Notifications API

Living Standard — Last Updated 18 January 2021

Participate:

GitHub whatwg/notifications (new issue, open issues)
IRC: #whatwg on Freenode

Commits:

GitHub whatwg/notifications/commits
Snapshot as of this commit
@notifyapi

Tests:

web-platform-tests notifications/ (ongoing work)

Translations (non-normative):

日本語
简体中文

Abstract

This standard defines an API to display notifications to the end user, typically outside the top-level browsing context’s viewport. It is designed to be compatible with existing notification systems, while remaining platform-independent.

Table of Contents

§ 1. Terminology

This specification depends on the Infra Standard. [INFRA]

Some terms used in this specification are defined in the DOM, Fetch, HTML, IDL, Service Workers, URL, and Vibration API Standards. [DOM] [FETCH] [HTML] [WEBIDL] [SERVICE-WORKERS] [URL] [VIBRATION]

§ 2. Notifications

A **notification** is an abstract representation of something that happened, such as the delivery of a message.

A notification *can* have an associated **service worker registration**.

A notification has an associated **title** which is a DOMString.

A notification has an associated **body** which is a DOMString.

A notification has an associated **direction** which is one of *auto*, *ltr*, and *rtl*.

A notification has an associated **language** which is a DOMString representing either a valid BCP 47 language tag or the empty string.

A notification has an associated **tag** which is a DOMString.

A notification has an associated **data**.

A notification has an associated **timestamp** which is a `DOMTimeStamp` representing the time, in milliseconds since 00:00:00 UTC on 1 January 1970, of the event for which the notification was created.

Note

Timestamps can be used to indicate the time at which a notification is actual. For example, this could be in the past when a notification is used for a message that couldn't immediately be delivered because the device was offline, or in the future for a meeting that is about to start.

A notification has an associated **origin**.

A notification has an associated **renotify preference flag** which is initially unset. When set indicates that the end user should be alerted after the show steps have run with a new notification that has the same tag as an existing notification.

A notification has an associated **silent preference flag** which is initially unset. When set indicates that no sounds or vibrations should be made.

A notification has an associated **require interaction preference flag** which is initially unset. When set, indicates that on devices with a sufficiently large screen, the notification should remain readily available until the user activates or dismisses the notification.

A notification *can* have these associated graphics: an **image URL**, **icon URL**, and **badge URL**; and their corresponding image resource, icon resource, and badge resource.

An **image resource** is a picture shown as part of the content of the notification, and should be displayed with higher visual priority than the icon resource and badge resource, though it may be displayed in fewer circumstances.

An **icon resource** is an image that reinforces the notification (such as an icon, or a photo of the sender).

A **badge resource** is an icon representing the web application, or the category of the notification if the web application sends a wide variety of notifications. It *may* be used to represent the notification when there is not enough space to display the notification itself. It *may* also be displayed inside the notification, but then it should have less visual priority than the image resource and icon resource.

A notification *can* have a **vibration pattern**.

Note

Developers are encouraged to not convey information through an image, icon, badge, or vibration pattern that is not otherwise accessible to the end user, especially since notification platforms that do not support these features might ignore them.

A notification has an associated list of zero or more **actions**. Each action has an associated **title** and **name** and *can* have an associated **icon URL** and **icon resource**. Users may activate actions, as alternatives to activating the notification itself. The user agent must determine the **maximum number of actions** supported, within the constraints of the notification platform.

Note

Since display of actions is platform-dependent, developers are encouraged to make sure that any action a user can invoke from a notification is also available within the web application.

Note

Some platforms might modify an icon resource to better match the platform's visual style before displaying it to the user, for example by rounding the corners or painting it in a specific color. Developers are encouraged to use an icon that handles such cases gracefully and does not lose important information through, e.g., loss of color or clipped corners.

A **non-persistent notification** is a notification without an associated service worker registration.

A **persistent notification** is a notification with an associated service worker registration.

