Geolocation API Specification

W3C Editor's Draft 20 November 2017

This version:
https://w3c.github.io/geolocation-api/

Latest published version:
https://www.w3.org/TR/geolocation-API/

Latest editor's draft:
https://w3c.github.io/geolocation-api/

Test suite:
https://wpt.fyi/geolocation-API/

Editor:
Andrei Popescu (Google Inc.)

Repository:
We are on Github.
File a bug.
Commit history.

Implementation:
Test Suite Results
Test Suite repository

Abstract

This specification defines an API that provides scripted access to geographical location information associated with the hosting device.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

WARNING

Implementors need to be aware that the future work is now happening on the Geolocation Sensor API.

This document was published by the Geolocation Working Group as an Editor's Draft. Comments regarding this document are welcome. Please send them to public-geolocation@w3.org (subscribe, archives).

Publication as an Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 February 2018 W3C Process Document.

Table of Contents

1. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *OPTIONAL*, *SHOULD*, and *SHOULD NOT* are to be interpreted as described in [RFC2119].

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("*MUST*", "*SHOULD*", "*MAY*", etc) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps *MAY* be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

User agents *MAY* impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

2. Introduction

This section is non-normative.

The Geolocation API 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. Common sources of location information include Global Positioning System (GPS) and location inferred from network signals such as IP address, RFID, WiFi and Bluetooth MAC addresses, and GSM/CDMA cell IDs, as well as user input. No guarantee is given that the API returns the device's actual location.

The API is designed to enable both "one-shot" position requests and repeated position updates, as well as the ability to explicitly query the cached positions. Location information is represented by latitude and longitude coordinates. The Geolocation API in this specification builds upon earlier work in the industry, including [AZALOC], the Gears Geolocation API, and LocationAware.org .

The following code extracts illustrate how to obtain basic location information:

EXAMPLE 1: Example of a "one-shot" position request

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

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

EXAMPLE 2: Example of requesting repeated position updates

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

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

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

EXAMPLE 3: Example of requesting repeated position updates and handling errors

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

function handleError(error) {
  // Update a div element with error.message.
}

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

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

EXAMPLE 4: Example of requesting a potentially cached position

```
// Request a position. We accept positions whose age is not
// greater than 10 minutes. If the user agent does not have a
// fresh enough cached position object, it will automatically
// acquire a new one.
navigator.geolocation.getCurrentPosition(successCallback,
                                         errorCallback,
                                         {maximumAge:600000});

function successCallback(position) {
  // By using the 'maximumAge' option above, the position
  // object is guaranteed to be at most 10 minutes old.
}

function errorCallback(error) {
  // Update a div element with error.message.
}
```

EXAMPLE 5: Forcing the user agent to return a fresh cached position

```
// Request a position. We only accept cached positions whose age is not
// greater than 10 minutes. If the user agent does not have a fresh
// enough cached position object, it will immediately invoke the error
// callback.
navigator.geolocation.getCurrentPosition(successCallback,
                                         errorCallback,
                                         {maximumAge:600000, timeout:0});

function successCallback(position) {
  // By using the 'maximumAge' option above, the position
  // object is guaranteed to be at most 10 minutes old.
  // By using a 'timeout' of 0 milliseconds, if there is
  // no suitable cached position available, the user agent
  // will asynchronously invoke the error callback with code
  // TIMEOUT and will not initiate a new position
  // acquisition process.
}

function errorCallback(error) {
  switch(error.code) {
    case error.TIMEOUT:
      // Quick fallback when no suitable cached position exists.
      doFallback();
      // Acquire a new position object.
      navigator.geolocation.getCurrentPosition(successCallback, errorCallback);
      break;
    case ... // treat the other error cases.
  };
}

function doFallback() {
  // No fresh enough cached position available.
  // Fallback to a default position.
}
```

