Geolocation API Specification

W3C Working Draft 22 December 2008

This version:
http://www.w3.org/TR/2008/WD-geolocation-API-20081222/
Latest published version:
http://www.w3.org/TR/geolocation-API/
Editors:
Andrei Popescu, Google, Inc

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 http://www.w3.org/TR/.

This document was published by the Geolocation Working Group. If you wish to make comments regarding this document, please send them to public-geolocation@w3.org (subscribe, archives).

Open issue notes are indicated in red boxes. These issues include: privacy considerations for implementors and users of the API, how accuracy and issues such as power consumption may be balanced in the API, and moving the Use-Cases and Requirements to a separate note.

All feedback is welcome.

Publication as a Working 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.

As noted above, readers should be aware that this is a First Public Working draft, and as such is not stable. **Implementors who are not taking part in the discussions are likely to find the specification changing out from under them in incompatible ways.** Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation stage should join the aforementioned mailing list and take part in the discussions.

This document was produced by a group operating under the 5 February 2004 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.

Table of Contents

Introduction

This section is non-normative.

The Geolocation API defines a high-level interface to location information associated with the hosting device, 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.

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], [GEARSLOC], and [LOCATIONAWARE].

The following code extract illustrates how to obtain basic location information:

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 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 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 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.
}
```

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 immediately 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.
      }
```

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:Infinite, timeout:0});

      function successCallback(position) {
        // By setting the 'maximumAge' to Infinite, 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.
      }
```

Scope

This section is non-normative.

This specification is limited to providing a scripting APIs 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 URI schemes for building URIs that identify geographic locations.

Security and privacy considerations

The API defined in this specification can be 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 without the user's informed consent.

Privacy considerations for implementors of the Geolocation API

This section is a placeholder for a set of recommendations addressed to the implementers of the Geolocation API. These recommendations would provide best practices and general advice on how to implement a mechanism that protects the user's privacy with respect to location information.

Privacy considerations for recipients of location information

This section is a placeholder for a set of recommendations addressed to the recipients of location information (e.g. Web sites that use the Geolocation API). These recommendations would provide best practices and general advice on how to deal with location information once it has been acquired from the User Agent by means of the Geolocation API.

While the above wording reflects the current thinking of the Geolocation WG, there is a significant amount of debate on whether this specification should leave the details of the privacy protection mechanism entirely to the User Agent or whether it should also define a privacy framework that all conforming User Agents must implement. Please see the following threads:

- http://lists.w3.org/Archives/Public/public-geolocation/2008Oct/0070.html
- http://lists.w3.org/Archives/Public/public-geolocation/2008Oct/0121.html

Conformance requirements

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC2119. For readability, these words do not appear in all uppercase letters in this specification. [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.

Implementations that use ECMAScript to implement the APIs defined in this specification must implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification, as this specification uses that specification's terminology.[WEBIDL]

API Description

Geolocation interface

The `Geolocation` object can be 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.

Objects implementing the `Navigator` interface (e.g. the `window.navigator` object) must also implement the `NavigatorGeolocation` interface. [NAVIGATOR]. An instance of `NavigatorGeolocation` would be then obtained by using binding-specific casting methods on an instance of`Navigator`.

```
[NoInterfaceObject]
interface NavigatorGeolocation {
  readonly attribute Geolocation geolocation;
};
```

```
interface Geolocation {
   void getCurrentPosition(in PositionCallback successCallback);
   void getCurrentPosition(in PositionCallback successCallback, in PositionErrorCallback errorCallback);
   void getCurrentPosition(in PositionCallback successCallback, in PositionErrorCallback errorCallback, in PositionOptions options)

   int watchPosition(in PositionCallback successCallback);
   int watchPosition(in PositionCallback successCallback, in PositionErrorCallback errorCallback);
   int watchPosition(in PositionCallback successCallback, in PositionErrorCallback errorCallback, in PositionOptions options);

   void clearWatch(in int watchId);
 };

 interface PositionCallback {
   void handleEvent(in Position position);
 };

 interface PositionErrorCallback {
   void handleEvent(in PositionError error);
 };
```

The **`getCurrentPosition()`** takes one, two or three arguments. When called, it must immediately return and then asynchronously acquire a new `Position` object. If successful, this method must invoke its associated `successCallback` argument with a `Position` object as an argument. If the attempt fails, and the method was invoked with a non-null `errorCallback` argument, this method must invoke the `errorCallback` with a `PositionError` object as an argument.

The **`watchPosition()`** takes one, two or three arguments. When called, it must immediately return and then asynchronously start a*watch process* defined as the following set of steps:

1. Acquire a new `Position` object. If successful, invoke the associated `successCallback` with a `Position` object as an argument. If the attempt fails, and the method was invoked with a non-null `errorCallback` argument, this method must invoke the `errorCallback` with a `PositionError` object as an argument.
2. Invoke the appropriate callback with a new `Position` object every time the implementation determines that the position of the hosting device has changed.

This method returns an integer value that uniquely identifies the watch process. When the **`clearWatch()`** method is called with this identifier, the watch process must stop acquiring any new position fixes and must cease invoking any callbacks.

PositionOptions interface

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

`PositionOptions` objects are regular ECMAScript objects that have the following properties:

