Service Workers 1

W3C Working Draft, 13 August 2019

This version:
https://www.w3.org/TR/2019/WD-service-workers-1-20190813/

Latest published version:
https://www.w3.org/TR/service-workers-1/

Editor's Draft:
https://w3c.github.io/ServiceWorker/v1/

Previous Versions:
https://www.w3.org/TR/2017/WD-service-workers-1-20171102/

Issue Tracking:
GitHub

Editors:
Alex Russell (Google)
Jungkee Song (Microsoft, represented Samsung until April 2018)
Jake Archibald (Google)
Marijn Kruisselbrink (Google)

Tests:
web-platform-tests service-workers/ (ongoing work)

▼ V1 Branch

This spec is a subset of the nightly version that is advancing toward a W3C

Recommendation. For implementers and developers who seek all the latest features, Service Workers Nightly is a right document that is constantly reflecting new requirements.

Abstract

This specification describes a method that enables applications to take advantage of persistent background processing, including hooks to enable bootstrapping of web applications while offline.

The core of this system is an event-driven Web Worker, which responds to events dispatched from documents and other sources. A system for managing installation, versions, and upgrades is provided.

The service worker is a generic entry point for event-driven background processing in the Web Platform that is extensible

by other specifications.

Status of this document

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

This document was published by the Service Workers Working Group as a Working Draft. This document is intended to become a W3C Recommendation.

Feedback and comments on this specification are welcome, please send them to public-webapps@w3.org (subscribe, archives) with `[service-workers]` at the start of your email's subject.

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or ob-

soleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

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

This document is governed by the 1 March 2019 W3C Process Document.

Table of Contents

§ 1. Motivations

This section is non-normative.

Web Applications traditionally assume that the network is reachable. This assumption pervades the platform. HTML documents are loaded over HTTP and traditionally fetch all of their sub-resources via subsequent HTTP requests. This places web content at a disadvantage versus other technology stacks.

The service worker is designed first to redress this balance by providing a Web Worker context, which can be started by a runtime when navigations are about to occur. This event-driven worker is registered against an origin and a path (or pattern), meaning it can be consulted when navigations occur to that location. Events that correspond to network requests are dispatched to the worker and the responses generated by the worker may override default network stack behavior. This puts the service worker, conceptually, between the network and a document renderer, allowing the ser-

vice worker to provide content for documents, even while offline.

Web developers familiar with previous attempts to solve the offline problem have reported a deficit of flexibility in those solutions. As a result, the service worker is highly procedural, providing a maximum of flexibility at the price of additional complexity for developers. Part of this complexity arises from the need to keep service workers responsive in the face of a single-threaded execution model. As a result, APIs exposed by service workers are almost entirely asynchronous, a pattern familiar in other JavaScript contexts but accentuated here by the need to avoid blocking document and resource loading.

Developers using the HTML5 Application Cache have also reported that several attributes of the design contribute to unrecoverable errors. A key design principle of the service worker is

that errors should *always* be recoverable. Many details of the update process of service workers are designed to avoid these hazards.

Service workers are started and kept alive by their relationship to events, not documents. This design borrows heavily from developer and vendor experience with Shared Workers and Chrome Background Pages. A key lesson from these systems is the necessity to time-limit the execution of background processing contexts, both to conserve resources and to ensure that background context loss and restart is top-of-mind for developers. As a result, service workers bear more than a passing resemblance to Chrome Event Pages, the successor to Background Pages. Service workers may be started by user agents *without an attached document* and may be killed by the user agent at nearly any time. Conceptually, service workers can be thought of as Shared Workers that can start, pro-

cess events, and die without ever handling messages from documents. Developers are advised to keep in mind that service workers may be started and killed many times a second.

Service workers are generic, event-driven, time-limited script contexts that run at an origin. These properties make them natural endpoints for a range of runtime services that may outlive the context of a particular document, e.g. handling push notifications, background data synchronization, responding to resource requests from other origins, or receiving centralized updates to expensive-to-calculate data (e.g., geolocation or gyroscope).

§ 2. Model

§ 2.1. Service Worker

A ***service worker*** is a type of web worker. A service worker executes in the registering service worker client's origin.

A service worker has an associated ***state***, which is one of

"parsed", "installing", "installed", "activating", "activated", and "redundant". It is initially "parsed".

A service worker has an associated ***script url*** (a URL).

A service worker has an associated ***type*** which is either "classic" or "module". Unless stated otherwise, it is "classic".

A service worker has an associated ***containing service worker registration*** (a service worker registration), which contains itself.

A service worker has an associated ***global object*** (a ServiceWorkerGlobalScope object or null).

A service worker has an associated ***script resource*** (a script), which represents its own script resource. It is initially set to null.

A script resource has an associated ***has ever been evaluated flag***. It is initially unset.

A script resource has an associated ***HTTPS state*** (an HTTPS state value). It is initially "none".

A script resource has an associated ***referrer policy*** (a referrer policy). It is initially the empty string.

A service worker has an associated ***script resource map*** which is an ordered map where the keys are URLs and the values are responses.

A service worker has an associated ***skip waiting flag***. Unless stated otherwise it is unset.

A service worker has an associated ***classic scripts imported flag***. It is initially unset.

A service worker has an associated ***set of event types to handle*** (a set) whose item is an event listener's event type. It is initially an empty set.

A service worker has an associated ***set of extended events*** (a set) whose item is an `ExtendableEvent`. It is initially an empty set.

§ 2.1.1. Lifetime

The lifetime of a service worker is tied to the execution lifetime of events and not references held by service worker clients to

the `ServiceWorker` object.

A user agent *may* terminate service workers at any time it:

- Has no event to handle.
- Detects abnormal operation: such as infinite loops and tasks exceeding imposed time limits (if any) while handling the events.

A service worker has an associated ***start status*** which can be null or a Completion. It is initially null.

A service worker is said to be ***running*** if its event loop is running.

§ 2.1.2. Events

The Service Workers specification defines ***service worker events*** (each of which is an event) that include (see the list):

- ***Lifecycle events***: `install` and `activate`.
- ***Functional events***: `fetch` and the events defined by other specifications that extend the Service Workers specification. (See the list.)
- `message` and `messageerror`.

§ 2.2. Service Worker Registration

A ***service worker registration*** is a tuple of a scope url and a set of service workers, an installing worker, a waiting worker, and an active worker. A user agent *may* enable many service worker registrations at a single origin so long as the scope url of the service worker registration differs. A service worker registration of an identical scope url when one already exists in the user agent causes the existing service worker registration to be replaced.

A service worker registration has an associated ***scope url*** (a URL).

A service worker registration has an associated ***installing worker*** (a service worker or null) whose state is "`installing`". It is initially set to null.

A service worker registration has an associated ***waiting worker*** (a service worker or null) whose state is "`installed`". It is initially set to null.

A service worker registration has an associated ***active worker*** (a service worker or null) whose state is either "`activating`" or "`activated`". It is initially set to null.

A service worker registration has an associated ***last update check time***. It is initially set to null.

A service worker registration is said to be ***stale*** if the registration's last update check time is non-null and the time difference in seconds calculated by the current time minus the registration's last update check time is greater than 86400.

A service worker registration has an associated ***update via cache mode***, which is "`imports`", "`all`", or "`none`". It is initially set to "`imports`".

A service worker registration has one or more ***task queues*** that back up the tasks from its active worker's event loop's corresponding task queues. (The target task sources for this back up operation are the handle fetch

task source and the handle functional event task source.) The user agent dumps the active worker's tasks to the service worker registration's task queues when the active worker is terminated and re-queues those tasks to the active worker's event loop's corresponding task queues when the active worker spins off. Unlike the task queues owned by event loops, the service worker registration's task queues are not processed by any event loops in and of itself.

A service worker registration is said to be ***unregistered*** if scope to registration map[this service worker registration's scope url] is not this service worker registration.

§ 2.2.1. Lifetime

A user agent *must* persistently keep a list of registered service worker registrations unless otherwise they are explicitly unregistered. A user agent has a scope to registration map that stores the entries of the tuple of service worker registra-

tion's scope url, serialized, and the corresponding service worker registration. The lifetime of service worker registrations is beyond that of the `ServiceWorkerRegistration` objects which represent them within the lifetime of their corresponding service worker clients.

2.3. Service Worker Client

A ***service worker client*** is an environment.

A service worker client has an associated ***discarded flag***. It is initially unset.

Each service worker client has the following environment discarding steps:

1. Set *client*'s discarded flag.

> Note: Implementations can discard clients whose discarded flag is set.

A service worker client has an algorithm defined as the ***origin*** that returns the service worker client's origin if the service worker client is an environment settings object, and the service worker client's creation URL's origin otherwise.

A ***window client*** is a service worker client whose global object is a `Window` object.

A ***dedicated worker client*** is a service worker client whose global object is a `DedicatedWorkerGlobalScope` object.

A ***shared worker client*** is a service worker client whose global object is a `SharedWorkerGlobalScope` object.

A ***worker client*** is either a dedicated worker client or a shared worker client.

§ 2.4. Control and Use

A service worker client has an active service worker that serves its own loading and its subresources. When a service worker client has a non-null active service worker, it is said to be ***controlled*** by that active service worker. When a service worker client is controlled by a service worker, it is said that the service worker client is ***using*** the service worker's containing service worker registration. A service worker client's active service

worker is determined as explained in the following subsections.

The rest of the section is non-normative.

> Note: The behavior in this section is not fully specified yet and will be specified in HTML Standard. The work is tracked by the issue and the pull request. For any Service Workers changes, we will incorporate them into Service Workers Nightly.

2.4.1. The window client case

A window client is created when a browsing context is created and when it navigates.

When a window client is created in the process of a browsing context creation:

If the browsing context's initial active document's origin is an opaque origin, the window client's active service worker is set to null. Otherwise, it is set to the creator document's service worker client's active service worker.

When a window client is created in the process of the browsing context's navigation:

If the fetch is routed through HTTP fetch, the window client's active service worker is set to the result of the service worker registration matching. Otherwise, if the created document's origin is an opaque origin or not the same as its creator document's origin, the window client's active service worker is set to null. Otherwise, it is set to the creator document's service worker client's active service worker.