To **create a notification**, given a *title*, *options*, and optionally a *serviceWorkerRegistration*, run these steps:

1. Let *notification* be a new notification.
2. If a *serviceWorkerRegistration* was provided, set *notification*'s service worker registration to *serviceWorkerRegistration*.
3. If a *serviceWorkerRegistration* was not provided and *options*'s `actions` is not empty, throw a `TypeError` exception.

   Note

   Actions are only currently supported for persistent notifications.

4. If *options*'s `silent` is true and *options*'s `vibrate` is present, throw a `TypeError` exception.
5. If *options*'s `renotify` is true and *options*'s `tag` is the empty string, throw a

`TypeError` exception.

6. Set *notification*'s data to StructuredSerializeForStorage(*options*'s `data`). Rethrow any exceptions.

7. Set *notification*'s title to *title*.

8. Set *notification*'s direction to *options*'s `dir`.

9. Set *notification*'s language to *options*'s `lang`.

10. Set *notification*'s origin to the entry settings object's origin.

11. Set *notification*'s body to *options*'s `body`.

12. Set *notification*'s tag to *options*'s `tag`.

13. Let *baseURL* be the API base URL specified by the entry settings object.
    Or incumbent?

14. If *options*'s `image` is present, parse it using *baseURL*, and if that does not return failure, set *notification*'s image URL to the return value. (Otherwise image URL is not set.)

15. If *options*'s `icon` is present, parse it using *baseURL*, and if that does not return failure, set *notification*'s icon URL to the return value. (Otherwise icon URL is not set.)

16. If *options*'s `badge` is present, parse it using *baseURL*, and if that does not return failure, set *notification*'s badge URL to the return value. (Otherwise badge URL is not set.)

17. If *options*'s `vibrate` is present, validate and normalize it and set *notification*'s vibration pattern to the return value. (Otherwise vibration pattern is not set.)

18. If *options*'s `timestamp` is present, set *notification*'s timestamp to the value. Otherwise, set *notification*'s timestamp to the number of milliseconds that passed between 00:00:00 UTC on 1 January 1970 and the time at which the `Notification` constructor was called.

19. If *options*'s `renotify` is true, set *notification*'s renotify preference flag.

20. If *options*'s `silent` is true, set *notification*'s silent preference flag.

21. If *options*'s `requireInteraction` is true, set *notification*'s require interaction preference flag.

22. Set *notification*'s list of actions to an empty list, then for each *entry* in *options*'s `actions`, up to the maximum number of actions supported (skip any excess entries), perform the following steps:

    1. Let *action* be a new action.

    2. Set *action*'s name to the *entry*'s `action`.

    3. Set *action*'s title to the *entry*'s `title`.

    4. If *entry*'s `icon` is present, parse it using *baseURL*, and if that does not return failure, set *action*'s icon URL to the return value. (Otherwise icon URL is not set.)

    5. Append *action* to *notification*'s list of actions.

23. Return *notification*.

§ 2.1. Lifetime and UI integration

The user agent must keep a **list of notifications**, which is a list of zero or more notifications.

User agents should run the close steps for a non-persistent notification a couple of seconds after they have been created.

User agents should not display non-persistent notification in a platform’s "notification center" (if available).

User agents should persist persistent notifications until they are removed from the list of notifications.

¶ Example

A persistent notification could have the `close()` method invoked of one of its `Notification` objects.

User agents should display persistent notifications in a platform’s "notification center" (if available).

§ 2.2. Permission model

Notifications can only be displayed if the user (or user agent on behalf of the user) has granted **permission**. The permission to show notifications for a given origin is one of three strings:

`"default"`

This is equivalent to "`denied`", but the user has made no explicit choice thus far.

`"denied"`

This means the user does not want notifications.

`"granted"`

This means notifications can be displayed.

Note

There is no equivalent to "`default`" *meaning* "`granted`"*. In that case* "`granted`" *is simply returned as there would be no reason for the application to ask for permission.*

§ 2.3. Direction

This section is written in terms equivalent to those used in the Rendering section of HTML. [HTML]

User agents are expected to honor the Unicode semantics of the text of a notification’s title, body, and the title of each of its actions. Each is expected to be treated as an independent set of one or more bidirectional algorithm paragraphs when displayed, as defined by the bidirectional algorithm’s rules P1, P2, and P3, including, for instance, supporting the paragraph-breaking behavior of U+000A LINE FEED (LF) characters. For each paragraph of the title, body and the title of each of the actions, the notification’s direction provides the higher-level override of rules P2 and P3 if it has a value other than "`auto`". [BIDI]

The notification’s direction also determines the relative order in which the notification’s actions should be displayed to the user, if the notification platform displays them side by side.

§ 2.4. Language

The notification’s language specifies the primary language for the notification’s title, body and the title of each of its actions. Its value is a string. The empty string indicates that the primary language is unknown. Any other string must be interpreted as a language tag. Validity or well-formedness are not enforced. [LANG]

Note

Developers are encouraged to only use valid language tags.

§ 2.5. Resources

The **fetch steps** for a given notification *notification* are:

1. If the notification platform supports images, fetch *notification*’s image URL, if image URL is set.

   Note

   The intent is to fetch this resource similar to an `<img>`, but this needs abstracting.

   Then, in parallel:

   1. Wait for the response.
   2. If the response’s internal response’s type is "`default`", then attempt to decode the resource as image.
   3. If the image format is supported, set *notification*’s image resource to the decoded resource. (Otherwise *notification* has no image resource.)

2. If the notification platform supports icons, fetch *notification*’s icon URL, if icon URL is set.

Note

The intent is to fetch this resource similar to an `<img>`*, but this needs abstracting.*

Then, in parallel:

1. Wait for the response.
2. If the response's internal response's type is "`default`", then attempt to decode the resource as image.
3. If the image format is supported, set *notification*'s icon resource to the decoded resource. (Otherwise *notification* has no icon resource.)

3. If the notification platform supports badges, fetch *notification*'s badge URL, if badge URL is set.

Note

The intent is to fetch this resource similar to an `<img>`*, but this needs abstracting.*

Then, in parallel:

1. Wait for the response.
2. If the response's internal response's type is "`default`", then attempt to decode the resource as image.
3. If the image format is supported, set *notification*'s badge resource to the decoded resource. (Otherwise *notification* has no badge resource.)

4. If the notification platform supports actions and action icons, then for each *action* in *notification*'s list of actions fetch *action*'s icon URL, if icon URL is set.

Note

The intent is to fetch this resource similar to an `<img>`*, but this needs abstracting.*

Then, in parallel:

1. Wait for the response.
2. If the response's internal response's type is "`default`", then attempt to decode the resource as image.
3. If the image format is supported, set *action*'s icon resource to the decoded resource. (Otherwise *action* has no icon resource.)

§ 2.6. Showing a notification

The **show steps** for a given notification *notification* are:

1. Wait for any fetches to complete and *notification*'s image resource icon resource, and badge resource to be set (if any), as well as the icon resources for the *notification*'s actions (if any).
2. Let *shown* be false.

3. Let *oldNotification* be the notification in the list of notifications whose tag is not the empty string and is *notification*’s tag, and whose origin is same origin with *notification*’s origin, if any, and null otherwise.

4. If *oldNotification* is non-null, then:

    1. Handle close events with *oldNotification*.

    2. If the notification platform supports replacement, then:

        1. Replace *oldNotification* with *notification*, in the list of notifications.

        2. Set *shown* to true.

    Note

    Notification platforms are strongly encouraged to support native replacement as it leads to a better user experience.

    3. Otherwise, remove *oldNotification* from the list of notifications.

5. If *shown* is false, then:

    1. Append *notification* to the list of notifications.

    2. Display *notification* on the device (e.g., by calling the appropriate notification platform API).

6. If *shown* is false or *oldNotification* is non-null and *notification*’s renotify preference flag has been set, then run the alert steps for *notification*.

7. If *notification* is a non-persistent notification, then queue a task to fire an event named `show` on the `Notification` object representing *notification*.

§ 2.7. Activating a notification

When a notification *notification*, or one of its actions, is activated by the user, assuming the underlying notification platform supports activation, the user agent must (unless otherwise specified) run these steps:

1. If *notification* is a persistent notification, then:

    1. Let *action* be the empty string.

    2. If one of *notification*’s actions was activated by the user, then set *action* to that action’s name.

    3. Fire a service worker notification event named "`notificationclick`" given *notification* and *action*.

2. Otherwise, queue a task to run these steps:

    1. Let *intoFocus* be the result of firing an event named `click` on the `Notification` object representing *notification*, with its `cancelable` attribute initialized to true.

    Note

    User agents are encouraged to make `focus()` work from within the *event listener for the event named* `click`*.*

2. If *intoFocus* is true, then the user agent should bring the *notification*’s related browsing context’s viewport into focus.

Note

Throughout the web platform "activate" is intentionally misnamed as "click".

§ 2.8. Closing a notification

When a notification is closed, either by the underlying notification platform or by the user, the close steps for it must be run.

The **close steps** for a given *notification* are:

1. If the list of notifications does not contain *notification*, then abort these steps.
2. Handle close events with *notification*.
3. Remove *notification* from the list of notifications.

To **handle close events** given a *notification*, run these steps:

1. If *notification* is a persistent notification and *notification* was closed by the user, then fire a service worker notification event named "`notificationclose`" given *notification*.
2. If *notification* is a non-persistent notification, then queue a task to fire an event named `close` on the `Notification` object representing *notification*.

§ 2.9. Alerting the user

The **alert steps** for alerting the user about a given *notification* are:

1. Perform vibration using *notification*’s vibration pattern, if any.

§ 3. API

```
[Exposed=(Window,Worker)]
interface Notification : EventTarget {
  constructor(DOMString title, optional NotificationOptions
options = {});

  static readonly attribute NotificationPermission permission;
  [Exposed=Window] static Promise<NotificationPermission>
requestPermission(optional NotificationPermissionCallback
deprecatedCallback);

  static readonly attribute unsigned long maxActions;

  attribute EventHandler onclick;
  attribute EventHandler onshow;
  attribute EventHandler onerror;
  attribute EventHandler onclose;

  readonly attribute DOMString title;
  readonly attribute NotificationDirection dir;
  readonly attribute DOMString lang;
  readonly attribute DOMString body;
  readonly attribute DOMString tag;
  readonly attribute USVString image;
  readonly attribute USVString icon;
  readonly attribute USVString badge;
  [SameObject] readonly attribute FrozenArray<unsigned long>
vibrate;
  readonly attribute DOMTimeStamp timestamp;
  readonly attribute boolean renotify;
  readonly attribute boolean silent;
  readonly attribute boolean requireInteraction;
  [SameObject] readonly attribute any data;
  [SameObject] readonly attribute
FrozenArray<NotificationAction> actions;

  undefined close();
};

dictionary NotificationOptions {
  NotificationDirection dir = "auto";
  DOMString lang = "";
  DOMString body = "";
  DOMString tag = "";
  USVString image;
  USVString icon;
  USVString badge;
  VibratePattern vibrate;
  DOMTimeStamp timestamp;
  boolean renotify = false;
  boolean silent = false;
  boolean requireInteraction = false;
  any data = null;
  sequence<NotificationAction> actions = [];
};

enum NotificationPermission {
  "default",
  "denied",
  "granted"
};

enum NotificationDirection {
  "auto",
  "ltr",
  "rtl"
};

dictionary NotificationAction {
  required DOMString action;
  required DOMString title;
  USVString icon;
};

callback NotificationPermissionCallback = undefined
(NotificationPermission permission);
```

A non-persistent notification is represented by one `Notification` object and can be created through `Notification`'s constructor.

A persistent notification is represented by zero or more `Notification` objects and can be created through the `showNotification()` method.

