Quota Management API

W3C Working Group Note 23 May 2016

This version:
http://www.w3.org/TR/2016/NOTE-quota-api-20160523/

Latest published version:
http://www.w3.org/TR/quota-api/

Latest editor's draft:
http://w3c.github.io/quota-api/

Previous version:
http://www.w3.org/TR/2016/NOTE-quota-api-20160515/

Editor:
Kinuko Yasuda, Google

Repository and Participation:
We are on Github.
File a bug/issue.
Commit history.
Mailing list.

Abstract

This specification defines an API to manage usage and availability of local storage resources, and defines a means by which a user agent (UA) may grant Web applications permission to use more local space, temporarily or persistently, via various different storage APIs.

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 is a **proposal** and may change without any notices. Interested parties should bring discussions to the Web Platform Incubator Community Group.

This document was published by the Web Platform Working Group as a Working Group Note for history purposes. If you wish to make comments regarding this document, please direct them to the Web Platform Incubator Community Group instead. All comments are welcome.

You may find historical discussion in public-webapps@w3.org (subscribe, archives) with `[quota-api]` at the start of your email's subject.

Table of Contents

1. **Introduction**

2. **Conformance**

3. **Terminology**

4. **Storage Types**

4.1 Temporary vs Persistent

4.2 Examples

5. Quota Management API
5.1 StorageType enum
5.2 StorageInfo dictionary
5.2.1 Dictionary StorageInfo Members
5.3 StorageQuota interface
5.3.1 Attributes
5.3.2 Methods
5.3.3 Accessing StorageQuota interface
5.3.3.1 Attributes
5.4 StorageEvent interface
5.4.1 Attributes
5.5 StorageWatcher interface
5.5.1 Attributes
5.5.2 Methods
5.6 Restrictions

6. Storage Pressure Events
6.1 Extensions to the ServiceWorkerGlobalScope interface
6.1.1 Attributes
6.2 BeforeEvictEvent interface
6.2.1 Attributes
6.3 EvictedEvent interface

7. Quota handling in storage APIs

A. Acknowledgements

B. References
B.1 Normative references
B.2 Informative references

1. Introduction

This section is non-normative.

Today we have a variety of storage APIs that can store inherently complex or large data in order to satisfy offline data requirements of Web applications. Examples of these APIs include: Application Cache [OFFLINE-WEBAPPS], FileSystem API [FILE-SYSTEM][NEW-FILE-SYSTEM], Indexed Database [INDEXEDDB] and Web SQL Database [WEB-SQL].

These APIs may require larger local space than the conventional cookie storage or Web Storage [WEBSTORAGE], but they do not provide a means by which a Web application can query and manage how much data is currently stored and how much more can be stored.

This specification defines an API to query and manage usage and availability of a user's local storage. The storage space granted by the API is intended to be shared by different storage APIs, therefore a user and UA only need to manage single upper limit for all storage per logical application unit, e.g. per origin or per browser (which is implementation specific).

2. 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 word MUST is to be interpreted as described in [RFC2119].

User agents 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 [WEBIDL] as this specification uses that specification and terminology.

3. Terminology

The terms ***event handler***, ***event handler event type***, ***fires a simple event*** are defined in [HTML5].

The ***Event***, ***EventTarget***, ***EventListener*** ***interfaces are defined in [DOM4].***

The ***DOMError*** and ***DOMException*** interfaces are defined in [DOM4].

The ***Promise*** interface is currently defined and discussed in the WHATWG DOM specification.

The terms ***service worker*** and ***scope url***, and the types ***ServiceWorkerGlobalScope***, ***ExtendableEvent***, and ***ExtendableEventInit*** are defined in [SERVICE-WORKERS].

4. Storage Types

4.1 Temporary vs Persistent

This section is non-normative.

A Web application can request temporary or persistent local storage space depending on its purpose.

- ***Temporary*** storage is a transient storage that is more easily accessible by any Web applications without upfront quota request or user prompting, but the data stored in the storage may get deleted at any time at the UA's discretion. This storage can be viewed as analogous to the '/tmp' directory of a traditional filesystem.
- ***Persistent*** storage needs upfront quota request which possibly triggers user prompting, but should never get deleted by the UA without the user's explicit instruction.

Temporary type of storage is especially useful if an application wants to cache data locally to improve its performance, but can fall back to fetching or recreating the same data at a data loss.

Conversely, persistent type of storage is useful if an application wants to store critical offline data that is necessary to be functional, or wants to manage local data amount and lifetime on its own policy rather than relying on the UA's default eviction policy for temporary storage.

4.2 Examples

Suppose there is a photo editing application. This application manages user photo data using Indexed Database [INDEXEDDB], stores photo images using Filesystem API [FILE-SYSTEM] [NEW-FILE-SYSTEM] and optionally utilizes Application Cache [OFFLINE-WEBAPPS] to make it work offline.

The application needs to query how much data it can store in the temporary storage to determine its initial cache size.

EXAMPLE 1

```
// Query current usage and availability in Temporary storage:
navigator.storageQuota.queryInfo("temporary").then(
  function(storageInfo) {
    // Continue to initialize local cache using the obtained
    // usage and remaining space (quota - usage) information.
    initializeCache(storageInfo.usage,
                    storageInfo.quota - storageInfo.usage);
  });
```

Similarly, the application needs to request additional persistent storage to support offline mode when it is enabled by the user.

EXAMPLE 2

```
// A function which is to be called when 'offline-mode' is enabled
// by the user.
function onOfflineModeEnabled(amountOfSpaceNeeded) {
  // First check how much we can use in the Persistent storage.
  navigator.storageQuota.queryInfo("persistent").then(
    function (storageInfo) {
      var availableSpace = storageInfo.quota - storageInfo.usage;
      if (availableSpace >= amountOfSpaceNeeded) {
        // We're fine; just continue with the returned storage info.
        return storageInfo;
      }
      return navigator.storageQuota.requestPersistentQuota(
          amountOfSpaceNeeded + storageInfo.usage);
    }
  ).then(
    function (storageInfo) {
      // Prepare for offline mode using the current available
      // storage space.
      prepareForOfflineMode(storageInfo.quota - storageInfo.usage);
    }, function (error) {
      // Handle error.
    }
  );
}
```

5. Quota Management API

5.1 `StorageType` enum

WebIDL

```
enum StorageType {
    "temporary",
    "persistent"
};
```

Enumeration description	
`temporary`	Indicates temporary storage type.
`persistent`	Indicates persistent storage type.

5.2 `StorageInfo` dictionary

WebIDL

```
dictionary StorageInfo {
             unsigned long long usage;
             unsigned long long quota;
};
```

5.2.1 Dictionary `StorageInfo` Members

`quota` of type unsigned long long

The current upper limit of the storage space that can be used by the application for a given storage type. This includes the storage area that is already used by the application, so `storageInfo.usage` needs to be subtracted from `storageInfo.quota` to get the remaining available storage space.

For temporary storage this value may reflect the actual storage space available on the user's local device and may change from time to time. For persistent storage this value must return the consistent quota size that is granted to the application by `requestPersistentQuota`. If the application does not have the associated persistent quota yet the UA may return a UA-specific default quota value (which could be 0).

`usage` of type unsigned long long

The total amount of data (in bytes) stored by the application for a given storage type. Depending on how the UA calculates data usage the returned value may differ from the exact real-time usage of the user's physical local storage.

5.3 `StorageQuota` interface

The **StorageQuota** interface provides means to query and request storage usage and quota information. The API provided by the interface is asynchronous since querying or allocating space in a user's local storage may require blocking I/O operations, e.g. examining the local disk status or making changes in a local database.

WebIDL

```
[NoInterfaceObject]
interface StorageQuota {
    [SameObject]
    readonly        attribute frozen array supportedTypes;
    Promise queryInfo (StorageType type);
    Promise requestPersistentQuota ([Clamp] unsigned long long
newQuota);
};
```

5.3.1 Attributes

`supportedTypes` of type frozen array<StorageType>, readonly
List of all storage types supported by the UA.

5.3.2 Methods

`queryInfo`
This method queries the storage info of the given storage `type`. This returns `StorageInfo` that has the current data usage and available quota information for the application. When `queryInfo` method is called, the UA must run the following steps:

1. Let *storageType* be the first argument.
2. Let *p* be a new promise.
3. Run the following substeps asynchronously:
    1. If the *storageType* is not supported by the user agent (i.e. not in the `supportedTypes` array), reject *p* with a "NotSupportedError" exception and abort these steps.
    2. Let *usage* be the total amount of data stored by the application in the storage of *storageType*.

3. Let *quota* be the upper limit of the usage that can be stored by the application for *storageType*.
4. Resolve *p* with a new `StorageInfo` object, created with the *usage* and *quota* values.

4. Return *p*.

Parameter	Type	Nullable	Optional	Description
type	`StorageType`	✘	✘	

Return type: `Promise`

requestPersistentQuota

Requests a new quota in persistent storage for the requesting application. It is not guaranteed that the requested amount of space is granted just by calling this API, and the UA may return a new `StorageInfo` object (without rejecting the request) with a smaller quota value than requested. Calling this API may trigger user prompting to request explicit user permission to proceed.