> Note: For an initial navigation with replacement enabled, the initial window client that was created when the browsing context was created is reused, but the active service worker is determined by the same behavior as above.

> Note: Sandboxed iframes without the sandboxing directives, `allow-same-origin` and `allow-scripts`, result in having the active service worker value of null as their origin is an opaque origin..

§ 2.4.2. The worker client case

A worker client is created when the user agent runs a worker.

When the worker client is created:

When the fetch is routed through HTTP fetch, the worker client's active service worker is set to the result of the service worker registration matching. Otherwise, if the worker client's origin is an opaque origin, or the request's URL is a blob URL and the worker client's origin is not the same as the origin of the last item in the worker client's global object's owner set, the worker client's active service worker is set to null. Otherwise, it is set to the active service worker of the environment settings object of

the last item in the worker client's global object's owner set.

> Note: Window clients and worker clients with a data: URL result in having the active service worker value of null as their origin is an opaque origin. Window clients and worker clients with a blob URL can inherit the active service worker of their creator document or owner, but if the request's origin is not the same as the origin of their creator document or owner, the active service worker is set to null.

§ 2.5. Task Sources

The following additional task sources are used by service workers.

The *handle fetch task source*
: This task source is used for dispatching `fetch` events to service workers.

The *handle functional event task source*
: This task source is used for features that dispatch other functional events, e.g. `push` events, to service workers.

> Note: A user agent may use a separate task source for each functional event type in order to avoid a head-of-line blocking phenomenon for certain functional events.

§ 2.6. User Agent Shutdown

A user agent *must* maintain the state of its stored service worker registrations across restarts with the following rules:

- An installing worker does not persist but is discarded. If the installing worker was the only service worker for the service worker registration, the service worker registration is discarded.
- A waiting worker promotes to an active worker.

To attain this, the user agent *must* invoke Handle User Agent Shutdown when it terminates.

§ 3. Client Context

EXAMPLE 1

Bootstrapping with a service worker:

```
// scope defaults to the path the script sits in
// "/" in this example
navigator.serviceWorker.register("/serviceworker.js").then(registration => {
  console.log("success!");
  if (registration.installing) {
    registration.installing.postMessage("Howdy from your installing page.");
  }
}, err => {
  console.error("Installing the worker failed!", err);
});
```

§ 3.1. ServiceWorker

```
[SecureContext, Exposed=(Window,Worker)]
interface ServiceWorker : EventTarget {
  readonly attribute USVString scriptURL;
  readonly attribute ServiceWorkerState state;
  void postMessage(any message, optional sequence<object> transfer = []);

  // event
  attribute EventHandler onstatechange;
};
ServiceWorker includes AbstractWorker;

enum ServiceWorkerState {
  "installing",
  "installed",
  "activating",
  "activated",
  "redundant"
};
```

A ServiceWorker object represents a service worker. Each ServiceWorker object is associated with a service worker. Multiple separate objects implementing the ServiceWorker interface across documents and workers can all be associated with the same service worker simultaneously.

A ServiceWorker object has an as-

sociated `ServiceWorkerState` object which is itself associated with service worker's state.

3.1.1. Getting `ServiceWorker` instances

An environment settings object has a ***service worker object map***, a map where the keys are service workers and the values are `ServiceWorker` objects.

To ***get the service worker object*** representing *serviceWorker* (a service worker) in *environment* (an environment settings object), run these steps:

1. Let *objectMap* be *environment*'s service worker object map.
2. If *objectMap*[*serviceWorker*] does not exist, then:
    1. Let *serviceWorkerObj* be a new `ServiceWorker` in *environment*'s Realm, and associate it with *serviceWorker*.
    2. Set *serviceWorkerObj*'s `state` to *serviceWorker*'s state.
    3. Set *objectMap*[*serviceWorker*] to *serviceWorkerObj*.
3. Return *objectMap*[*service-*

Worker].

3.1.2. `scriptURL`

The ***`scriptURL`*** attribute *must* return the service worker's serialized script url.

> EXAMPLE 2
> For example, consider a document created by a navigation to `https://example.com/app.html` which matches via the following registration call which has been previously executed:
>
> ```
> // Script on the page https://example.com/app.html
> navigator.serviceWorker.register("/service_worker.js");
> ```
>
> The value of `navigator.serviceWorker.controller.scriptURL` will be `"https://example.com/service_worker.js"`.

3.1.3. `state`

The ***`state`*** attribute *must* return the value (in `ServiceWorkerState` enumeration) to which it was last set.

3.1.4. `postMessage(message, transfer)`

The ***`postMessage(message, transfer)`*** method *must* run these steps:

1. Let *service-Worker* be the service worker represented by the context object.
2. Let *incumbent-Settings* be the incumbent settings object.
3. Let *incumbentGlobal* be *incumbentSettings*'s global object.
4. Let *serialize-WithTransfer-Result* be StructuredSerializeWithTransfer(*message*, *transfer*). Rethrow any exceptions.
5. If the result of running the Should Skip Event algorithm with "message" and *service-Worker* is true, then return.
6. Run these substeps in parallel:

    1. If the result of running the Run Service Worker algorithm with *serviceWorker* is *failure*, then return.
    2. Queue a task on the DOM manipulation task source to run the following steps:

        1. Let *source* be determined by switching on the type of *incumbentGlobal*:

            ↪ `ServiceWorkerGlobalScope`
            The result of getting the service worker object that represents *incumbentGlobal*'s service worker

in the relevant settings object of *serviceWorker*'s global object.

↪ `Window`
a new `WindowClient` object that represents *incumbentGlobal*'s relevant settings object.

↪ **Otherwise**
a new `Client` object that represents *incumbentGlobal*'s associated worker

2. Let *origin* be the serialization of *incumbentSettings*'s origin.
3. Let *destination* be the `ServiceWorkerGlobalScope` object associated with *serviceWorker*.
4. Let *deserializeRecord* be StructuredDeserializeWithTransfer(*serializeWithTransferResult*, *destination*'s Realm).

   If this throws an exception, catch it, fire an event named `messageerror` at *destination*, using `MessageEvent`, with the `origin` attribute initialized to *origin* and the `source` attribute initialized to *source*, and then abort these steps.
5. Let *messageClone* be *deserializeRecord*.[[Deserialized]].

6. Let *newPorts* be a new frozen array consisting of all `MessagePort` objects in *deserializeRecord*.[[TransferredValues]], if any, maintaining their relative order.
7. Let *e* be the result of creating an event named `message`, using `ExtendableMessageEvent`, with the `origin` attribute initialized to *origin*, the `source` attribute initialized to *source*, the `data` attribute initialized to *messageClone*, and the `ports` attribute initialized to *newPorts*.
8. Dispatch *e* at *destination*.
9. Invoke Update Service Worker Extended Events Set with *serviceWorker* and *e*.

§ 3.1.5. Event handler

The following is the event handler (and its corresponding event handler event type) that *must* be supported, as event handler IDL attributes, by all objects implementing `ServiceWorker` interface:

event handler	event handler event type
onstatechange	`statechange`

§ 3.2. ServiceWorkerRegistration

```
[SecureContext, Exposed=(Window,Worker)]
interface ServiceWorkerRegistration : EventTarget {
  readonly attribute ServiceWorker? installing;
  readonly attribute ServiceWorker? waiting;
  readonly attribute ServiceWorker? active;

  readonly attribute USVString scope;
  readonly attribute ServiceWorkerUpdateViaCache updateViaCache;

  [NewObject] Promise<void> update();
  [NewObject] Promise<boolean> unregister();

  // event
  attribute EventHandler onupdatefound;
};

enum ServiceWorkerUpdateViaCache {
  "imports",
  "all",
  "none"
};
```

A ServiceWorkerRegistration has a ***service worker registration*** (a service worker registration).

§ 3.2.1. Getting ServiceWorkerRegistration instances

An environment settings object has a ***service worker registration object map***, a map where the keys are service worker registrations and the values are ServiceWorkerRegistration objects.

To ***get the service worker registration object*** representing *registration* (a service worker registration) in *environment* (an envir-

onment settings object), run these steps:

1. Let *objectMap* be *environment*'s service worker registration object map.
2. If *objectMap*[*registration*] does not exist, then:

    1. Let *registrationObject* be a new `ServiceWorkerRegistration` in *environment*'s Realm.
    2. Set *registrationObject*'s service worker registration to *registration*.
    3. Set *registrationObject*'s `installing` attribute to null.
    4. Set *registrationObject*'s `waiting` attribute to null.
    5. Set *registrationObject*'s `active` attribute to null.
    6. If *registration*'s installing worker is not null, then set *registrationObject*'s `installing` attribute to the result of getting the service worker object that represents *registration*'s installing worker in *environment*.
    7. If *registration*'s waiting worker is not null, then set *registrationObject*'s `waiting` attribute to the result of getting the service worker object that rep-

resents *registration*'s waiting worker in *environment*.

8. If *registration*'s active worker is not null, then set *registrationObject*'s `active` attribute to the result of getting the service worker object that represents *registration*'s active worker in *environment*.
9. Set *objectMap*[*registration*] to *registrationObject*.

3. Return *objectMap*[*registration*].

3.2.2. `installing`

`installing` attribute *must* return the value to which it was last set.

> Note: Within a Realm, there is only one `ServiceWorker` object per associated service worker.

3.2.3. `waiting`

`waiting` attribute *must* return the value to which it was last set.

> Note: Within a Realm, there is only one `ServiceWorker` object per associated service worker.

3.2.4. `active`

`active` attribute *must* return the

value to which it was last set.

> Note: Within a Realm, there is only one `ServiceWorker` object per associated service worker.

3.2.5. `scope`

The ***`scope`*** attribute *must* return service worker registration's serialized scope url.

> EXAMPLE 3
> In the example in § 3.1.2 scriptURL, the value of `registration.scope`, obtained from `navigator.serviceWorker.ready.then(registration => console.log(registration.scope))` for example, will be `"https://example.com/"`.

3.2.6. `updateViaCache`

The ***`updateViaCache`*** attribute *must* return service worker registration's update via cache mode.