```
[NoInterfaceObject]
interface PositionOptions {
  attribute boolean enableHighAccuracy;
  attribute long timeout;
  attribute long maximumAge;
};
```

The `enableHighAccuracy`, `timeout` and `maximumAge` attributes are all optional.

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.

See

- http://lists.w3.org/Archives/Public/public-geolocation/2008Jun/0094.html
- http://lists.w3.org/Archives/Public/public-geolocation/2008Jun/0107.html

for two descriptions of the problem that `enableHighAccuracy` is trying to solve.

The **`timeout`** attribute denotes the maximum length of time (expressed in milliseconds) that is allowed to pass from the 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.

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

In case of a `watchPosition()`, the `errorCallback` could be invoked repeatedly: the first timeout is relative to the moment `watchPosition()` was called, while 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 not specified or set to 0, the implementation must immediately attempt to acquire a new position object. Setting the `maximumAge` to Infinite will force the implementation to return a cached position regardless of its age. If an implementation does not have available a cached position 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.

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).

```
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 [DOMTIMESTAMP].

Coordinates interface

```
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 **`altitudeAccuracy`** attribute is specified in meters. If the implementation cannot provide altitude information, the value of this attribute must be null.

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 counting clockwise relative to the true north. If the implementation cannot provide heading information, the value of this attribute must be null.

The **speed** attribute denotes the current ground speed of the hosting device and is specified in meters per second. If the implementation cannot provide speed information, the value of this attribute must be null.

PositionError interface

```
interface PositionError {
  const unsigned short UNKNOWN_ERROR = 0;
  const unsigned short PERMISSION_DENIED = 1;
  const unsigned short POSITION_UNAVAILABLE = 2;
  const unsigned short TIMEOUT = 3;
  readonly unsigned short code;
  readonly DOMString message;
};
```

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

UNKNOWN_ERROR (numeric value 0)
The location acquisition process failed due to an error not covered by the definition of any other error code in this interface.

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

POSITION_UNAVAILABLE (numeric value 2)
The position of the device could not be determined. 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 specified maximum length of time 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.

Use-Cases and Requirements

Issue note: This section will get moved to a separate document

Use-Cases

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.

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.

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.

Turn-by-turn route navigation

Following from use-case 4, 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.

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.

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.

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 and using only certain parts from the Address object. Each user can control the granularity of the location information (e.g. city or neighbourhood 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.

Requirements

The Geolocation API must provide location data in terms of a pair of latitude and longitude coordinates.

The Geolocation API must provide information about the accuracy of the retrieved location data.

The Geolocation API must support "one-shot" position updates.

The Geolocation API must allow an application to register to receive repeated position updates.

The Geolocation API must allow an application to cheaply query the last known position.

The Geolocation API must provide a way for the application to receive updates about errors that may have occurred while obtaining a location fix.

The Geolocation API must allow an application to specify a desired accuracy level of the location information.

The Geolocation API must be agnostic to the underlying sources of location information.

Acknowledgments

Alec Berntson, Steve Block, Greg Bolsinga, Aaron Boodman, Dave Burke, Chris Butler, Shyam Habarakada, Ian Hickson, Brad Lassey, Daniel Park, Stuart Parmenter, Olli Pettay, Chris Prince, Arun Ranganathan, Aza Raskin, Carl Reed, Martin Thomson, Doug Turner, Erik Wilde, Mohamed Zergaoui

References

[AZALOC]
(Non-normative) Geolocation in Firefox and Beyond. See http://azarask.in/blog/post/geolocation-in-firefox-and-beyond
[NAVIGATOR]
Navigator interface in HTML5. See http://www.whatwg.org/specs/web-apps/current-work/#navigator
[DOMTIMESTAMP]
The DOMTimeStamp Type. See http://www.w3.org/TR/DOM-Level-3-Core/core.html#Core-DOMTimeStamp
[GEARSLOC]
(Non-normative) Gears Geolocation API. See http://code.google.com/p/google-gears/wiki/GeolocationAPI
[LOCATIONAWARE]
(Non-normative) LocationAware.org Working Draft. See http://locationaware.org/wiki/index.php?title=Working_Draft
[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. See http://www.ietf.org/rfc/rfc2119.txt
[WEBIDL]
Web IDL. See http://www.w3.org/TR/WebIDL/
[WGS84]
National Imagery and Mapping Agency Technical Report 8350.2, Third Edition. See http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf