Storage

Living Standard — Last Updated 26 February 2021

Participate:

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

Commits:

GitHub whatwg/storage/commits
Snapshot as of this commit
@storagestandard

Tests:

web-platform-tests storage/ (ongoing work)

Translations (non-normative):

日本語

Abstract

The Storage Standard defines an API for persistent storage and quota estimates, as well as the platform storage architecture.

Table of Contents

§ 1. Introduction

Over the years the web has grown various APIs that can be used for storage, e.g., IndexedDB, `localStorage`, and `showNotification()`. The Storage Standard consolidates these APIs by defining:

- A bucket, the primitive these APIs store their data in
- A way of making that bucket persistent
- A way of getting usage and quota estimates for an origin

Traditionally, as the user runs out of storage space on their device, the data stored with these APIs gets lost without the user being able to intervene. However, persistent buckets cannot be cleared without consent by the user. This thus brings data guarantees users have enjoyed on native platforms to the web.

¶ Example

A simple way to make storage persistent is through invoking the `persist()` method. It simultaneously requests the end user for permission and changes the storage to be persistent once granted:

```
navigator.storage.persist().then(persisted => {
  if (persisted) {
    /* … */
  }
});
```

To not show user-agent-driven dialogs to the end user unannounced slightly more involved code can be written:

```
Promise.all([
  navigator.storage.persisted(),
  navigator.permissions.query({name: "persistent-storage"})
]).then(([persisted, permission]) => {
  if (!persisted && permission.state == "granted") {
    navigator.storage.persist().then( /* … */ );
  } else if (!persisted && permission.state == "prompt") {
    showPersistentStorageExplanation();
  }
});
```

The `estimate()` method can be used to determine whether there is enough space left to store content for an application:

```
function retrieveNextChunk(nextChunkInfo) {
  return navigator.storage.estimate().then(info => {
    if (info.quota - info.usage > nextChunkInfo.size) {
      return fetch(nextChunkInfo.url);
    } else {
      throw new Error("insufficient space to store next
chunk");
    }
  }).then( /* … */ );
}
```

§ 2. Terminology

This specification depends on the Infra Standard. [INFRA]

This specification uses terminology from the HTML, IDL, and Permissions Standards. [HTML] [WEBIDL] [PERMISSIONS]

§ 3. Lay of the land

A user agent has various kinds of semi-persistent state:

Credentials

End-user credentials, such as username and passwords submitted through HTML forms

Permissions

Permissions for various features, such as geolocation

Network

HTTP cache, cookies, authentication entries, TLS client certificates

Storage

Indexed DB, Cache API, service worker registrations, `localStorage`, `sessionStorage`, application caches, notifications, etc.

This standard primarily concerns itself with storage.

§ 4. Model

Standards defining local or session storage APIs will define a storage endpoint and register it by changing this standard. They will invoke either the obtain a local storage bottle map or the obtain a session storage bottle map algorithm, which will give them:

- Failure, which might mean the API has to throw or otherwise indicate there is no storage available for that environment settings object.
- A storage proxy map that operates analogously to a map, which can be used to store data in a manner that suits the API. This standard takes care of isolating that data from other APIs, storage keys, and storage types.

Note

If you are defining a standard for such an API, consider filing an issue against this standard for assistance and review.

To isolate this data this standard defines a storage shed which segments storage shelves by a storage key. A storage shelf in turn consists of a storage bucket and will likely consist of multiple storage buckets in the future to allow for different storage policies. And lastly, a storage bucket consists of storage bottles, one for each storage endpoint.

§ 4.1. Storage endpoints

A **storage endpoint** is a local or session storage API that uses the infrastructure defined by this standard, most notably storage bottles, to keep track of its storage needs.

A storage endpoint has an **identifier**, which is a storage identifier.

A storage endpoint also has **types**, which is a set of storage types.

A storage endpoint also has a **quota**, which is null or a number representing a recommended quota (in bytes) for each storage bottle corresponding to this storage endpoint.

A **storage identifier** is an ASCII string.

A **storage type** is "`local`" or "`session`".

The **registered storage endpoints** are a set of storage endpoints defined by the following table:

Identifier	Type	Quota
"caches"	« "local" »	null
"indexedDB"	« "local" »	null
"localStorage"	« "local" »	5×2^{20} (i.e., 5 mebibytes)
"serviceWorkerRegistrations"	« "local" »	null
"sessionStorage"	« "session" »	5×2^{20} (i.e., 5 mebibytes)

Note

As mentioned, standards can use these storage identifiers with obtain a local storage bottle map and obtain a session storage bottle map. It is anticipated that some APIs will be applicable to both storage types going forward.