§ 3.1. Garbage collection

A `Notification` object must not be garbage collected while the list of notifications contains its corresponding notification and it has an event listener whose **type** is `click`, `show`, `close`, or `error`.

§ 3.2. Constructors

The **`Notification(title, options)`** constructor, when invoked, must run these steps:

1. If the current global object is a `ServiceWorkerGlobalScope` object, then throw a `TypeError` exception.

2. Let *notification* be the result of creating a notification given *title* and *options*. Rethrow any exceptions.

3. Let *n* be a new `Notification` object associated with *notification*.

4. Run these steps in parallel:

    1. If permission for *notification*'s origin is not "`granted`", then queue a

task to fire an event named `error` on *n*, and abort these steps.

    2. Run the fetch steps for *notification*.

    3. Run the show steps for *notification*.

5. Return *n*.

§ 3.3. Static members

The static **`permission`** attribute’s getter must return permission for the entry settings object’s origin.

Note

If you edit standards please refrain from copying the above. Synchronous permissions are like synchronous IO, a bad idea.

The static **`requestPermission(`*`deprecatedCallback`*`)`** method, when invoked, must run these steps:

1. Let *promise* be a new promise.

2. Run these steps in parallel:

    1. Let *permission* be permission for entry settings object’s origin.

    2. If *permission* is "`default`", ask the user whether showing notifications for the entry settings object’s origin is acceptable. If it is, set *permission* to "`granted`", and "`denied`" otherwise.

    3. Queue a task to run these steps:

        1. Set permission for the entry settings object’s origin to *permission*.

        2. If *deprecatedCallback* is given, invoke *deprecatedCallback* with *permission* as single argument. If this throws an exception, report the exception.

        3. Fullfil *promise* with *permission*.

3. Return *promise*.

⚠Warning!

Notifications are the one instance thus far where asking the user upfront makes sense. Specifications for other APIs should not use this pattern and instead employ one of the many more suitable alternatives.

The static **`maxActions`** attribute’s getter must return the maximum number of actions supported.

§ 3.4. Object members

The following are the event handlers (and their corresponding event handler event types) that must be supported as attributes by the `Notification` object.

event handler	**event handler event type**
`onclick`	`click`
`onshow`	`show`
`onerror`	`error`
`onclose`	`close`

The **`close()`** method, when invoked, must run the close steps for the notification.

The **`title`** attribute’s getter must return the notification’s title.

The **`dir`** attribute’s getter must return the notification’s direction.

The **`lang`** attribute’s getter must return the notification’s language.

The **`body`** attribute’s getter must return the notification’s body.

The **`tag`** attribute’s getter must return the notification’s tag.

The **`image`** attribute’s getter must return the notification’s image URL, serialized, and the empty string if there is no notification’s image URL otherwise.

The **`icon`** attribute’s getter must return the notification’s icon URL, serialized, and the empty string if there is no notification’s icon URL otherwise.

The **`badge`** attribute’s getter must return the notification’s badge URL, serialized, and the empty string if there is no notification’s badge URL otherwise.

The **`vibrate`** attribute’s getter must return the notification’s vibration pattern, if any, and the empty list otherwise.

The **`timestamp`** attribute’s getter must return the notification’s timestamp.

The **`renotify`** attribute’s getter must return the notification’s renotify preference flag.

The **`silent`** attribute’s getter must return the notification’s silent preference flag.

The **`requireInteraction`** attribute’s getter must return the notification’s require interaction preference flag.

The **`data`** attribute’s getter must return StructuredDeserialize(notification’s data, context object’s relevant Realm). If this throws an exception, then return null.

The **`actions`** attribute’s getter must return the result of the following steps:

1. Let *frozenActions* be an empty list of type `NotificationAction`.
2. For each *entry* in the notification’s list of actions, perform the following steps:
    1. Let *action* be a new `NotificationAction`.
    2. Set *action*’s `action` to *entry*’s name.
    3. Set *action*’s `title` to *entry*’s title.
    4. Set *action*’s `icon` to *entry*’s icon URL.

        5. Call Object.freeze on *action*, to prevent accidental mutation by scripts.

        6. Append *action* to *frozenActions*.

    3. Create a frozen array from *frozenActions*.

§ 3.5. Examples

§ 3.5.1. Using events from a page

Non-persistent `Notification` objects dispatch events during their lifecycle, which developers can use to generate desired behaviors.

The `click` event dispatches when the user activates a notification.

¶ Example

```
var not = new Notification("Gebrünn Gebrünn by Paul Kalkbrenner", {
icon: "newsong.svg", tag: "song" });
not.onclick = function() { displaySong(this); };
```

§ 3.5.2. Using actions from a service worker

Persistent notifications fire `notificationclick` events on the `ServiceWorkerGlobalScope`.

Here a service worker shows a notification with a single "Archive" action, allowing users to perform this common task from the notification without having to open the website (for example the notification platform might show a button on the notification). The user can also activate the main body of the notification to open their inbox.

¶ Example

```
self.registration.showNotification("New mail from Alice", {
  actions: [{action: 'archive', title: "Archive"}]
});

self.addEventListener('notificationclick', function(event) {
  event.notification.close();
  if (event.action === 'archive') {
    silentlyArchiveEmail();
  } else {
    clients.openWindow("/inbox");
  }
}, false);
```

§ 3.5.3. Using the `tag` member for multiple instances

Web applications frequently operate concurrently in multiple instances, such as when a user opens a mail application in multiple browser tabs. Since the desktop is a shared resource, the notifications API provides a way for these instances to easily coordinate, by using the `tag` member.

Notifications which represent the same conceptual event can be tagged in the same way, and when both are shown, the user will only receive one notification.

¶ Example

```
Instance 1                                   | Instance 2
                                             |
// Instance notices there is new mail.       |
new Notification("New mail from John Doe",   |
                 { tag: 'message1' });       |
                                             |
                                             |  // Slightly later,
this instance notices
                                             |  // there is new mail.
                                             |  new Notification("New
mail from John Doe",
                                             |                  {
tag: 'message1' });
```

The result of this situation, if the user agent follows the algorithms here, is a **single** notification "New mail from John Doe".

§ 3.5.4. Using the tag member for a single instance

The `tag` member can also be used by a single instance of an application to keep its notifications as current as possible as state changes.

For example, if Alice is using a chat application with Bob, and Bob sends multiple messages while Alice is idle, the application may prefer that Alice not see a desktop notification for each message.

¶ Example

```
// Bob says "Hi"
new Notification("Bob: Hi", { tag: 'chat_Bob' });

// Bob says "Are you free this afternoon?"
new Notification("Bob: Hi / Are you free this afternoon?", { tag:
'chat_Bob' });
```

The result of this situation is a *single* notification; the second one replaces the first having the same tag. In a platform that queues notifications (first-in-first-out), using the tag allows the notification to also maintain its position in the queue. Platforms where the newest notifications are shown first, a similar result could be achieved using the `close()` method.

§ 4. Service worker API

```
dictionary GetNotificationOptions {
  DOMString tag = "";
};

partial interface ServiceWorkerRegistration {
  Promise<undefined> showNotification(DOMString title,
optional NotificationOptions options = {});
  Promise<sequence<Notification>> getNotifications(optional
GetNotificationOptions filter = {});
};

[Exposed=ServiceWorker]
interface NotificationEvent : ExtendableEvent {
  constructor(DOMString type, NotificationEventInit
eventInitDict);

  readonly attribute Notification notification;
  readonly attribute DOMString action;
};

dictionary NotificationEventInit : ExtendableEventInit {
  required Notification notification;
  DOMString action = "";
};

partial interface ServiceWorkerGlobalScope {
  attribute EventHandler onnotificationclick;
  attribute EventHandler onnotificationclose;
};
```

The **`showNotification(title, options)`** method, when invoked, must run these steps:

1. Let *promise* be a new promise.

2. If the context object's active worker is null, then reject *promise* with a `TypeError` exception and return *promise*.