EXAMPLE 6: Forcing the user agent to return any available cached position

```
// Request a position. We only accept cached positions, no matter what
// their age is. If the user agent does not have a cached position at
// all, it will immediately invoke the error callback.
navigator.geolocation.getCurrentPosition(successCallback,
                                         errorCallback,
                                         {maximumAge:Infinity, timeout:0});

function successCallback(position) {
  // By setting the 'maximumAge' to Infinity, the position
  // object is guaranteed to be a cached one.
  // By using a 'timeout' of 0 milliseconds, if there is
  // no cached position available at all, the user agent
  // will immediately invoke the error callback with code
  // TIMEOUT and will not initiate a new position
  // acquisition process.
  if (position.timestamp < freshness_threshold &&
      position.coords.accuracy < accuracy_threshold) {
    // The position is relatively fresh and accurate.
  } else {
    // The position is quite old and/or inaccurate.
  }
}

function errorCallback(error) {
  switch(error.code) {
    case error.TIMEOUT:
      // Quick fallback when no cached position exists at all.
      doFallback();
      // Acquire a new position object.
      navigator.geolocation.getCurrentPosition(successCallback, errorCallback);
      break;
    case ... // treat the other error cases.
  };
}

function doFallback() {
  // No cached position available at all.
  // Fallback to a default position.
}
```

3. Scope

This section is non-normative.

This specification is limited to providing a scripting API for retrieving geographic position information associated with a hosting device. The geographic position information is provided in terms of World Geodetic System coordinates [WGS84].

The scope of this specification does not include providing a markup language of any kind.

The scope of this specification does not include defining new URL schemes for building URLs that identify geographic locations.

4. Security and privacy considerations

The API defined in this specification is used to retrieve the geographic location of a hosting device. In almost all cases, this information also discloses the location of the user of the device, thereby potentially compromising the user's privacy. A conforming implementation of this specification *MUST* provide a mechanism that protects the user's privacy and this mechanism *SHOULD* ensure that no location information is made available through this API without the user's express permission.

4.1 Privacy considerations for implementers of the Geolocation API

User agents *MUST NOT* send location information to Web sites without the express permission of the user. User agents *MUST* acquire permission through a user interface, unless they have prearranged trust relationships with users, as described below. The user interface *MUST* include the host component of the document's URI. Those permissions that are acquired through the user interface and that are preserved beyond the current browsing session (i.e. beyond the time when the browsing context is navigated to another URL) *MUST* be revocable and user agents *MUST* respect revoked permissions.

Some user agents will have prearranged trust relationships that do not require such user interfaces. For example, while a Web browser will present a user interface when a Web site performs a geolocation request, a VOIP telephone *MAY* NOT present any user interface when using location information to perform an E911 function.

4.2 Privacy considerations for recipients of location information

Recipients *MUST* only request location information when necessary. Recipients *MUST* only use the location information for the task for which it was provided to them. Recipients *MUST* dispose of location information once that task is completed, unless expressly permitted to retain it by the user. Recipients *MUST* also take measures to protect this information against unauthorized access. If location information is stored, users *SHOULD* be allowed to update and delete this information.

The recipient of location information *MUST NOT* retransmit the location information without the user's express permission. Care *SHOULD* be taken when retransmitting and use of encryption is encouraged.

Recipients *MUST* clearly and conspicuously disclose the fact that they are collecting location data, the purpose for the collection, how long the data is retained, how the data is secured, how the data is shared if it is shared, how users *MAY* access, update and delete the data, and any other choices that users have with respect to the data. This disclosure *MUST* include an explanation of any exceptions to the guidelines listed above.

4.3 Additional implementation considerations

This section is non-normative.

Further to the requirements listed in the previous section, implementers of the Geolocation API are also advised to consider the following aspects that *MAY* negatively affect the privacy of their users: in certain cases, users *MAY* inadvertently grant permission to the user agent to disclose their location to Web sites. In other cases, the content hosted at a certain URL changes in such a way that the previously granted location permissions no longer apply as far as the user is concerned. Or the users might simply change their minds.