3.2.7. `update()`

`update()` method *must* run these steps:

1. Let *registration* be the service worker registration.
2. Let *newestWorker* be the

result of running Get Newest Worker algorithm passing *registration* as its argument.

3. If *newestWorker* is null, return a promise rejected with an "`InvalidStateError`" `DOMException` and abort these steps.
4. If the context object's relevant settings object's global object *globalObject* is a `ServiceWorkerGlobalScope` object, and *globalObject*'s associated service worker's state is "`installing`", return a promise rejected with an "`InvalidStateError`" `DOMException` and abort these steps.
5. Let *promise* be a promise.
6. Let *job* be the result of running Create Job with *update*, *registration*'s scope url, *newestWorker*'s script url, *promise*, and the context object's relevant settings object.
7. Set *job*'s worker type to *newestWorker*'s type.
8. Invoke Schedule Job with *job*.
9. Return *promise*.

§ **3.2.8. `unregister()`**

> Note: The `unregister()` method unregisters the service worker registration. It is important to note that the currently controlled service worker client's active service worker's containing service worker registration is effective until all the service worker clients (including itself) using this service worker registration unload. That is, the `unregister()` method only affects subsequent navigations.

`unregister()` method *must* run these steps:

1. Let *promise* be a new promise.
2. Let *job* be the result of running Create Job with *unregister*, the scope url of the service worker registration, null, *promise*, and the context object's relevant settings object.
3. Invoke Schedule Job with *job*.
4. Return *promise*.

§ 3.2.9. Event handler

The following is the event handler (and its corresponding event handler event type) that *must* be

supported, as event handler IDL attributes, by all objects implementing ServiceWorkerRegistration interface:

event handler	event handler event type
onupdatefound	updatefound

§ 3.3. navigator.serviceWorker

```
partial interface Navigator {
  [SecureContext, SameObject] readonly attribute ServiceWorkerContainer serviceWorker;
};

partial interface WorkerNavigator {
  [SecureContext, SameObject] readonly attribute ServiceWorkerContainer serviceWorker;
};
```

The *serviceWorker* attribute *must* return the ServiceWorkerContainer object that is associated with the context object.

§ 3.4. ServiceWorkerContainer

```
[SecureContext, Exposed=(Window,Worker)]
interface ServiceWorkerContainer : EventTarget {
  readonly attribute ServiceWorker? controller;
  readonly attribute Promise<ServiceWorkerRegistration> ready;

  [NewObject] Promise<ServiceWorkerRegistration> register(USVString scriptURL, optional Re

  [NewObject] Promise<any> getRegistration(optional USVString clientURL = "");
  [NewObject] Promise<FrozenArray<ServiceWorkerRegistration>> getRegistrations();

  void startMessages();


  // events
  attribute EventHandler oncontrollerchange;
  attribute EventHandler onmessage; // event.source of message events is ServiceWorker obj
  attribute EventHandler onmessageerror;
};
```

```
dictionary RegistrationOptions {
  USVString scope;
  WorkerType type = "classic";
  ServiceWorkerUpdateViaCache updateViaCache = "imports";
};
```

The user agent *must* create a `ServiceWorkerContainer` object when a `Navigator` object or a `WorkerNavigator` object is created and associate it with that object.

A `ServiceWorkerContainer` provides capabilities to register, unregister, and update the service worker registrations, and provides access to the state of the service worker registrations and their associated service workers.

A `ServiceWorkerContainer` has an associated ***service worker client***, which is a service worker client whose global object is associated with the `Navigator` object or the `WorkerNavigator` object that the `ServiceWorkerContainer` is retrieved from.

A `ServiceWorkerContainer` object has an associated ***ready promise*** (a promise). It is

initially set to a new promise.

A `ServiceWorkerContainer` object has a task source called the ***client message queue***, initially empty. A client message queue can be enabled or disabled, and is initially disabled. When a `ServiceWorkerContainer` object's client message queue is enabled, the event loop *must* use it as one of its task sources. When the `ServiceWorkerContainer` object's relevant global object is a `Window` object, all tasks queued on its client message queue *must* be associated with its relevant settings object's responsible document.

3.4.1. `controller`

`controller` attribute *must* run these steps:

1. Let *client* be the context object's service worker client.
2. If *client*'s active service worker is null, then return null.
3. Return the result of getting the service worker object that represents *client*'s active service worker in the context object's

relevant settings object.

Note: `navigator.serviceWorker.controller` returns `null` if the request is a force refresh (shift+refresh).

3.4.2. `ready`

ready attribute *must* run these steps:

1. Let *readyPromise* be the context object's ready promise.
2. If *readyPromise* is pending, run the following substeps in parallel:

    1. Let *registration* be the result of running Match Service Worker Registration with the context object's service worker client's creation URL.
    2. If *registration* is not null, and *registration*'s active worker is not null, queue a task on *readyPromise*'s relevant settings object's responsible event loop, using the DOM manipulation task source, to resolve *readyPromise* with the result of getting the service worker registration object that represents *registration* in *readyPromise*'s relevant settings object.

3. Return *readyPromise*.

> Note: The returned ready promise will never reject. If it does not resolve in this algorithm, it will eventually resolve when a matching service worker registration is registered and its active worker is set. (See the relevant Activate algorithm step.)

3.4.3. `register(scriptURL, options)`

> Note: The `register(scriptURL, options)` method creates or updates a service worker registration for the given scope url. If successful, a service worker registration ties the provided *scriptURL* to a scope url, which is subsequently used for navigation matching.

***`register(`*`scriptURL, options`*`)`** method *must* run these steps:

1. Let *p* be a promise.
2. Let *client* be the context object's service worker client.
3. Let *scriptURL* be the result of parsing *scriptURL* with the context object's relevant settings

object's API base URL.

4. If *scriptURL* is failure, reject *p* with a `TypeError` and abort these steps.
5. Set *scriptURL*'s fragment to null.

   > Note: The user agent does not store the fragment of the script's url. This means that the fragment does not have an effect on identifying service workers.

6. If *scriptURL*'s scheme is not one of "`http`" and "`https`", reject *p* with a `TypeError` and abort these steps.
7. If any of the strings in *scriptURL*'s path contains either ASCII case-insensitive "`%2f`" or ASCII case-insensitive "`%5c`", reject *p* with a `TypeError` and abort these steps.
8. Let *scopeURL* be null.
9. If *options*.`scope` is not present, set *scopeURL* to the result of parsing a string "`./`" with *scriptURL*.

   > Note: The scope url for the registration is set to the location of the service worker script by default.

10. Else, set *scopeURL* to the

result of parsing *options*.`scope` with the context object's relevant settings object's API base URL.

11. If *scopeURL* is failure, reject *p* with a `TypeError` and abort these steps.
12. Set *scopeURL*'s fragment to null.

    > Note: The user agent does not store the fragment of the scope url. This means that the fragment does not have an effect on identifying service worker registrations.

13. If *scopeURL*'s scheme is not one of "`http`" and "`https`", reject *p* with a `TypeError` and abort these steps.
14. If any of the strings in *scopeURL*'s path contains either ASCII case-insensitive "`%2f`" or ASCII case-insensitive "`%5c`", reject *p* with a `TypeError` and abort these steps.
15. Let *job* be the result of running Create Job with *register*, *scopeURL*, *scriptURL*, *p*, and *client*.
16. Set *job*'s worker type to *options*.`type`.
17. Set *job*'s update via cache mode to

options.updateViaCache.

18. Invoke Schedule Job with *job*.
19. Return *p*.

§ **3.4.4. getRegistration(clientURL)**

getRegistration(clientURL) method *must* run these steps:

1. Let *client* be the context object's service worker client.
2. Let *clientURL* be the result of parsing *clientURL* with the context object's relevant settings object's API base URL.
3. If *clientURL* is failure, return a promise rejected with a `TypeError`.
4. Set *clientURL*'s fragment to null.
5. If the origin of *clientURL* is not *client*'s origin, return a *promise* rejected with a "`SecurityError`" `DOMException`.
6. Let *promise* be a new promise.
7. Run the following substeps in parallel:

    1. Let *registration* be the result of running Match Service Worker Registration algorithm with *clientURL* as its argument.
    2. If *registration* is null, resolve *promise* with undefined and

abort these steps.

3. Resolve *promise* with the result of getting the service worker registration object that represents *registration* in *promise*'s relevant settings object.

8. Return *promise*.

3.4.5. `getRegistrations()`

`getRegistrations()` method *must* run these steps:

1. Let *client* be the context object's service worker client.
2. Let *promise* be a new promise.
3. Run the following steps in parallel:

    1. Let *registrations* be a new list.
    2. For each *scope* → *registration* of scope to registration map:

        1. If the origin of the result of parsing *scope* is the same as *client*'s origin, then append *registration* to *registrations*.

    3. Queue a task on *promise*'s relevant settings object's responsible event loop, using the DOM manipulation task source, to run the following steps:

        1. Let *registrationObjects* be a new list.

2. For each *registration* of *registrations*:

    1. Let *registrationObj* be the result of getting the service worker registration object that represents *registration* in *promise*'s relevant settings object.
    2. Append *registrationObj* to *registrationObjects*.

3. Resolve *promise* with a new frozen array of *registrationObjects* in *promise*'s relevant Realm.

4. Return *promise*.

3.4.6. `startMessages()`

`startMessages()` method *must* enable the context object's client message queue if it is not enabled.

3.4.7. Event handlers

The following are the event handlers (and their corresponding event handler event types) that *must* be supported, as event handler IDL attributes, by all objects implementing the `ServiceWorkerContainer` interface:

event handler	event handler event type
`oncontrollerchange`	`controllerchange`
`onmessage`	`message`

event handler	event handler event type
onmessageerror	messageerror

The first time the context object's onmessage IDL attribute is set, its client message queue *must* be enabled.

§ 3.5. Events

The following event is dispatched on ServiceWorker object:

Event name	Interface	Dispatched when...
statechange	Event	The state attribute of the ServiceWorker object is changed.

The following event is dispatched on ServiceWorkerRegistration object:

Event name	Interface	Dispatched when...
updatefound	Event	The service worker registration's installing worker changes. (See step 8 of the Install algorithm.)

The following events are dispatched on ServiceWorkerContainer object:

Event name	Interface	Dispatched when...

Event name	Interface	Dispatched when...
controllerchange	Event	The service worker client's active service worker changes. (See step 9.2 of the Activate algorithm. The skip waiting flag of a service worker causes activation of the service worker registration to occur while service worker clients are using the service worker registration, navigator.serviceWorker.controller immediately reflects the active worker as the service worker that controls the service worker client.)

§ 4. Execution Context

EXAMPLE 4

Serving Cached Resources:

```
// caching.js
self.addEventListener("install", event => {
  event.waitUntil(
    // Open a cache of resources.
    caches.open("shell-v1").then(cache => {
      // Begins the process of fetching them.
      // The coast is only clear when all the resources are ready.
      return cache.addAll([
        "/app.html",
        "/assets/v1/base.css",
        "/assets/v1/app.js",
        "/assets/v1/logo.png",
        "/assets/v1/intro_video.webm"
      ]);
    })
  );
});

self.addEventListener("fetch", event => {
  // No "fetch" events are dispatched to the service worker until it
  // successfully installs and activates.

  // All operations on caches are async, including matching URLs, so we use
  // promises heavily. e.respondWith() even takes promises to enable this:
  event.respondWith(
    caches.match(e.request).then(response => {
      return response || fetch(e.request);
    }).catch(() => {
      return caches.match("/fallback.html");
    })
  );
});
```

4.1. ServiceWorkerGlobalScope

```
[Global=(Worker,ServiceWorker), Exposed=ServiceWorker]
interface ServiceWorkerGlobalScope : WorkerGlobalScope {
  [SameObject] readonly attribute Clients clients;
  [SameObject] readonly attribute ServiceWorkerRegistration registration;

  [NewObject] Promise<void> skipWaiting();

  attribute EventHandler oninstall;
  attribute EventHandler onactivate;
  attribute EventHandler onfetch;

  attribute EventHandler onmessage;
  attribute EventHandler onmessageerror;
};
```

A ServiceWorkerGlobalScope object represents the global execution context of a service worker. A

ServiceWorkerGlobalScope object has an associated ***service worker*** (a service worker). A ServiceWorkerGlobalScope object has an associated ***force bypass cache for import scripts flag***. It is initially unset.

> Note: ServiceWorkerGlobalScope object provides generic, event-driven, time-limited script execution contexts that run at an origin. Once successfully registered, a service worker is started, kept alive and killed by their relationship to events, not service worker clients. Any type of synchronous requests must not be initiated inside of a service worker.

4.1.1. `clients`

The ***`clients`*** attribute *must* return the Clients object that is associated with the context object.

4.1.2. `registration`

The ***`registration`*** attribute *must* return the result of getting the service worker registration object

representing the context object's service worker's containing service worker registration in context object's relevant settings object.

4.1.3. `skipWaiting()`

> Note: The `skipWaiting()` method allows this service worker to progress from the registration's waiting position to active even while service worker clients are using the registration.

`skipWaiting()` method *must* run these steps:

1. Let *promise* be a new promise.
2. Run the following substeps in parallel:
    1. Set service worker's skip waiting flag.
    2. Invoke Try Activate with service worker's containing service worker registration.
    3. Resolve *promise* with undefined.
3. Return *promise*.

4.1.4. Event handlers

The following are the event handlers (and their corresponding event handler event types) that *must* be supported, as

event handler IDL attributes, by all objects implementing the ServiceWorkerGlobalScope interface:

event handler	event handler event type
oninstall	install
onactivate	activate
onfetch	fetch
onmessage	message
onmessageerror	messageerror

§ 4.2. Client

```
[Exposed=ServiceWorker]
interface Client {
  readonly attribute USVString url;
  readonly attribute FrameType frameType;
  readonly attribute DOMString id;
  readonly attribute ClientType type;
  void postMessage(any message, optional sequence<object> transfer = []);
};

[Exposed=ServiceWorker]
interface WindowClient : Client {
  readonly attribute VisibilityState visibilityState;
  readonly attribute boolean focused;
  [SameObject] readonly attribute FrozenArray<USVString> ancestorOrigins;
  [NewObject] Promise<WindowClient> focus();
  [NewObject] Promise<WindowClient?> navigate(USVString url);
};

enum FrameType {
  "auxiliary",
  "top-level",
  "nested",
  "none"
};
```

A Client object has an associated ***service worker client*** (a service worker client).

A Client object has an associated ***frame type***, which is one of "auxiliary", "top-level", "nested", and "none". Unless stated otherwise it is "none".

A `WindowClient` object has an associated ***browsing context***, which is its service worker client's global object's browsing context.

A `WindowClient` object has an associated ***visibility state***, which is one of `visibilityState` attribute value.

A `WindowClient` object has an associated ***focus state***, which is either true or false (initially false).

A `WindowClient` object has an associated ***ancestor origins array***.

4.2.1. `url`

The ***`url`*** attribute *must* return the context object's associated service worker client's serialized creation URL.

4.2.2. `frameType`

The ***`frameType`*** attribute *must* return the context object's frame type.

4.2.3. `id`

The ***`id`*** attribute *must* return its associated ser-

vice worker client's id.

4.2.4. `type`

The ***type*** attribute *must* run these steps:

1. Let *client* be context object's service worker client.
2. If *client* is an environment settings object, then:
    1. If *client* is a window client, return `"window"`.
    2. Else if *client* is a dedicated worker client, return `"worker"`.
    3. Else if *client* is a shared worker client, return `"sharedworker"`.
3. Else:
    1. Return `"window"`.

4.2.5. `postMessage(message, transfer)`

The ***postMessage(*`message, transfer`*)*** method *must* run these steps:

1. Let *contextObject* be the context object.
2. Let *sourceSettings* be the *contextObject*'s relevant settings object.
3. Let *serializeWithTransferResult* be StructuredSerializeWithTransfer(*message*, *transfer*).

Rethrow any exceptions.

4. Run the following steps in parallel:

    1. Let *targetClient* be null.
    2. For each service worker client *client*:

        1. If *client* is the *contextObject*'s service worker client, set *targetClient* to *client*, and break.

    3. If *targetClient* is null, return.
    4. Let *destination* be the `ServiceWorkerContainer` object whose associated service worker client is *targetClient*.
    5. Add a task that runs the following steps to *destination*'s client message queue:

        1. Let *origin* be the serialization of *sourceSettings*'s origin.
        2. Let *source* be the result of getting the service worker object that represents *contextObject*'s relevant global object's service worker in *targetClient*.
        3. Let *deserializeRecord* be StructuredDeserializeWithTransfer(*serializeWithTransferResult*, *destination*'s relevant Realm).

            If this throws an exception, catch

it, fire an event named `messageerror` at *destination*, using `MessageEvent`, with the `origin` attribute initialized to *origin* and the `source` attribute initialized to *source*, and then abort these steps.

4. Let *messageClone* be *deserializeRecord*.[[Deserialized]].
5. Let *newPorts* be a new frozen array consisting of all `MessagePort` objects in *deserializeRecord*.[[TransferredValues]], if any.
6. Dispatch an event named `message` at *destination*, using `MessageEvent`, with the `origin` attribute initialized to *origin*, the `source` attribute initialized to *source*, the `data` attribute initialized to *messageClone*, and the `ports` attribute initialized to *newPorts*.

4.2.6. `visibilityState`

The ***`visibilityState`*** attribute *must* return the context object's visibility state.

4.2.7. `focused`

The ***`focused`*** attribute *must* return the context

object's focus state.

4.2.8. `ancestorOrigins`

The ***`ancestorOrigins`*** attribute *must* return the context object's associated ancestor origins array.

4.2.9. `focus()`

The ***`focus()`*** method *must* run these steps:

1. If this algorithm is not triggered by user activation, return a promise rejected with an "`InvalidAccessError`" `DOMException`.
2. Let *serviceWorkerEventLoop* be the current global object's event loop.
3. Let *promise* be a new promise.
4. Queue a task to run the following steps on the context object's associated service worker client's responsible event loop using the user interaction task source:

    1. Run the focusing steps with the context object's browsing context.
    2. Let *frameType* be the result of running Get Frame Type with the context object's browsing context.

3. Let *visibilityState* be the context object's browsing context's active document's `visibilityState` attribute value.
4. Let *focusState* be the result of running the has focus steps with the context object's browsing context's active document.
5. Let *ancestorOriginsList* be the context object's browsing context's active document's relevant global object's `Location` object's ancestor origins list's associated list.
6. Queue a task to run the following steps on *serviceWorkerEventLoop* using the DOM manipulation task source:

    1. Let *windowClient* be the result of running Create Window Client with the context object's associated service worker client, *frameType*, *visibilityState*, *focusState*, and *ancestorOriginsList*.
    2. If *windowClient*'s focus state is true, resolve *promise* with *windowClient*.
    3. Else, reject *promise* with a `TypeError`.

5. Return *promise*.

§ **4.2.10. `navigate(url)`**

The ***navigate(url)*** method *must* run these steps:

1. Let *url* be the result of parsing *url* with the context object's relevant settings object's API base URL.
2. If *url* is failure, return a promise rejected with a `TypeError`.
3. If *url* is `about:blank`, return a promise rejected with a `TypeError`.
4. If the context object's associated service worker client's active service worker is not the context object's relevant global object's service worker, return a promise rejected with a `TypeError`.
5. Let *serviceWorkerEventLoop* be the current global object's event loop.
6. Let *promise* be a new promise.
7. Queue a task to run the following steps on the context object's associated service worker client's responsible event loop using the user interaction task source:

    1. Let *browsingContext* be the context object's browsing context.
    2. If *browsingContext* has discarded its `Document`,

queue a task to reject *promise* with a `TypeError`, on *service-WorkerEventLoop* using the DOM manipulation task source, and abort these steps.

3. *HandleNavigate*: Navigate *browsingContext* to *url* with exceptions enabled. The source browsing context must be *browsingContext*.
4. If the algorithm steps invoked in the step labeled *HandleNavigate* throws an exception, queue a task to reject *promise* with the exception, on *service-WorkerEventLoop* using the DOM manipulation task source, and abort these steps.
5. Let *frameType* be the result of running Get Frame Type with *browsingContext*.
6. Let *visibilityState* be *browsingContext*'s active document's `visibilityState` attribute value.
7. Let *focusState* be the result of running the has focus steps with *browsingContext*'s active document.
8. Let *ancestorOriginsList* be *browsingContext*'s active document's relevant

global object's Location object's ancestor origins list's associated list.

9. Queue a task to run the following steps on *service-WorkerEventLoop* using the DOM manipulation task source:

    1. If *browsingContext*'s Window object's environment settings object's creation URL's origin is not the same as the service worker's origin, resolve *promise* with null and abort these steps.
    2. Let *windowClient* be the result of running Create Window Client with the context object's service worker client, *frameType*, *visibilityState*, *focusState*, and *ancestorOriginsList*.
    3. Resolve *promise* with *windowClient*.

8. Return *promise*.

§ 4.3. Clients

```
[Exposed=ServiceWorker]
interface Clients {
  // The objects returned will be new instances every time
  [NewObject] Promise<any> get(DOMString id);
  [NewObject] Promise<FrozenArray<Client>> matchAll(optional ClientQueryOptions options =
  [NewObject] Promise<WindowClient?> openWindow(USVString url);
  [NewObject] Promise<void> claim();
};
```

```
dictionary ClientQueryOptions {
  boolean includeUncontrolled = false;
  ClientType type = "window";
};
```

```
enum ClientType {
  "window",
  "worker",
  "sharedworker",
  "all"
};
```

The user agent *must* create a `Clients` object when a `ServiceWorkerGlobalScope` object is created and associate it with that object.

4.3.1. `get(id)`

The ***get(*id*)*** method *must* run these steps:

1. Let *promise* be a new promise.
2. Run these sub-steps in parallel:
    1. For each service worker client *client* whose origin is the same as the associated service worker's origin:
        1. If *client*'s id is not *id*, continue.
        2. Wait for either *client*'s execution ready flag to be set or for *client*'s discarded flag to be set.
        3. If *client*'s execution ready flag is set, then invoke Resolve Get Client Promise with *client* and *promise*, and abort these steps.
    2. Resolve *promise* with undefined.
3. Return *promise*.

4.3.2. `matchAll(options)`

The ***matchAll(*option**

`s)` method *must* run these steps:

1. Let *promise* be a new promise.
2. Run the following steps in parallel:

    1. Let *targetClients* be a new list.
    2. For each service worker client *client* whose origin is the same as the associated service worker's origin:

        1. If *client*'s execution ready flag is unset or *client*'s discarded flag is set, continue.
        2. If *client* is not a secure context, continue.
        3. If *options*["`includeUncontrolled`"] is false, and if *client*'s active service worker is not the associated service worker, continue.
        4. Add *client* to *targetClients*.

    3. Let *matchedWindowData* be a new list.
    4. Let *matchedClients* be a new list.
    5. For each service worker client *client* in *targetClients*:

        1. If *options*["`type`"] is `"window"` or `"all"`, and *client* is not an environment settings object or is a window client, then:

            1. Let *windowData* be «["client" →

client, "ancestorOrigins List" → a new list]».

2. Let *browsingContext* be null.
3. Let *isClientEnumerable* be true.
4. If *client* is an environment settings object, set *browsingContext* to *client*'s global object's browsing context.
5. Else, set *browsingContext* to *client*'s target browsing context.
6. Queue a task *task* to run the following substeps on *browsingContext*'s event loop using the user interaction task source:

    1. If *browsingContext* has been discarded, then set *isClientEnumerable* to false and abort these steps.
    2. If *client* is a window client and *client*'s responsible document is not *browsingContext*'s active document, then set *isClientEnumerable* to false and abort these steps.
    3. Set *windowData*["`frameType`"] to the result of running Get Frame Type with *browsingContext*.
    4. Set *windowData*["`visibilityState`"] to *browsingContext*'s active document's

`visibilityState` attribute value.

5. Set *windowData*["`focusState`"] to the result of running the has focus steps with *browsingContext*'s active document as the argument.
6. If *client* is a window client, then set *windowData*["`ancestorOriginsList`"] to *browsingContext*'s active document's relevant global object's `Location` object's ancestor origins list's associated list.

7. Wait for *task* to have executed.

> Note: Wait is a blocking wait, but implementers may run the iterations in parallel as long as the state is not broken.

8. If *isClientEnumerable* is true, then:

    1. Add *windowData* to *matchedWindowData*.

2. Else if *options*["`type`"] is `"worker"` or `"all"` and *client* is a dedicated worker client, or *options*["`type`"] is `"sharedworker"` or `"all"` and *client* is a shared worker client, then:

    1. Add *client* to *matchedClients*.

6. Queue a task to run the following steps on *promise*'s relevant settings object's responsible event loop using the DOM manipulation task source:

    1. Let *clientObjects* be a new list.
    2. For each *windowData* in *matchedWindowData*:

        1. Let *windowClient* be the result of running Create Window Client algorithm with *windowData*["`client`"], *windowData*["`frameType`"], *windowData*["`visibilityState`"], *windowData*["`focusState`"], and *windowData*["`ancestorOriginsList`"] as the arguments.
        2. Append *windowClient* to *clientObjects*.

    3. For each *client* in *matchedClients*:

        1. Let *clientObject* be the result of running Create Client algorithm with *client* as the argument.
        2. Append *clientObject* to *clientObjects*.

    4. Sort *clientObjects* such that:

        - `WindowClient` objects whose browsing context has been focused are placed first, sorted in the most

recently focused order.

- `WindowClient` objects whose browsing context has never been focused are placed next, sorted in their service worker client's creation order.
- `Client` objects whose associated service worker client is a worker client are placed next, sorted in their service worker client's creation order.

> Note: Window clients are always placed before worker clients.

5. Resolve *promise* with a new frozen array of *clientObjects* in *promise*'s relevant Realm.

3. Return *promise*.

4.3.3. `openWindow(url)`

The ***`openWindow(`*`url`*`)`*** method *must* run these steps:

1. Let *url* be the result of parsing *url* with the context object's relevant settings object's API base URL.
2. If *url* is failure, return a promise rejected with a `TypeError`.
3. If *url* is `about:blank`, return a promise rejected with a `TypeError`.

4. If this algorithm is not triggered by user activation, return a promise rejected with an "`InvalidAccessError`" `DOMException`.
5. Let *serviceWorkerEventLoop* be the current global object's event loop.
6. Let *promise* be a new promise.
7. Run these substeps in parallel:

    1. Let *newContext* be a new top-level browsing context.
    2. Queue a task to run the following steps on *newContext*'s `Window` object's environment settings object's responsible event loop using the user interaction task source:

        1. *HandleNavigate*: Navigate *newContext* to *url* with exceptions enabled and replacement enabled.
        2. If the algorithm steps invoked in the step labeled *HandleNavigate* throws an exception, queue a task to reject *promise* with the exception, on *serviceWorkerEventLoop* using the DOM manipulation task source, and abort these steps.
        3. Let *frameType* be the result of running Get Frame

Type with *newContext*.

4. Let *visibilityState* be *newContext*'s active document's `visibilityState` attribute value.
5. Let *focusState* be the result of running the has focus steps with *newContext*'s active document as the argument.
6. Let *ancestorOriginsList* be *newContext*'s active document's relevant global object's `Location` object's ancestor origins list's associated list.
7. Queue a task to run the following steps on *serviceWorkerEventLoop* using the DOM manipulation task source:

    1. If *newContext*'s `Window` object's environment settings object's creation URL's origin is not the same as the service worker's origin, resolve *promise* with null and abort these steps.
    2. Let *client* be the result of running Create Window Client with *newContext*'s `Window` object's environment settings object, *frameType*, *visibilityState*, *focusState*, and *ancestorOriginsList* as the arguments.
    3. Resolve *promise* with *client*.

8. Return *promise*.

4.3.4. `claim()`

The ***`claim()`*** method *must* run these steps:

1. If the service worker is not an active worker, return a promise rejected with an "`InvalidStateError`" `DOMException`.
2. Let *promise* be a new promise.
3. Run the following substeps in parallel:

    1. For each service worker client *client* whose origin is the same as the service worker's origin:

        1. If *client*'s execution ready flag is unset or *client*'s discarded flag is set, continue.
        2. If *client* is not a secure context, continue.
        3. Let *registration* be the result of running Match Service Worker Registration algorithm passing *client*'s creation URL as the argument.
        4. If *registration* is not the service worker's containing service worker registration, continue.

            Note: *registration* will be null if the service worker's containing service worker registration is unregistered.

5. If *client*'s active service worker is not the service worker, then:

    1. Invoke Handle Service Worker Client Unload with *client* as the argument.
    2. Set *client*'s active service worker to service worker.
    3. Invoke Notify Controller Change algorithm with *client* as the argument.

2. Resolve *promise* with undefined.

4. Return *promise*.

4.4. ExtendableEvent

```
[Constructor(DOMString type, optional ExtendableEventInit eventInitDict = {}), Exposed=Ser
interface ExtendableEvent : Event {
  void waitUntil(Promise<any> f);
};
```

```
dictionary ExtendableEventInit : EventInit {
  // Defined for the forward compatibility across the derived events
};
```

An ExtendableEvent object has an associated ***extend lifetime promises*** (an array of promises). It is initially an empty array.

An ExtendableEvent object has an associated ***pending promises count*** (the number of pending promises in the extend lifetime promises). It is initially set to zero.

An ExtendableEvent object has an associated ***timed out flag***. It is initially unset, and is set after an optional user agent imposed delay if the pending promises count is greater than zero.

An ExtendableEvent object is said to be ***active*** when its timed out flag is unset and either its pending promises count is greater than zero or its dispatch flag is set.

Service workers have two lifecycle events, `install` and `activate`. Service workers use the ExtendableEvent interface for `activate` event and `install` event.

Service worker extensions that define event handlers *may* also use or extend the ExtendableEvent interface.

4.4.1. `event.waitUntil(f)`

> Note: `waitUntil()` method extends the lifetime of the event.

`waitUntil(f)` method *must* run these steps:

1. Let *event* be the context object.
2. Add lifetime promise *f* to *event*.

To ***add lifetime promise*** *promise* (a promise) to *event* (an `ExtendableEvent`), run these steps:

1. If *event*'s `isTrusted` attribute is false, throw an "`InvalidStateError`" `DOMException`.
2. If *event* is not active, throw an "`InvalidStateError`" `DOMException`.

   > Note: If no lifetime extension promise has been added in the task that called the event handlers, calling `waitUntil()` in subsequent asynchronous tasks will throw.

3. Add *promise* to *event*'s extend lifetime promises.
4. Increment *event*'s pending promises count by one.

> Note: The pending promises count is incremented even if the given promise has already been settled. The corresponding count decrement is done in the microtask queued by the reaction to the promise.

5. Upon fulfillment or rejection of *promise*, queue a microtask to run these substeps:

    1. Decrement *event*'s pending promises count by one.
    2. If *event*'s pending promises count is 0, then:

        1. Let *registration* be the current global object's associated service worker's containing service worker registration.
        2. If *registration* is unregistered, invoke Try Clear Registration with *registration*.
        3. If *registration* is not null, invoke Try Activate with *registration*.

The user agent *should not* terminate a service worker if Service Worker Has No Pending Events returns false for that service worker.

Service workers and extensions

that define event handlers *may* define their own behaviors, allowing the extend lifetime promises to suggest operation length, and the rejected state of any of the promise in extend lifetime promises to suggest operation failure.

Note: Service workers delay treating the installing worker as "installed" until all the promises in the install event's extend lifetime promises resolve successfully. (See the relevant Install algorithm step.) If any of the promises rejects, the installation fails. This is primarily used to ensure that a service worker is not considered "installed" until all of the core caches it depends on are populated. Likewise, service workers delay treating the active worker as "activated" until all the promises in the activate event's extend lifetime promises settle. (See the relevant Activate algorithm step.) This is primarily used to ensure that any functional events are not dispatched to the service worker until it upgrades database schemas and deletes the outdated cache entries.

§ 4.5. FetchEvent

```
[Constructor(DOMString type, FetchEventInit eventInitDict), Exposed=ServiceWorker]
interface FetchEvent : ExtendableEvent {
  [SameObject] readonly attribute Request request;
  readonly attribute DOMString clientId;

  void respondWith(Promise<Response> r);
};
```

```
dictionary FetchEventInit : ExtendableEventInit {
  required Request request;
  DOMString clientId = "";
};
```

Service workers have an essential functional event `fetch`. For `fetch` event, service workers use the `FetchEvent` interface which extends the `ExtendableEvent` interface.

Each event using `FetchEvent` interface has an associated ***potential response*** (a response), initially set to null, and the following associated flags that are initially unset:

- ***wait to respond flag***
- ***respond-with entered flag***
- ***respond-with error flag***

§ 4.5.1. `event.request`

request attribute *must* return the value it was initialized to.

§ 4.5.2. `event.clientId`

clientId attribute *must* return the value it was initialized to.

When an event is created the attribute *must* be initialized to the empty string.

4.5.3. `event.respondWith(r)`

> Note: Developers can set the argument *r* with either a promise that resolves with a `Response` object or a `Response` object (which is automatically cast to a promise). Otherwise, a network error is returned to Fetch. Renderer-side security checks about tainting for cross-origin content are tied to the types of filtered responses defined in Fetch.

`respondWith(r)` method *must* run these steps:

1. Let *event* be the context object.
2. If *event*'s dispatch flag is unset, throw an "`InvalidStateError`" `DOMException`.
3. If *event*'s respond-with entered flag is set, throw an "`InvalidStateError`" `DOMException`.
4. Add lifetime promise *r* to *event*.

> Note: `event.respondWith(r)` extends the lifetime of the event by default as if `event.waitUntil(r)` is called.

5. Set *event*'s stop propagation flag and stop immediate propagation flag.
6. Set *event*'s respond-with entered flag.
7. Set *event*'s wait to respond flag.
8. Let *targetRealm* be *event*'s relevant Realm.
9. Upon rejection of *r*:

    1. Set *event*'s respond-with error flag.
    2. Unset *event*'s wait to respond flag.

10. Upon fulfillment of *r* with *response*:

    1. If *response* is not a `Response` object, then set the respond-with error flag.

        > Note: If the respond-with error flag is set, a network error is returned to Fetch through Handle Fetch algorithm. (See the step 21.1.) Otherwise, the value *response* is returned to Fetch through Handle Fetch algorithm. (See the step 22.1.)

    2. Else:

        1. Let *bytes* be an empty byte

sequence.

2. Let *end-of-body* be false.
3. Let *done* be false.
4. Let *potentialResponse* be a copy of *response*'s associated response, except for its body.
5. If *response*'s body is non-null, run these substeps:

    1. Let *reader* be the result of getting a reader from *response*'s body's stream.
    2. Let *highWaterMark* be a non-negative, non-NaN number, chosen by the user agent.
    3. Let *sizeAlgorithm* be an algorithm that accepts a chunk object and returns a non-negative, non-NaN, non-infinite number, chosen by the user agent.
    4. Let *pull* be an action that runs these subsubsteps:

        1. Let *promise* be the result of reading a chunk from *response*'s body's stream with *reader*.
        2. When *promise* is fulfilled with an object whose `done` property is false and whose `value` property is a `Uint8Array` object, append the bytes represented by the `value` property to *bytes* and per-

form ! DetachArrayBuffer with the `ArrayBuffer` object wrapped by the `value` property.

3. When *promise* is fulfilled with an object whose `done` property is true, set *end-of-body* to true.
4. When *promise* is fulfilled with a value that matches with neither of the above patterns, or *promise* is rejected, error *newStream* with a `TypeError`.

5. Let *cancel* be an action that cancels *response*'s body's stream with *reader*.
6. Let *newStream* be the result of construct a ReadableStream object with *highWaterMark*, *sizeAlgorithm*, *pull*, and *cancel* in *targetRealm*.
7. Set *potentialResponse*'s body to a new body whose stream is *newStream*.
8. Run these substeps repeatedly in parallel while *done* is false:

    1. If *newStream* is errored, then set *done* to true.
    2. Otherwise, if *bytes* is empty and *end-of-body* is true, then close *newStream* and set *done* to true.
    3. Otherwise, if *bytes* is not

empty, run these subsubsubsteps:

1. Let *chunk* be a subsequence of *bytes* starting from the beginning of *bytes*.
2. Remove *chunk* from *bytes*.
3. Let *buffer* be an `ArrayBuffer` object created in *targetRealm* and containing *chunk*.
4. Enqueue a `Uint8Array` object created in *targetRealm* and wrapping *buffer* to *newStream*.

Note: These substeps are meant to produce the observable equivalent of "piping" *response*'s body's stream into *potentialResponse*.

6. Set *event*'s potential response to *potentialResponse*.

3. Unset *event*'s wait to respond flag.

§ 4.6. ExtendableMessageEvent

```
[Constructor(DOMString type, optional ExtendableMessageEventInit eventInitDict = {}), Expo
interface ExtendableMessageEvent : ExtendableEvent {
  readonly attribute any data;
  readonly attribute USVString origin;
  readonly attribute DOMString lastEventId;
  [SameObject] readonly attribute (Client or ServiceWorker or MessagePort)? source;
  readonly attribute FrozenArray<MessagePort> ports;
};
```

```
dictionary ExtendableMessageEventInit : ExtendableEventInit {
  any data = null;
  USVString origin = "";
  DOMString lastEventId = "";
  (Client or ServiceWorker or MessagePort)? source = null;
  sequence<MessagePort> ports = [];
};
```

Service workers define the extendable `message` event to allow extending the lifetime of the event. For the `message` event, service workers use the `ExtendableMessageEvent` interface which extends the `ExtendableEvent` interface.

4.6.1. `event.data`

The ***`data`*** attribute *must* return the value it was initialized to. When the object is created, this attribute *must* be initialized to null. It represents the message being sent.

4.6.2. `event.origin`

The ***`origin`*** attribute *must* return the value it was initialized to. When the object is created, this attribute *must* be initialized to the empty string. It represents the origin of the service worker client that sent the message.

4.6.3. `event.lastEventId`

The ***`lastEventId`*** attribute *must* return the value it was initialized to. When the object is created,

this attribute *must* be initialized to the empty string.

4.6.4. `event.source`

The ***`source`*** attribute *must* return the value it was initialized to. When the object is created, this attribute *must* be initialized to null. It represents the `Client` object from which the message is sent.

4.6.5. `event.ports`

The ***`ports`*** attribute *must* return the value it was initialized to. When the object is created, this attribute *must* be initialized to the empty array. It represents the `MessagePort` array being sent.

4.7. Events

The following events, called service worker events, are dispatched on `ServiceWorkerGlobalScope` object:

Event name	Interface	Category	Dispatched when…

Event name	Interface	Category	Dispatched when…
install	ExtendableEvent	Lifecycle	The service worker's containing service worker registration's installing worker changes. (See step 11.2 of the Install algorithm.)
activate	ExtendableEvent	Lifecycle	The service worker's containing service worker registration's active worker changes. (See step 12.2 of the Activate algorithm.)
fetch	FetchEvent	Functional	The http fetch invokes Handle Fetch with *request*. As a result of performing Handle Fetch, the service worker returns a response to the http fetch. The response, represented by a Response object, can be retrieved from a Cache object or directly from network using self.fetch(input, init) method. (A custom Response object can be another option.)
push	PushEvent	Functional	(See Firing a push event.)
notificationclick	NotificationEvent	Functional	(See Activating a notification.)
notificationclose	NotificationEvent	Functional	(See Closing a notification.)
sync	SyncEvent	Functional	(See Firing a sync event.)

Event name	Interface	Category	Dispatched when…
canmakepayment	CanMakePaymentEvent	Functional	(See Handling a CanMakePaymentEvent.)
paymentrequest	PaymentRequestEvent	Functional	(See Handling a PaymentRequestEvent.)
message	`ExtendableMessageEvent`	Legacy	When it receives a message.
messageerror	`MessageEvent`	Legacy	When it was sent a message that cannot be deserialized.

§ 5. Caches

To allow authors to fully manage their content caches for off-line use, the `Window` and the `WorkerGlobalScope` provide the asynchronous caching methods that open and manipulate `Cache` objects. An origin can have multiple, named `Cache` objects, whose contents are entirely under the control of scripts. Caches are not shared across origins, and they are completely isolated from the browser's HTTP cache.

§ 5.1. Constructs

A ***request response list*** is a list of pairs consisting of a request (a request) and a response (a response).

The ***relevant request response list*** is the in-

stance that the context object represents.

A ***name to cache map*** is an ordered map whose entry consists of a key (a string that represents the name of a request response list) and a value (a request response list).

Each origin has an associated name to cache map.

The ***relevant name to cache map*** is the instance of the context object's associated global object's environment settings object's origin.

§ 5.2. Understanding Cache Lifetimes

The Cache instances are not part of the browser's HTTP cache. The Cache objects are exactly what authors have to manage themselves. The Cache objects do not get updated unless authors explicitly request them to be. The Cache objects do not expire unless authors delete the entries. The Cache objects do not disappear just because the service worker script is updated. That is,

caches are not updated automatically. Updates must be manually managed. This implies that authors should version their caches by name and make sure to use the caches only from the version of the service worker that can safely operate on.

§ 5.3. self.caches

```
partial interface WindowOrWorkerGlobalScope {
  [SecureContext, SameObject] readonly attribute CacheStorage caches;
};
```

§ 5.3.1. caches

caches attribute *must* return this object's associated CacheStorage object.

§ 5.4. Cache

```
[SecureContext, Exposed=(Window,Worker)]
interface Cache {
  [NewObject] Promise<any> match(RequestInfo request, optional CacheQueryOptions options =
  [NewObject] Promise<FrozenArray<Response>> matchAll(optional RequestInfo request, optiona
  [NewObject] Promise<void> add(RequestInfo request);
  [NewObject] Promise<void> addAll(sequence<RequestInfo> requests);
  [NewObject] Promise<void> put(RequestInfo request, Response response);
  [NewObject] Promise<boolean> delete(RequestInfo request, optional CacheQueryOptions opti
  [NewObject] Promise<FrozenArray<Request>> keys(optional RequestInfo request, optional Ca
};
```

```
dictionary CacheQueryOptions {
  boolean ignoreSearch = false;
  boolean ignoreMethod = false;
  boolean ignoreVary = false;
};
```

A Cache object represents a request response list. Multiple separate objects implementing the Cache inter-

face across documents and workers can all be associated with the same request response list simultaneously.

A ***cache batch operation*** is a struct that consists of:

- A ***type*** ("`delete`" or "`put`").
- A ***request*** (a request).
- A ***response*** (a response).
- An ***options*** (a `CacheQueryOptions`).

5.4.1. `match(request, options)`

***`match(`*`request, options`*`)`** method *must* run these steps:

1. Let *promise* be a new promise.
2. Run these substeps in parallel:

    1. Let *p* be the result of running the algorithm specified in `matchAll(request, options)` method with *request* and *options*.
    2. Wait until *p* settles.
    3. If *p* rejects with an exception, then:

        1. Reject *promise* with that exception.

    4. Else if *p* resolves with an array, *responses*, then:

        1. If *responses* is an empty array, then:

1. Resolve *promise* with undefined.

2. Else:

    1. Resolve *promise* with the first element of *responses*.

3. Return *promise*.

5.4.2. `matchAll(request, options)`

matchAll(*`request, options`*) method *must* run these steps:

1. Let *r* be null.
2. If the optional argument *request* is not omitted, then:

    1. If *request* is a `Request` object, then:

        1. Set *r* to *request*'s request.
        2. If *r*'s method is not `GET` and *options*.ignoreMethod is false, return a promise resolved with an empty array.

    2. Else if *request* is a string, then:

        1. Set *r* to the associated request of the result of invoking the initial value of `Request` as constructor with *request* as its argument. If this throws an exception, return a promise rejected with that exception.

3. Let *realm* be the context object's relevant realm.
4. Let *promise* be a new promise.
5. Run these substeps in parallel:

1. Let *responses* be an empty list.
2. If the optional argument *request* is omitted, then:
    1. For each *requestResponse* of the relevant request response list:
        1. Add a copy of *requestResponse*'s response to *responses*.
3. Else:
    1. Let *requestResponses* be the result of running Query Cache with *r* and *options*.
    2. For each *requestResponse* of *requestResponses*:
        1. Add a copy of *requestResponse*'s response to *responses*.
4. Queue a task, on *promise*'s relevant settings object's responsible event loop using the DOM manipulation task source, to perform the following steps:
    1. Let *responseList* be a list.
    2. For each *response* of *responses*:
        1. Add a new `Response` object associated with *response* and a new `Headers` object whose guard is "`immutable`" to *responseList*.
    3. Resolve *promise* with a frozen array created from

responseList, in *realm*.

6. Return *promise*.

5.4.3. `add(request)`

add(*`request`*) method *must* run these steps:

1. Let *requests* be an array containing only *request*.
2. Let *responseArrayPromise* be the result of running the algorithm specified in `addAll(requests)` passing *requests* as the argument.
3. Return the result of transforming *responseArrayPromise* with a fulfillment handler that returns undefined.

5.4.4. `addAll(requests)`

addAll(*`requests`*) method *must* run these steps:

1. Let *responsePromises* be an empty list.
2. Let *requestList* be an empty list.
3. For each *request* whose type is `Request` in *requests*:

    1. Let *r* be *request*'s request.
    2. If *r*'s url's scheme is not one of "`http`" and "`https`", or *r*'s method is not `` `GET` ``, return a promise rejected

with a `TypeError`.

4. For each *request* in *requests*:

    1. Let *r* be the associated request of the result of invoking the initial value of `Request` as constructor with *request* as its argument. If this throws an exception, return a promise rejected with that exception.
    2. If *r*'s url's scheme is not one of "`http`" and "`https`", then:

        1. Terminate all the ongoing fetches initiated by *requests* with the aborted flag set.
        2. Return a promise rejected with a `TypeError`.

    3. If *r*'s client's global object is a `ServiceWorkerGlobalScope` object, set *request*'s service-workers mode to "`foreign`".
    4. Set *r*'s initiator to "`fetch`" and destination to "`subresource`".
    5. Add *r* to *requestList*.
    6. Let *responsePromise* be a new promise.
    7. Run the following substeps in parallel:

        - Fetch *r*.
        - To process response for *response*, run these substeps:

1. If *response*'s type is "`error`", or *response*'s status is not an ok status or is `206`, reject *responsePromise* with a `TypeError`.
2. Else if *response*'s header list contains a header named `` `Vary` ``, then:
    1. Let *fieldValues* be the list containing the elements corresponding to the field-values of the Vary header.
    2. For each *fieldValue* of *fieldValues*:
        1. If *fieldValue* matches "`*`", then:
            1. Reject *responsePromise* with a `TypeError`.
            2. Terminate all the ongoing fetches initiated by *requests* with the aborted flag set.
            3. Abort these steps.

- To process response end-of-body for *response*, run these substeps:
    1. If *response*'s aborted flag is set, reject *responsePromise* with an "`AbortError`" `DOMException` and abort these steps.
    2. Resolve *responsePromise* with *response*.

Note: The cache commit is allowed when the response's body is fully received.

8. Add *responsePromise* to *responsePromises*.

5. Let *p* be waiting for all of *responsePromises*.
6. Return the result of transforming *p* with a fulfillment handler that, when called with argument *responses*, performs the following substeps:

    1. Let *operations* be an empty list.
    2. Let *index* be zero.
    3. For each *response* in *responses*:

        1. Let *operation* be a cache batch operation.
        2. Set *operation*'s type to `"put"`.
        3. Set *operation*'s request to *requestList*[*index*].
        4. Set *operation*'s response to *response*.
        5. Append *operation* to *operations*.
        6. Increment *index* by one.

    4. Let *realm* be the context object's relevant realm.
    5. Let *cacheJobPromise* be a new promise.
    6. Run the following substeps in parallel:

        1. Let *errorData* be null.

2. Invoke Batch Cache Operations with *operations*. If this throws an exception, set *errorData* to the exception.
3. Queue a task, on *cacheJobPromise*'s relevant settings object's responsible event loop using the DOM manipulation task source, to perform the following substeps:
    1. If *errorData* is null, resolve *cacheJobPromise* with undefined.
    2. Else, reject *cacheJobPromise* with a new exception with *errorData* and a user agent-defined message, in *realm*.
7. Return *cacheJobPromise*.

5.4.5. `put(request, response)`

put(*`request, response`*) method *must* run these steps:

1. Let *r* be null.
2. If *request* is a `Request` object, then:
    1. Set *r* to *request*'s request.
    2. If *r*'s url's scheme is not one of "`http`" and "`https`", or *r*'s method is not `` `GET` ``, return a promise rejected with a `TypeError`.

3. Else if *request* is a string, then:

    1. Set *r* to the associated request of the result of invoking the initial value of `Request` as constructor with *request* as its argument. If this throws an exception, return a promise rejected with that exception.
    2. If *r*'s url's scheme is not one of "`http`" and "`https`", return a promise rejected with a `TypeError`.

4. If *response*'s associated response's status is `206`, return a promise rejected with a `TypeError`.
5. If *response*'s associated response's header list contains a header named `` `Vary` ``, then:

    1. Let *fieldValues* be the list containing the items corresponding to the Vary header's field-values.
    2. For each *fieldValue* in *fieldValues*:

        1. If *fieldValue* matches "`*`", return a promise rejected with a `TypeError`.

6. If *response* is disturbed or locked, return a promise rejected with a `TypeError`.
7. Let *clonedResponse* be the

result of cloning *response*'s associated response.