When `requestPersistentQuota` method is called, the UA must run the following steps:

1. Let *newQuota* be the first argument.
2. Let *p* be a new promise.
3. Run the following substeps asynchronously:
    1. Let *usage* be the total amount of data stored by the application in persistent storage.
    2. Let *quota* be the current quota size that is already granted to the application for persistent storage. If this application does not have associated persistent quota yet, initialize *quota* with a UA-specific default quota value (which could be 0).
        1. If *quota* is equal to or greater than *newQuota*, let *grantedQuota* be the same value as the current *quota*.
        2. Else:
            1. Check how much amount of storage the UA can allocate for the application. This step may optionally prompt the user in a UA-

specific manner for permission to allow the application to store more data.

    2. Let *grantedQuota* be the new quota size associated to the application. The *grantedQuota* may be smaller than the requested *newQuota*, but is guaranteed to be equal to or greater than the old *quota*.

3. Resolve *p* with a new `StorageInfo` object, created with *usage* and *grantedQuota*.

4. Return *p*.

Parameter	Type	Nullable	Optional	Description
newQuota	`unsigned long long`	✘	✘	

Return type: `Promise`

5.3.3 Accessing `StorageQuota` interface

WebIDL

```
partial interface Navigator {
    readonly        attribute StorageQuota storageQuota;
};
```

5.3.3.1 Attributes

`storageQuota` of type *StorageQuota*, readonly
Returns the `StorageQuota` interface.

5.4 `StorageEvent` interface

An event object implementing this interface is passed to `onstoragechange` event handler when storage information is updated.

WebIDL

```
[Constructor(DOMString type, optional StorageEventInit
eventInitDict)]
interface StorageEvent : Event {
    readonly        attribute unsigned long long usage;
    readonly        attribute unsigned long long quota;
};
```

5.4.1 Attributes

quota of type unsigned long long, readonly
: The current upper limit of the storage space that can be used by the application for a given storage type.

usage of type unsigned long long, readonly
: The total amount of data (in bytes) stored by the application for a given storage type.

Events are constructed as defined in constructing events in [DOM4].

WebIDL

```
dictionary StorageEventInit : EventInit {
                unsigned long long usage = 0;
                unsigned long long quota = 0;
};
```

Dictionary `StorageEventInit` Members

quota of type unsigned long long, defaulting to `0`

usage of type unsigned long long, defaulting to `0`

5.5 `StorageWatcher` interface

`StorageWatcher` interface is to watch real time storage changes. This fires `storagechange` event every time a storage status change is detected by the UA, or about every `rate` millisecond(s), whichever is least frequent.

If `rate` is not given in the constructor, the UA fires `storagechange` event whenever it detects the usage or quota changes in the backend, but no more frequent than 50ms.

Regardless of the `rate` value, the UA must fire one `storagechange` event with the current storage status immediately (but asynchronously) after a watcher is constructed, unless `close()` is called before the first `storagechange` event is dispatched.

WebIDL

```
[Constructor(StorageType type, optional unsigned long rate)]
interface StorageWatcher : EventTarget {
    void close ();
    readonly        attribute StorageType   type;
    readonly        attribute unsigned long rate;
                    attribute EventListener onstoragechange;
};
```

5.5.1 Attributes

`onstoragechange` of type EventListener
: The event handler for the `storagechange` event.

`rate` of type unsigned long, readonly
: Returns the `rate` value which this watcher is constructed with.

`type` of type *StorageType*, readonly
: Returns the storage type `type` which this watcher is constructed with and is monitoring changes on.

5.5.2 Methods

`close`
: Explicitly closes the watcher. Once close() is called the UA never fires events on this instance.
No parameters.
Return type: `void`

5.6 Restrictions

The space queried and granted by **`StorageQuota`** have the following properties:

- The space granted for persistent type should not be deleted by the UA, other than in a response to a removal API call, without explicit authorization from the user.
- The space granted for temporary type may be deleted by the UA at its discretion, without application or user intervention due to local resource shortage or for other reasons.
- When the UA deletes the data in temporary storage, the UA should avoid partial or incomplete deletion which could leave the application in an unrecoverable inconsistent state.

6. Storage Pressure Events

6.1 Extensions to the `ServiceWorkerGlobalScope` interface

The Service Worker specification [SERVICE-WORKERS] defines a `ServiceWorkerGlobalScope` interface, which this specification extends.

WebIDL

```
partial interface ServiceWorkerGlobalScope {
                        attribute EventHandler onbeforeevicted;
                        attribute EventHandler onevicted;
};
```

6.1.1 Attributes

`onbeforeevicted` of type EventHandler
: The `onbeforeevicted` attribute is an event handler whose corresponding event handler type is `beforeevicted`.

`onevicted` of type EventHandler
: The `onevicted` attribute is an event handler whose corresponding event handler type is `evicted`.