3. Let *serviceWorkerRegistration* be the context object.

4. Let *notification* be the result of creating a notification given *title*, *options*, and *serviceWorkerRegistration*. If this threw an exception, reject *promise* with that exception and return *promise*.

5. Run these steps in parallel:

    1. If permission for *notification*'s origin is not "`granted`", then reject *promise* with a `TypeError` exception, and abort these steps.

    2. Run the fetch steps for *notification*.

    3. Run the show steps for *notification*.

    4. Resolve *promise* with undefined.

6. Return *promise*.

The **`getNotifications(filter)`** method, when invoked, must run these steps:

1. Let *promise* be a new promise.
2. Run these steps in parallel:
    1. Let *tag* be *filter*'s `tag`.
    2. Let *notifications* be a list of all notifications in the list of notifications whose origin is the entry settings object's origin, whose service worker registration is the context object, and whose tag, if *tag* is not the empty string, is *tag*.
    3. Let *objects* be an empty JavaScript array.
    4. For each notification in *notifications*, in creation order, create a new `Notification` object representing notification and push that object to *objects*.
    5. Resolve *promise* with *objects*.
3. Return *promise*.

Note

This method returns zero or more new `Notification` objects which might represent the same underlying notification of `Notification` objects already in existence.

To **fire a service worker notification event** named *name* given *notification* (a notification), and an optional *action* (a DOMString, defaulting to the empty string), Fire Functional Event *name* using `NotificationEvent` on *notification*'s service worker registration with the following properties:

`notification`
: A new `Notification` object representing *notification*.

`action`
: *action*

The `notification` attribute's getter must return the value it was initialized to.

The `action` attribute's getter must return the value it was initialized to.

The following is the event handler (and its corresponding event handler event type) that must be supported as attribute by the `ServiceWorkerGlobalScope` object:

event handler	**event handler event type**
`onnotificationclick`	`notificationclick`
`onnotificationclose`	`notificationclose`

§ Acknowledgments

Thanks to Addison Phillips, Aharon (Vladimir) Lanin, Alex Russell, Anssi Kostiainen, Arkadiusz Michalski, Boris Zbarsky, David Håsäther, Doug Turner, Drew Wilson, Ehsan Akhgari, Frederick Hirsch, Ian Hickson, Jake Archibald, James Graham, John Mellor, Jon Lee, Jonas Sicking, Michael Cooper, Michael Henretty, Michael™ Smith, Michael van Ouwerkerk, Nicolás Satragno, Olli Pettay, Peter Beverloo, Philip Jägenstedt, Reuben Morais, Rich Tibbett, Robert Bindar, 박상현 (Sanghyun Park), Simon Pieters, Theresa O'Connor, timeless, and triple-underscore for being awesome.

This standard is written by Anne van Kesteren (Mozilla, annevk@annevk.nl). An earlier iteration was written by John Gregg (Google, johnnyg@google.com).

§ Intellectual property rights

§ Index

§ Terms defined by this specification

§ Terms defined by reference

- [DOM] defines the following terms:
    - EventTarget
    - cancelable
    - context object
    - event listener
    - fire an event
- [FETCH] defines the following terms:
    - fetch
    - internal response
    - response
    - type
- [HTML] defines the following terms:
    - EventHandler
    - StructuredDeserialize
    - StructuredSerializeForStorage
    - browsing context
    - current global object
    - event handler
    - event handler event type
    - focus()
    - in parallel
    - origin (for environment settings object)
    - queue a task
    - relevant realm
    - report the exception
    - same origin
- [INFRA] defines the following terms:

    - append
    - contain
    - for each
    - list
    - remove
    - replace
- [SERVICE-WORKERS] defines the following terms:
    - ExtendableEvent
    - ExtendableEventInit
    - ServiceWorkerGlobalScope
    - ServiceWorkerRegistration
    - active worker
    - fire functional event
- [URL] defines the following terms:
    - url parser
    - url serializer
- [WEBIDL] defines the following terms:
    - DOMString
    - DOMTimeStamp
    - Exposed
    - FrozenArray
    - Promise
    - SameObject
    - USVString
    - any
    - boolean
    - create a frozen array
    - sequence
    - throw
    - undefined
    - unsigned long

§ References

§ Normative References

[BIDI]
: Mark Davis; Aharon Lanin; Andrew Glass. Unicode Bidirectional Algorithm. 12 February 2020. Unicode Standard Annex #9. URL: https://www.unicode.org/reports/tr9/tr9-42.html

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

[FETCH]
: Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[HTML]
: Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
: Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[LANG]
: A. Phillips; M. Davis. Tags for Identifying Languages. September 2009. IETF Best Current Practice. URL: https://tools.ietf.org/html/bcp47

[SERVICE-WORKERS]
: Alex Russell; et al. Service Workers 1. URL: https://w3c.github.io/ServiceWorker/

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

[VIBRATION]
: Anssi Kostiainen. Vibration API (Second Edition). URL: https://w3c.github.io/vibration/

[WEBIDL]
: Boris Zbarsky. Web IDL. URL: https://heycam.github.io/webidl/

§ IDL Index

```
[Exposed=(Window,Worker)]
interface Notification : EventTarget {
  constructor(DOMString title, optional NotificationOptions
options = {});

  static readonly attribute NotificationPermission permission;
  [Exposed=Window] static Promise<NotificationPermission>
requestPermission(optional NotificationPermissionCallback
deprecatedCallback);

  static readonly attribute unsigned long maxActions;

  attribute EventHandler onclick;
  attribute EventHandler onshow;
  attribute EventHandler onerror;
  attribute EventHandler onclose;

  readonly attribute DOMString title;
  readonly attribute NotificationDirection dir;
  readonly attribute DOMString lang;
  readonly attribute DOMString body;
  readonly attribute DOMString tag;
  readonly attribute USVString image;
  readonly attribute USVString icon;
  readonly attribute USVString badge;
  [SameObject] readonly attribute FrozenArray<unsigned long>
vibrate;
  readonly attribute DOMTimeStamp timestamp;
  readonly attribute boolean renotify;
  readonly attribute boolean silent;
  readonly attribute boolean requireInteraction;
  [SameObject] readonly attribute any data;
  [SameObject] readonly attribute
FrozenArray<NotificationAction> actions;

  undefined close();
};

dictionary NotificationOptions {
  NotificationDirection dir = "auto";
  DOMString lang = "";
  DOMString body = "";
  DOMString tag = "";
  USVString image;
  USVString icon;
  USVString badge;
  VibratePattern vibrate;
  DOMTimeStamp timestamp;
  boolean renotify = false;
  boolean silent = false;
  boolean requireInteraction = false;
  any data = null;
  sequence<NotificationAction> actions = [];
};

enum NotificationPermission {
  "default",
  "denied",
  "granted"
};

enum NotificationDirection {
  "auto",
  "ltr",
  "rtl"
};

dictionary NotificationAction {
  required DOMString action;
  required DOMString title;
  USVString icon;
};

callback NotificationPermissionCallback = undefined
(NotificationPermission permission);

dictionary GetNotificationOptions {
  DOMString tag = "";
};

partial interface ServiceWorkerRegistration {
  Promise<undefined> showNotification(DOMString title,
optional NotificationOptions options = {});
  Promise<sequence<Notification>> getNotifications(optional
GetNotificationOptions filter = {});
};

[Exposed=ServiceWorker]
interface NotificationEvent : ExtendableEvent {
  constructor(DOMString type, NotificationEventInit
eventInitDict);

  readonly attribute Notification notification;
  readonly attribute DOMString action;
};

dictionary NotificationEventInit : ExtendableEventInit {
  required Notification notification;
  DOMString action = "";
};

partial interface ServiceWorkerGlobalScope {
  attribute EventHandler onnotificationclick;
  attribute EventHandler onnotificationclose;
};
```