§ 4.2. Storage keys

A **storage key** is an origin. [HTML]

This is expected to change; see Client-Side Storage Partitioning.

To **obtain a storage key**, given an environment settings object *environment*, run these steps:

1. Let *key* be *environment*'s origin.
2. If *key* is an opaque origin, then return failure.
3. If the user has disabled storage, then return failure.
4. Return *key*.

§ 4.3. Storage sheds

A **storage shed** is a map of storage keys to storage shelves. It is initially empty.

A user agent holds a **storage shed**, which is a storage shed. A user agent's storage shed holds all **local storage** data.

A browsing session holds a **storage shed**, which is a storage shed. A browsing session's storage shed holds all **session storage** data.

To **legacy-clone a browsing session storage shed**, given a browsing session *A* and a browsing session *B*, run these steps:

1. For each *key* → *shelf* of *A*'s storage shed:
    1. Let *newShelf* be the result of running create a storage shelf with `"session"`.
    2. Set *newShelf*'s bucket map["`default`"]'s bottle map["`sessionStorage`"]'s map to a clone of *shelf*'s bucket map["`default`"]'s bottle map["`sessionStorage`"]'s map.
    3. Set *B*'s storage shed[*key*] to *newShelf*.

Note

This is considered legacy as the benefits, if any, do not outweigh the implementation complexity. And therefore it will not be expanded or used outside

of HTML. *[HTML]*

§ 4.4. Storage shelves

A **storage shelf** exists for each storage key within a storage shed. It holds a **bucket map**, which is a map of strings to storage buckets.

Note

For now `"default"` *is the only key that exists in a bucket map. See issue #2. It is given a value when a storage shelf is obtained for the first time.*

To **obtain a storage shelf**, given a storage shed *shed*, an environment settings object *environment*, and a storage type *type*, run these steps:

1. Let *key* be the result of running obtain a storage key with *environment*.
2. If *key* is failure, then return failure.
3. If *shed*[*key*] does not exist, then set *shed*[*key*] to the result of running create a storage shelf with *type*.
4. Return *shed*[*key*].

To **obtain a local storage shelf**, given an environment settings object *environment*, return the result of running obtain a storage shelf with the user agent's storage shed, *environment*, and `"local"`.

To **create a storage shelf**, given a storage type *type*, run these steps:

1. Let *shelf* be a new storage shelf.
2. Set *shelf*'s bucket map[`"default"`] to the result of running create a storage bucket with *type*.
3. Return *shelf*.

§ 4.5. Storage buckets

A **storage bucket** is a place for storage endpoints to store data.

A storage bucket has a **bottle map** of storage identifiers to storage bottles.

A **local storage bucket** is a storage bucket for local storage APIs.

A local storage bucket has a **mode**, which is `"best-effort"` or `"persistent"`. It is initially `"best-effort"`.

A **session storage bucket** is a storage bucket for session storage APIs.

To **create a storage bucket**, given a storage type *type*, run these steps:

1. Let *bucket* be null.
2. If *type* is "`local`", then set *bucket* to a new local storage bucket.
3. Otherwise:
    1. Assert: *type* is "`session`".
    2. Set *bucket* to a new session storage bucket.
4. For each *endpoint* of registered storage endpoints whose types contain *type*, set *bucket*'s bottle map[*endpoint*'s identifier] to a new storage bottle whose quota is *endpoint*'s quota.
5. Return *bucket*.

§ 4.6. Storage bottles

A **storage bottle** is a part of a storage bucket carved out for a single storage endpoint. A storage bottle has a **map**, which is initially an empty map. A storage bottle also has a **proxy map reference set**, which is initially an empty set. A storage bottle also has a **quota**, which is null or a number representing a conservative estimate of the total amount of bytes it can hold. Null indicates the lack of a limit.

Note *It is still bound by the storage quota of its encompassing storage shelf.*

A storage bottle's map is where the actual data meant to be stored lives. User agents are expected to store this data, and make it available across agent and even agent cluster boundaries, in an implementation-defined manner, so that this standard and standards using this standard can access the contents.

To **obtain a storage bottle map**, given a storage type *type*, environment settings object *environment*, and storage identifier *identifier*, run these steps:

1. Let *shed* be null.
2. If *type* is "`local`", then set *shed* to the user agent's storage shed.
3. Otherwise:
    1. Assert: *type* is "`session`".
    2. Set *shed* to *environment*'s browsing session's storage shed.
4. Let *shelf* be the result of running obtain a storage shelf, with *shed*, *environment*, and *type*.
5. If *shelf* is failure, then return failure.
6. Let *bucket* be *shelf*'s bucket map["`default`"].
7. Let *bottle* be *bucket*'s bottle map[*identifier*].
8. Let *proxyMap* be a new storage proxy map whose backing map is *bottle*'s map.
9. Append *proxyMap* to *bottle*'s proxy map reference set.

10. Return *proxyMap*.

To **obtain a local storage bottle map**, given an environment settings object *environment* and storage identifier *identifier*, return the result of running obtain a storage bottle map with "`local`", *environment*, and *identifier*.

To **obtain a session storage bottle map**, given an environment settings object *environment* and storage identifier *identifier*, return the result of running obtain a storage bottle map with "`session`", *environment*, and *identifier*.

§ 4.7. Storage proxy maps

A **storage proxy map** is equivalent to a map, except that all operations are instead performed on its **backing map**.

This allows for the backing map to be replaced. This is needed for issue #4 and potentially the Storage Access API.

§ 5. Persistence permission

A local storage bucket can only have its mode change to "`persistent`" if the user (or user agent on behalf of the user) has granted permission to use the "`persistent-storage`" feature.

Note

When granted to an origin, the persistence permission can be used to protect storage from the user agent's clearing policies. The user agent cannot clear storage marked as persistent without involvement from the origin or user. This makes it particularly useful for resources the user needs to have available while offline or resources the user creates locally.

The "`persistent-storage`" powerful feature's permission-related flags, algorithms, and types are defaulted, except for:

permission state

"`persistent-storage`"'s permission state must have the same value for all environment settings objects with a given origin.

permission revocation algorithm

1. If "`persistent-storage`"'s permission state is "`granted`", then return.
2. Let *shelf* be the result of running obtain a local storage shelf with current settings object.
3. Set *shelf*'s bucket map["`default`"]'s mode to "`best-effort`".

§ 6. Usage and quota

The **storage usage** of a storage shelf is an implementation-defined rough estimate of the amount of bytes used by it.

Note

This cannot be an exact amount as user agents might, and are encouraged to, use deduplication, compression, and other techniques that obscure exactly how much bytes a storage shelf uses.

The **storage quota** of a storage shelf is an implementation-defined conservative estimate of the total amount of bytes it can hold. This amount should be less than the total storage space on the device. It must not be a function of the available storage space on the device.

Note

User agents are strongly encouraged to consider navigation frequency, recency of visits, bookmarking, and permission for "`persistent-storage`" when determining quotas.

Directly or indirectly revealing available storage space can lead to fingerprinting and leaking information outside the scope of the origin involved.

§ 7. Management

Whenever a storage bucket is cleared by the user agent, it must be cleared in its entirety. User agents should avoid clearing storage buckets while script that is able to access them is running, unless instructed otherwise by the user.

If removal of storage buckets leaves the encompassing storage shelf's bucket map empty, then remove that storage shelf and corresponding storage key from the encompassing storage shed.

§ 7.1. Storage pressure

A user agent that comes under storage pressure should clear network state and local storage buckets whose mode is "`best-effort`", ideally prioritizing removal in a manner that least impacts the user.

If a user agent continues to be under storage pressure, then the user agent should inform the user and offer a way to clear the remaining local storage buckets, i.e., those whose mode is "`persistent`".

Session storage buckets must be cleared as browsing sessions are closed.

Note

If the user agent allows for revival of browsing sessions, e.g., through reopening browsing sessions or continued use of them after restarting the user agent, then clearing necessarily involves a more complex set of heuristics.

§ 7.2. User interface guidelines

User agents should offer users the ability to clear network state and storage for individual websites. User agents should not distinguish between network state and storage in their user interface. This ensures network state cannot be used to revive storage and reduces the number of concepts users need to be mindful of.

Credentials should be separated as they contain data the user might not be able to revive, such as an autogenerated password. Permissions are best separated too to avoid inconveniencing the user.

§ 8. API

```
[SecureContext]
interface mixin NavigatorStorage {
  [SameObject] readonly attribute StorageManager storage;
};
Navigator includes NavigatorStorage;
WorkerNavigator includes NavigatorStorage;
```

Each environment settings object has an associated StorageManager object. [HTML]

The **storage** getter steps are to return this's relevant settings object's StorageManager object.

```
[SecureContext,
 Exposed=(Window,Worker)]
interface StorageManager {
  Promise<boolean> persisted();
  [Exposed=Window] Promise<boolean> persist();

  Promise<StorageEstimate> estimate();
};

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

The **persisted()** method steps are:

1. Let *promise* be a new promise.
2. Let *shelf* be the result of running obtain a local storage shelf with this's relevant settings object.
3. If *shelf* is failure, then reject *promise* with a TypeError.
4. Otherwise, run these steps in parallel:
    1. Let *persisted* be true if *shelf*'s bucket map["default"]'s mode is "persistent"; otherwise false.

       Note

       It will be false when there's an internal error.

    2. Queue a task to resolve *promise* with *persisted*.
5. Return *promise*.

The **persist()** method steps are:

1. Let *promise* be a new promise.
2. Let *shelf* be the result of running obtain a local storage shelf with this's relevant settings object.
3. If *shelf* is failure, then reject *promise* with a TypeError.

4. Otherwise, run these steps in parallel:

    1. Let *permission* be the result of requesting permission to use "`persistent-storage`".

        Note

        User agents are encouraged to not let the user answer this question twice for the same origin around the same time and this algorithm is not equipped to handle such a scenario.

    2. Let *bucket* be *shelf*'s bucket map["`default`"].

    3. Let *persisted* be true if *bucket*'s mode is "`persistent`"; otherwise false.

        Note

        It will be false when there's an internal error.

    4. If *persisted* is false and *permission* is "`granted`", then:

        1. Set *bucket*'s mode to "`persistent`".

        2. If there was no internal error, then set *persisted* to true.

    5. Queue a task to resolve *promise* with *persisted*.

5. Return *promise*.

The **`estimate()`** method steps are:

1. Let *promise* be a new promise.

2. Let *shelf* be the result of running obtain a local storage shelf with this's relevant settings object.

3. If *shelf* is failure, then reject *promise* with a `TypeError`.

4. Otherwise, run these steps in parallel:

    1. Let *usage* be storage usage for *shelf*.

    2. Let *quota* be storage quota for *shelf*.

    3. Let *dictionary* be a new `StorageEstimate` dictionary whose `usage` member is *usage* and `quota` member is *quota*.

    4. If there was an internal error while obtaining *usage* and *quota*, then queue a task to reject *promise* with a `TypeError`.

        Note

        Internal errors are supposed to be extremely rare and indicate some kind of low-level platform or hardware fault. However, at the scale of the web with the diversity of implementation and platforms, the unexpected does occur.

    5. Otherwise, queue a task to resolve *promise* with *dictionary*.

5. Return *promise*.

§ Acknowledgments

With that, many thanks to Adrian Bateman, Aislinn Grigas, Alex Russell, Ali Alabbas, Andrew Sutherland, Ben Kelly, Ben Turner, Dale Harvey, David Grogan, Domenic Denicola, fantasai, Jake Archibald, Jeffrey Yasskin, Jesse Mykolyn, Jinho Bang, Jonas Sicking, Joshua Bell, Kenji Baheux, Kinuko Yasuda, Luke Wagner, Michael Nordman, Mounir Lamouri, Shachar Zohar, 黃強 (Shawn Huang), 簡冠庭 (Timothy Guan-tin Chien), and Victor Costan for being awesome!

This standard is written by Anne van Kesteren (Mozilla, annevk@annevk.nl).

§ Intellectual property rights

This is the Living Standard. Those interested in the patent-review version should view the Living Standard Review Draft.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

- [ECMASCRIPT] defines the following terms:
  - agent
  - agent cluster
- [HTML] defines the following terms:
  - Navigator
  - WorkerNavigator
  - browsing session (for environment settings object)
  - current settings object
  - environment settings object
  - in parallel
  - opaque origin
  - origin (for environment settings object)
  - queue a task
  - relevant settings object
- [INFRA] defines the following terms:
  - append
  - ascii string
  - clone
  - contain
  - exist
  - for each (for map)
  - implementation-defined
  - is empty
  - key
  - map
  - remove
  - set
  - string
  - user agent
  - value
- [PERMISSIONS] defines the following terms:
  - "granted"
  - "persistent-storage"
  - permission revocation algorithm
  - permission state
  - powerful feature
  - requesting permission to use
- [WEBIDL] defines the following terms:
  - Exposed
  - Promise
  - SameObject
  - SecureContext
  - TypeError
  - boolean
  - this
  - unsigned long long

§ References

§ Normative References

[ECMASCRIPT]

ECMAScript Language Specification. URL: https://tc39.es/ecma262/

[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/

[PERMISSIONS]

Mounir Lamouri; Marcos Caceres; Jeffrey Yasskin. Permissions. URL: https://w3c.github.io/permissions/

[WEBIDL]

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

§ IDL Index

```
[SecureContext]
interface mixin NavigatorStorage {
  [SameObject] readonly attribute StorageManager storage;
};
Navigator includes NavigatorStorage;
WorkerNavigator includes NavigatorStorage;

[SecureContext,
 Exposed=(Window,Worker)]
interface StorageManager {
  Promise<boolean> persisted();
  [Exposed=Window] Promise<boolean> persist();

  Promise<StorageEstimate> estimate();
};

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