Predicting or preventing these situations is inherently difficult. Mitigation and in-depth defensive measures are an implementation responsibility and not prescribed by this specification. However, in designing these measures, implementers are advised to enable user awareness of location sharing, and to provide easy access to interfaces that enable revocation of permissions.

5. API Description

5.1 `Navigator` interface extensions

WebIDL

```
partial interface Navigator {
    readonly attribute Geolocation geolocation;
};
```

The ***geolocation*** attribute gives access to location information associated with the hosting device.

5.2 *Geolocation* interface

The Geolocation object is used by scripts to programmatically determine the location information associated with the hosting device. The location information is acquired by applying a user-agent specific algorithm, creating a `Position` object, and populating that object with appropriate data accordingly.

WebIDL

```
[NoInterfaceObject]
interface Geolocation {
    void getCurrentPosition(PositionCallback successCallback,
                            optional PositionErrorCallback errorCallback,
                            optional PositionOptions options);

    long watchPosition(PositionCallback successCallback,
                       optional PositionErrorCallback errorCallback,
                       optional PositionOptions options);

    void clearWatch(long watchId);
};

callback PositionCallback = void (Position position);

callback PositionErrorCallback = void (PositionError positionError);
```

The ***getCurrentPosition()*** method takes one, two or three arguments. When called, it *MUST* immediately return and then asynchronously attempt to obtain the current location of the device. If the attempt is successful, the *successCallback MUST* be invoked (i.e. the `handleEvent` operation *MUST* be called on the callback object) 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. ▶ **tests: 2**

The implementation of the `getCurrentPosition` method *MUST* execute the following set of steps:

1. If a cached `Position` object, whose age is no greater than the value of the `maximumAge` variable, is available, invoke the *successCallback* with the cached `Position` object as a parameter and exit this set of steps.
2. If the value of the timeout variable is 0, invoke the *errorCallback* (if present) with a new `PositionError` object whose `code` attribute is set to `TIMEOUT` and exit this set of steps.
3. Start a location acquisition operation (e.g. by invoking a platform-specific API), possibly taking into account the value of the `enableHighAccuracy` variable.
4. Start a timer that will fire after the number of milliseconds denoted by the value of the timeout variable. When the timer fires, cancel any ongoing location acquisition operations associated with this instance of the steps, invoke the *errorCallback* (if present) with a new `PositionError` object whose `code` attribute is set to `TIMEOUT`, and exit this set of steps.
5. If the operation completes successfully before the timeout expires, cancel the pending timer, invoke the *successCallback* with a new `Position` object that reflects the result of the acquisition operation and exit this set of steps.
6. If the operation fails before the timeout expires, cancel the pending timer and invoke the *errorCallback* (if present) with a new `PositionError` object whose `code` is set to `POSITION_UNAVAILABLE`.

The ***`watchPosition()`*** method takes one, two or three arguments. When called, it *MUST* immediately return a long value that uniquely identifies a watch process and then asynchronously start the 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 (i.e. the `handleEvent` operation *MUST* be called on the callback object) 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 watch process then *MUST* continue to monitor the position of the device and invoke the appropriate callback every time this position changes. The watch process *MUST* continue until the `clearWatch()` method is called with the corresponding identifier.

The ***`PositionCallBack`*** callback is invoked when a `Position` object is available, resulting from a cached object or the acquisition operation. The `PositionCallBack` callback gets set using the ***successCallback*** parameter.

The ***`PositionErrorCallBack`*** callback is invoked when a `Position` object is not available, resulting from a timeout, a permission denied, or an unability to determine the position of the device. The

`PositionErrorCallBack` callback gets set using the ***errorCallback*** parameter.