This event is going to be dispatched when the Quota Management API detects that there is a storage pressure. If the service worker that received this event did not give back enough space the UA may evict its content entirely.

6.2 `BeforeEvictEvent` interface

WebIDL

```
[Constructor(DOMString type, optional BeforeEvictEventInit
eventInitDict), Exposed=ServiceWorker]
interface BeforeEvictEvent : ExtendableEvent {
    readonly        attribute unsigned long long requestedBytes;
};
```

6.2.1 Attributes

`requestedBytes` of type unsigned long long, readonly
    The number of bytes that is requested to free up.

6.3 `EvictedEvent` interface

WebIDL

```
[Constructor(DOMString type, optional EvictedEventInit
eventInitDict), Exposed=ServiceWorker]
interface EvictedEvent : ExtendableEvent {
};
```

This event is only going to be dispatched if there are foreground pages that are otherwise keep the service worker's script and registrations alive. It is thrown to let the service worker communicate to clients that the app has been removed and is unlikely to continue to work. The clients can show appropriate UI in response.

7. Quota handling in storage APIs

This section is non-normative.

Storage APIs except for Web Storage, i.e. Application Cache, File System API, Indexed Database and Web SQL Database, should respect the quota management API and must have following properties:

- Any write requests which will exceed the granted quota limit should fail with QuotaExceededError `DOMError` or `DOMException` (including when the granted quota is zero, i.e. the UA refuses to grant any quota or the storage is disabled for the site). As an exception, if the write is being made in the background where it cannot throw exception or return an error, the API may fail silently. For example, Application Cache may silently discard or fail to cache data when it is hitting quota limit.
- For FileSystem API [FILE-SYSTEM][NEW-FILE-SYSTEM] the UA must store data for **temporary filesystem** in temporary storage, and data for **persistent filesystem** in persistent storage.
- For Application Cache [OFFLINE-WEBAPPS] the UA must store its cache data in temporary storage.
- Other storage APIs may store data in either temporary or persistent storage. When an API stores data in persistent storage by default, storing data or opening the storage (e.g. opening a database) may trigger user prompting by the UA to request for user permission.

NOTE

Indexed Database [INDEXEDDB] is expected to have temporary and persistent storage types in its next version, and when that happens the UA should store data for temporary database in temporary storage and for persistent database in persistent storage, respectively.

A. Acknowledgements

Many thanks to Robin Berjon for making our lives so much easier with his cool tool.

B. References

B.1 Normative references

[RFC2119]
S. Bradner. IETF. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[SERVICE-WORKERS]
Alex Russell; Jungkee Song; Jake Archibald. W3C. *Service Workers*. 25 June 2015. W3C Working Draft. URL: http://www.w3.org/TR/service-workers/

B.2 Informative references

[DOM4]
Anne van Kesteren; Aryeh Gregor; Ms2ger; Alex Russell; Robin Berjon. W3C. *W3C DOM4*. 19 November 2015. W3C Recommendation. URL: http://www.w3.org/TR/dom/

[FILE-SYSTEM]
Eric Uhrhane. W3C. *File API: Directories and System*. 24 April 2014. W3C Note. URL: http://www.w3.org/TR/file-system-api/

[HTML5]
Ian Hickson; Robin Berjon; Steve Faulkner; Travis Leithead; Erika Doyle Navara; Theresa O'Connor; Silvia Pfeiffer. W3C. *HTML5*. 28 October 2014. W3C Recommendation. URL: http://www.w3.org/TR/html5/

[INDEXEDDB]
Nikunj Mehta; Jonas Sicking; Eliot Graff; Andrei Popescu; Jeremy Orlow; Joshua Bell. W3C. *Indexed Database API*. 8 January 2015. W3C Recommendation. URL: http://www.w3.org/TR/IndexedDB/

[NEW-FILE-SYSTEM]
Arun Ranganathan. W3C. *FileSystem API*. W3C Editor's Draft. URL: http://w3c.github.io/filesystem-api/

[OFFLINE-WEBAPPS]
Anne van Kesteren; Ian Hickson. W3C. *Offline Web Applications*. 30 May 2008. W3C Note. URL: http://www.w3.org/TR/offline-webapps/

[WEB-SQL]
Ian Hickson. W3C. *Web SQL Database*. 18 November 2010. W3C Note. URL: http://www.w3.org/TR/webdatabase/

[WEBIDL]

Cameron McCormack; Boris Zbarsky. W3C. *WebIDL Level 1*. 8 March 2016. W3C Candidate Recommendation. URL: http://www.w3.org/TR/WebIDL-1/

[WEBSTORAGE]

Ian Hickson. W3C. *Web Storage (Second Edition)*. 19 April 2016. W3C Recommendation. URL: http://www.w3.org/TR/webstorage/

↑