/generic-sensor/.

[16] Mike West. 2016. *Secure Contexts.* Candidate Recommendation. w3c, https://www.w3.org/TR/secure-contexts/.