The implementation of the ***watch process*** *MUST* execute the following set of steps:

1. If a cached `Position` object, whose age is no greater than the value of the maximumAge variable, is available, invoke the *successCallback* with the cached `Position` object as a parameter.
2. Register to receive system events that indicate that the position of the device *MAY* have changed (e.g. by listening or polling for changes in WiFi or cellular signals).
3. Start a location acquisition operation (e.g. by invoking a platform-specific API), possibly taking into account the value of the *enableHighAccuracy* variable (see the definition of `enableHighAccuracy` for details).
4. Run the following ***acquisition steps***:
    1. If the timer is not already running, start a timer that will fire after the number of milliseconds denoted by the value of the timeout variable. When the timer fires, invoke the *errorCallback* (if present) with a new `PositionError` object whose `code` attribute is set to `TIMEOUT` and jump to step 6.
    2. If the location acquisition operation successfully yields a new position before the timeout expires, perform the following two steps:
        1. Cancel the pending timer. Note that the timer *MUST* be restarted once this algorithm jumps back to the beginning of the acquisition steps.
        2. If the ***new position differs significantly from the previous position***, invoke the *successCallback* with a new `Position` object that reflects the result of the acquisition operation. This step *MAY* be subject to callback rate limitation (see below).
    3. Else, if the location acquisition operation reports an error before the `timeout` expires, invoke the *errorCallback* (if present) with a new `PositionError` object whose `code` is set to `POSITION_UNAVAILABLE`. This step *MAY* be subject to callback rate limitation (see below).
5. Wait for a system event to be received. When such an event is received jump to the acquisition steps above.

> If the new position differs significantly from the previous position in the watch process, the *successCallback* is only invoked when a new position is obtained and this position differs significantly from the previously reported position. The definition of what constitutes a significant difference is left to the implementation. Furthermore, in steps 4.2.2 and 4.3, implementations *MAY* impose limitations on the frequency of callbacks so as to avoid inadvertently consuming a disproportionate amount of resources.

For both `getCurrentPosition` and `watchPosition`, the implementation *MUST* never invoke the *successCallback* without having first obtained permission from the user to share location. Furthermore, the implementation *SHOULD* always obtain the user's permission to share location before executing any of the `getCurrentPosition` or `watchPosition` steps described above. If the user grants permission, the appropriate callback *MUST* be invoked as described above. If the user denies permission, the *errorCallback* (if present) *MUST* be invoked with `code` `PERMISSION_DENIED`, irrespective of any other errors encountered in the above steps. The time that is spent obtaining the user permission *MUST NOT* be included in the period covered by the `timeout` attribute of the `PositionOptions` parameter. The `timeout` attribute *MUST* only apply to the location acquisition operation. ▶ **tests: 3**

The ***`clearWatch()`*** method takes one argument. When called, it *MUST* first check the value of the given ***watchId*** argument. If this value does not correspond to any previously started watch process, then the method *MUST* return immediately without taking any further action. Otherwise, the watch process identified by the `watchId` argument *MUST* be immediately stopped and no further callbacks *MUST* be invoked. ▶ **tests: 1**

5.3 ***`PositionOptions`*** interface

The `getCurrentPosition()` and `watchPosition()` methods accept `PositionOptions` objects as their third argument.

In ECMAScript, `PositionOptions` objects are represented using regular native objects with optional properties named `enableHighAccuracy`, `timeout` and `maximumAge`.

WebIDL

```
dictionary PositionOptions {
    boolean       enableHighAccuracy = false;
    [Clamp]
    unsigned long timeout = 0xFFFFFFFF;
    [Clamp]
    unsigned long maximumAge = 0;
};
```

In ECMAScript, the `enableHighAccuracy`, `timeout` and `maximumAge` properties are all *OPTIONAL*: when creating a PositionOptions object, the developer *MAY* specify any of these properties.

The ***`enableHighAccuracy`*** attribute provides a hint that the application would like to receive the best possible results. This *MAY* result in slower response times or increased power consumption. The user

might also deny this capability, or the device might not be able to provide more accurate results than if the flag wasn't specified. The intended purpose of this attribute is to allow applications to inform the implementation that they do not require high accuracy geolocation fixes and, therefore, the implementation can avoid using geolocation providers that consume a significant amount of power (e.g. GPS). This is especially useful for applications running on battery-powered devices, such as mobile phones. ▶ **tests: 1**

The ***`timeout`*** attribute denotes the maximum length of time (expressed in milliseconds) that is allowed to pass from the call to `getCurrentPosition` or `watchPosition` until the corresponding *successCallback* is invoked. If the implementation is unable to successfully acquire a new `Position` before the given timeout elapses, and no other errors have occurred in this interval, then the corresponding *errorCallback MUST* be invoked with a `PositionError` object whose code attribute is set to `TIMEOUT`. Note that the time that is spent obtaining the user permission is not included in the period covered by the `timeout` attribute. The `timeout` attribute only applies to the location acquisition operation. ▶ **tests: 1**

In case of a `getCurrentPosition()` call, the *errorCallback* would be invoked at most once.

In case of a `watchPosition()`, the *errorCallback* could be invoked repeatedly: the first timeout is relative to the moment `watchPosition()` was called or the moment the user's permission was obtained, if that was necessary. Subsequent timeouts are relative to the moment when the implementation determines that the position of the hosting device has changed and a new `Position` object *MUST* be acquired.

The ***`maximumAge`*** attribute indicates that the application is willing to accept a cached position whose age is no greater than the specified time in milliseconds. If `maximumAge` is set to 0, the implementation *MUST* immediately attempt to acquire a new position object. Setting the `maximumAge` to `Infinity` *MUST* determine the implementation to return a cached position regardless of its age. If an implementation does not have a cached position available whose age is no greater than the specified `maximumAge`, then it *MUST* acquire a new `Position` object. In case of a `watchPosition()`, the `maximumAge` refers to the first `Position` object returned by the implementation. ▶ **tests: 1**

5.4 ***`Position`*** interface

The `Position` interface is the container for the geolocation information returned by this API. This version of the specification allows one attribute of type `Coordinates` and a `timestamp`. Future versions of the API *MAY* allow additional attributes that provide other information about this position (e.g. street addresses).

WebIDL

```
[NoInterfaceObject]
interface Position {
    readonly attribute Coordinates  coords;
    readonly attribute DOMTimeStamp timestamp;
};
```

The ***coords*** attribute contains a set of geographic coordinates together with their associated accuracy, as well as a set of other optional attributes such as altitude and speed.

The ***timestamp*** attribute represents the time when the `Position` object was acquired and is represented as a ***DOMTimeStamp***.

5.5 ***Coordinates*** interface

WebIDL

```
[NoInterfaceObject]
interface Coordinates {
    readonly attribute double  latitude;
    readonly attribute double  longitude;
    readonly attribute double? altitude;
    readonly attribute double  accuracy;
    readonly attribute double? altitudeAccuracy;
    readonly attribute double? heading;
    readonly attribute double? speed;
};
```

The geographic coordinate reference system used by the attributes in this interface is the World Geodetic System (2d) [WGS84]. No other reference system is supported.

The ***latitude*** and ***longitude*** attributes are geographic coordinates specified in decimal degrees.

The ***altitude*** attribute denotes the height of the position, specified in meters above the [WGS84] ellipsoid. If the implementation cannot provide altitude information, the value of this attribute *MUST* be null.

The ***accuracy*** attribute denotes the accuracy level of the latitude and longitude coordinates. It is specified in meters and *MUST* be supported by all implementations. The value of the accuracy attribute *MUST* be a non-negative real number.

The ***altitudeAccuracy*** attribute is specified in meters. If the implementation cannot provide altitude information, the value of this attribute *MUST* be null. Otherwise, the value of the altitudeAccuracy attribute *MUST* be a non-negative real number.

The `accuracy` and `altitudeAccuracy` values returned by an implementation *SHOULD* correspond to a 95% confidence level.

The ***heading*** attribute denotes the direction of travel of the hosting device and is specified in degrees, where 0° ≤ heading < 360°, counting clockwise relative to the true north. If the implementation cannot provide heading information, the value of this attribute *MUST* be null. If the hosting device is stationary (i.e. the value of the `speed` attribute is 0), then the value of the heading attribute *MUST* be NaN.

The ***speed*** attribute denotes the magnitude of the horizontal component of the hosting device's current velocity and is specified in meters per second. If the implementation cannot provide speed information, the value of this attribute *MUST* be null. Otherwise, the value of the speed attribute *MUST* be a non-negative real number.

5.6 ***PositionError*** interface

WebIDL

```
[NoInterfaceObject]
interface PositionError {
    const unsigned short PERMISSION_DENIED = 1;
    const unsigned short POSITION_UNAVAILABLE = 2;
    const unsigned short TIMEOUT = 3;
    readonly attribute unsigned short code;
    readonly attribute DOMString      message;
};
```

The ***code*** attribute *MUST* return the appropriate code from the following list:

PERMISSION_DENIED **(numeric value 1)**
: The location acquisition process failed because the document does not have permission to use the Geolocation API.

POSITION_UNAVAILABLE **(numeric value 2)**
: The position of the device could not be determined. For instance, one or more of the location providers used in the location acquisition process reported an internal error that caused the

process to fail entirely.

TIMEOUT **(numeric value 3)**

The length of time specified by the `timeout` property has elapsed before the implementation could successfully acquire a new `Position` object.

The ***message*** attribute *MUST* return an error message describing the details of the error encountered. This attribute is primarily intended for debugging and developers *SHOULD NOT* use it directly in their application user interface.

6. Use-Cases and Requirements

This section is non-normative.

6.1 Use-Cases

6.1.1 Find points of interest in the user's area

Someone visiting a foreign city could access a Web application that allows users to search or browse through a database of tourist attractions. Using the Geolocation API, the Web application has access to the user's approximate position and it is therefore able to rank the search results by proximity to the user's location.

6.1.2 Annotating content with location information

A group of friends is hiking through the Scottish highlands. Some of them write short notes and take pictures at various points throughout the journey and store them using a Web application that can work offline on their hand-held devices. Whenever they add new content, the application automatically tags it with location data from the Geolocation API (which, in turn, uses the on-board GPS device). Every time they reach a town or a village, and they are again within network coverage, the application automatically uploads their notes and pictures to a popular blogging Web site, which uses the geolocation data to construct links that point to a mapping service. Users who follow the group's trip can click on these links to see a satellite view of the area where the notes were written and the pictures

were taken. Another example is a life blog where a user creates content (e.g. images, video, audio) that records her every day experiences. This content can be automatically annotated with information such as time, geographic position or even the user's emotional state at the time of the recording.

6.1.3 Show the user's position on a map

A user finds herself in an unfamiliar city area. She wants to check her position so she uses her hand-held device to navigate to a Web-based mapping application that can pinpoint her exact location on the city map using the Geolocation API. She then asks the Web application to provide driving directions from her current position to her desired destination.

6.1.4 Turn-by-turn route navigation

A mapping application can help the user navigate along a route by providing detailed turn-by-turn directions. The application does this by registering with the Geolocation API to receive repeated location updates of the user's position. These updates are delivered as soon as the implementing user agent determines that the position of the user has changed, which allows the application to anticipate any changes of direction that the user might need to do.

6.1.5 Alerts when points of interest are in the user's vicinity

A tour-guide Web application can use the Geolocation API to monitor the user's position and trigger visual or audio notifications when interesting places are in the vicinity. An online task management system can trigger reminders when the user is in the proximity of landmarks that are associated with certain tasks.

6.1.6 Up-to-date local information

A widget-like Web application that shows the weather or news that are relevant to the user's current area can use the Geolocation API to register for location updates. If the user's position changes, the widget can adapt the content accordingly.

6.1.7 Location-tagged status updates in social networking applications

A social network application allows its users to automatically tag their status updates with location information. It does this by monitoring the user's position with the Geolocation API. Each user can control the granularity of the location information (e.g. city or neighborhood level) that is shared with the other users. Any user can also track his network of friends and get real-time updates about their current location.

6.2 Requirements

1. The Geolocation API *MUST* provide location data in terms of a pair of latitude and longitude coordinates.
2. The Geolocation API *MUST* provide information about the accuracy of the retrieved location data.
3. The Geolocation API *MUST* support "one-shot" position updates.
4. The Geolocation API *MUST* allow an application to register to receive updates when the position of the hosting device changes.
5. The Geolocation API *MUST* allow an application to request a cached position whose age is no greater than a specified value.
6. The Geolocation API *MUST* provide a way for the application to receive updates about errors that *MAY* have occurred while obtaining a location fix.
7. The Geolocation API *MUST* allow an application to specify a desired accuracy level of the location information.
8. The Geolocation API *MUST* be agnostic to the underlying sources of location information.

A. Acknowledgments

Alec Berntson, Alissa Cooper, Steve Block, Greg Bolsinga, Lars Erik Bolstad, Aaron Boodman, Dave Burke, Chris Butler, Max Froumentin, Shyam Habarakada, Marcin Hanclik, Ian Hickson, Brad Lassey, Angel Machin, Cameron McCormack, Daniel Park, Stuart Parmenter, Olli Pettay, Chris Prince, Arun Ranganathan, Aza Raskin, Carl Reed, Thomas Roessler, Dirk Segers, Allan Thomson, Martin Thomson, Doug Turner, Erik Wilde, Matt Womer, Mohamed Zergaoui

B. IDL Index

WebIDL

```
partial interface Navigator {
    readonly attribute Geolocation geolocation;
};
[NoInterfaceObject]
interface Geolocation {
    void getCurrentPosition(PositionCallback successCallback,
                            optional PositionErrorCallback errorCallback,
                            optional PositionOptions options);

    long watchPosition(PositionCallback successCallback,
                       optional PositionErrorCallback errorCallback,
                       optional PositionOptions options);

    void clearWatch(long watchId);
};
callback PositionCallback = void (Position position);
callback PositionErrorCallback = void (PositionError positionError);
dictionary PositionOptions {
    boolean       enableHighAccuracy = false;
    [Clamp]
    unsigned long timeout = 0xFFFFFFFF;
    [Clamp]
    unsigned long maximumAge = 0;
};
[NoInterfaceObject]
interface Position {
    readonly attribute Coordinates  coords;
    readonly attribute DOMTimeStamp timestamp;
};
[NoInterfaceObject]
interface Coordinates {
    readonly attribute double  latitude;
    readonly attribute double  longitude;
    readonly attribute double? altitude;
    readonly attribute double  accuracy;
    readonly attribute double? altitudeAccuracy;
    readonly attribute double? heading;
    readonly attribute double? speed;
};
[NoInterfaceObject]
interface PositionError {
    const unsigned short PERMISSION_DENIED = 1;
```

```
    const unsigned short POSITION_UNAVAILABLE = 2;
    const unsigned short TIMEOUT = 3;
    readonly attribute unsigned short code;
    readonly attribute DOMString      message;
};
```

C. References

C.1 Normative references

[HTML]

HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[URL]

URL Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://url.spec.whatwg.org/

[WEBIDL]

Web IDL. Cameron McCormack; Boris Zbarsky; Tobie Langel. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

[WGS84]

National Imagery and Mapping Agency Technical Report 8350.2, Third Edition. National Imagery and Mapping Agency. 3 January 2000. URL: http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

C.2 Informative references

[AZALOC]

Geolocation in Firefox and Beyond. Aza Raskin.URL: http://www.azarask.in/blog/post/geolocation-in-firefox-and-beyond/

[DOM]

DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/