8. If *response*'s body is non-null, run these substeps:

    1. Let *dummyStream* be an empty `ReadableStream` object.
    2. Set *response*'s body to a new body whose stream is *dummyStream*.
    3. Let *reader* be the result of getting a reader from *dummyStream*.
    4. Read all bytes from *dummyStream* with *reader*.

9. Let *operations* be an empty list.
10. Let *operation* be a cache batch operation.
11. Set *operation*'s type to `"put"`.
12. Set *operation*'s request to *r*.
13. Set *operation*'s response to *clonedResponse*.
14. Append *operation* to *operations*.
15. Let *realm* be the context object's relevant realm.
16. Let *cacheJobPromise* be a new promise.
17. Run the following substeps in parallel:

    1. Let *errorData* be null.
    2. Invoke Batch Cache Operations with *operations*. If this throws an exception, set

errorData to the exception.

3. Queue a task, on *cacheJobPromise*'s relevant settings object's responsible event loop using the DOM manipulation task source, to perform the following substeps:

    1. If *errorData* is null, resolve *cacheJobPromise* with undefined.
    2. Else, reject *cacheJobPromise* with a new exception with *errorData* and a user agent-defined message, in *realm*.

18. Return *cacheJobPromise*.

5.4.6. `delete(request, options)`

delete(*`request, options`*) method *must* run these steps:

1. Let *r* be null.
2. If *request* is a `Request` object, then:

    1. Set *r* to *request*'s request.
    2. If *r*'s method is not `GET` and *options*.ignoreMethod is false, return a promise resolved with false.

3. Else if *request* is a string, then:

    1. Set *r* to the associated request of the result of invoking the initial value of `Request` as con-

structor with *request* as its argument. If this throws an exception, return a promise rejected with that exception.

4. Let *operations* be an empty list.
5. Let *operation* be a cache batch operation.
6. Set *operation*'s type to "`delete`".
7. Set *operation*'s request to *r*.
8. Set *operation*'s options to *options*.
9. Append *operation* to *operations*.
10. Let *realm* be the context object's relevant realm.
11. Let *cacheJobPromise* be a new promise.
12. Run the following substeps in parallel:

    1. Let *errorData* be null.
    2. Let *requestResponses* be the result of running Batch Cache Operations with *operations*. If this throws an exception, set *errorData* to the exception.
    3. Queue a task, on *cacheJobPromise*'s relevant settings object's responsible event loop using the DOM manipulation task source, to perform the following substeps:

        1. If *errorData* is null, then:

1. If *requestResponses* is not empty, resolve *cacheJobPromise* with true.
2. Else, resolve *cacheJobPromise* with false.

2. Else, reject *cacheJobPromise* with a new exception with *errorData* and a user agent-defined message, in *realm*.

13. Return *cacheJobPromise*.

5.4.7. `keys(request, options)`

keys(*`request, options`*) method *must* run these steps:

1. Let *r* be null.
2. If the optional argument *request* is not omitted, then:

    1. If *request* is a `Request` object, then:

        1. Set *r* to *request*'s request.
        2. If *r*'s method is not `GET` and *options*.ignoreMethod is false, return a promise resolved with an empty array.

    2. Else if *request* is a string, then:

        1. Set *r* to the associated request of the result of invoking the initial value of `Request` as constructor with *request* as its argument. If this throws an

exception, return a promise rejected with that exception.

3. Let *realm* be the context object's relevant realm.
4. Let *promise* be a new promise.
5. Run these substeps in parallel:

    1. Let *requests* be an empty list.
    2. If the optional argument *request* is omitted, then:

        1. For each *requestResponse* of the relevant request response list:

            1. Add *requestResponse*'s request to *requests*.

    3. Else:

        1. Let *requestResponses* be the result of running Query Cache with *r* and *options*.
        2. For each *requestResponse* of *requestResponses*:

            1. Add *requestResponse*'s request to *requests*.

    4. Queue a task, on *promise*'s relevant settings object's responsible event loop using the DOM manipulation task source, to perform the following steps:

        1. Let *requestList* be a list.
        2. For each *request* of *requests*:

1. Add a new `Request` object associated with *request* and a new associated `Headers` object whose guard is "`immutable`" to *requestList*.

3. Resolve *promise* with a frozen array created from *requestList*, in *realm*.

6. Return *promise*.

§ 5.5. `CacheStorage`

```
[SecureContext, Exposed=(Window,Worker)]
interface CacheStorage {
  [NewObject] Promise<any> match(RequestInfo request, optional MultiCacheQueryOptions opti
  [NewObject] Promise<boolean> has(DOMString cacheName);
  [NewObject] Promise<Cache> open(DOMString cacheName);
  [NewObject] Promise<boolean> delete(DOMString cacheName);
  [NewObject] Promise<sequence<DOMString>> keys();
};

dictionary MultiCacheQueryOptions : CacheQueryOptions {
  DOMString cacheName;
};
```

Note: `CacheStorage` interface is designed to largely conform to ECMAScript 6 Map objects but entirely async, and with additional convenience methods. The methods, `clear`, `forEach`, `entries` and `values`, are intentionally excluded from the scope of the first version resorting to the ongoing discussion about the async iteration by TC39.

The user agent *must* create a `CacheStorage`

object when a Window object or a WorkerGlobalScope object is created and associate it with that ***global object***.

A CacheStorage object represents a name to cache map of its associated global object's environment settings object's origin. Multiple separate objects implementing the CacheStorage interface across documents and workers can all be associated with the same name to cache map simultaneously.

5.5.1. match(request, options)

match(*request, options*) method *must* run these steps:

1. If *options*.cacheName is present, then:
    1. Return a new promise *promise* and run the following substeps in parallel:
        1. For each *cacheName* → *cache* of the relevant name to cache map:
            1. If *options*.cacheName matches *cacheName*, then:

1. Resolve *promise* with the result of running the algorithm specified in `match(request, options)` method of `Cache` interface with *request* and *options* (providing *cache* as thisArgument to the `[[Call]]` internal method of `match(request, options)`.)
2. Abort these steps.

2. Resolve *promise* with undefined.

2. Else:

1. Let *promise* be a promise resolved with undefined.
2. For each *cacheName* → *cache* of the relevant name to cache map:

1. Set *promise* to the result of transforming itself with a fulfillment handler that, when called with argument *response*, performs the following substeps:

1. If *response* is not undefined, return *response*.
2. Return the result of running the algorithm specified in `match(request, options)` method of `Cache` interface with *request* and *options* as the arguments (providing *cache* as thisArgument to the `[[Call]]` internal method

of `match(request, options)`.)

3. Return *promise*.

5.5.2. `has(cacheName)`

has(*cacheName*)
method *must*
run these steps:

1. Let *promise* be a new promise.
2. Run the following substeps in parallel:

    1. For each *key* → *value* of the relevant name to cache map:

        1. If *cacheName* matches *key*, resolve *promise* with true and abort these steps.

    2. Resolve *promise* with false.

3. Return *promise*.

5.5.3. `open(cacheName)`

open(*cacheName*)
method *must*
run these steps:

1. Let *promise* be a new promise.
2. Run the following substeps in parallel:

    1. For each *key* → *value* of the relevant name to cache map:

        1. If *cacheName* matches *key*, then:

            1. Resolve *promise* with a new `Cache` object that represents *value*.
            2. Abort these steps.

2. Let *cache* be a new request response list.
3. Set the relevant name to cache map[*cacheName*] to *cache*. If this cache write operation failed due to exceeding the granted quota limit, reject *promise* with a "`QuotaExceededError`" `DOMException` and abort these steps.
4. Resolve *promise* with a new `Cache` object that represents *cache*.

3. Return *promise*.

5.5.4. `delete(cacheName)`

***`delete(`*`cacheName`*`)`** method *must* run these steps:

1. Let *promise* be the result of running the algorithm specified in `has(cacheName)` method with *cacheName*.
2. Return the result of transforming *promise* with a fulfillment handler that, when called with argument *cacheExists*, performs the following substeps:

    1. If *cacheExists* is false, then:

        1. Return false.

    2. Let *cacheJobPromise* be a new promise.
    3. Run the following substeps in

parallel:

1. Remove the relevant name to cache map[*cacheName*].
2. Resolve *cacheJobPromise* with true.

> Note: After this step, the existing DOM objects (i.e. the currently referenced Cache, Request, and Response objects) should remain functional.

4. Return *cacheJobPromise*.

5.5.5. `keys()`

`keys()` method *must* run these steps:

1. Let *promise* be a new promise.
2. Run the following substeps in parallel:

    1. Let *cacheKeys* be the result of getting the keys of the relevant name to cache map.

    > Note: The items in the result ordered set are in the order that their corresponding entry was added to the name to cache map.

    2. Resolve *promise* with *cacheKeys*.

3. Return *promise*.

6. Security Considerations

§ 6.1. Secure Context

Service workers *must* execute in secure contexts. Service worker clients *must* also be secure contexts to register a service worker registration, to get access to the service worker registrations and the service workers, to do messaging with the service workers, and to be manipulated by the service workers.

> Note: This effectively means that service workers and their service worker clients need to be hosted over HTTPS. A user agent can allow `localhost` (see the requirements), `127.0.0.0/8`, and `::1/128` for development purposes. The primary reason for this restriction is to protect users from the risks associated with insecure contexts.

§ 6.2. Content Security Policy

Whenever a user agent invokes the Run Service Worker algorithm with a service worker *serviceWorker*:

- If *serviceWorker*'s script resource

was delivered with a `Content-Security-Policy` HTTP header containing the value *policy*, the user agent *must* enforce *policy* for *serviceWorker*.

- If *serviceWorker*'s script resource was delivered with a `Content-Security-Policy-Report-Only` HTTP header containing the value *policy*, the user agent *must* monitor *policy* for *serviceWorker*.

The primary reason for this restriction is to mitigate a broad class of content injection vulnerabilities, such as cross-site scripting (XSS).

§ 6.3. Origin Relativity

§ 6.3.1. Origin restriction

This section is non-normative.

A service worker executes in the registering service worker client's origin. One of the advanced concerns that major applications would encounter is whether they can be hosted from a CDN. By definition, these are servers in other places, often on other ori-

gins. Therefore, service workers cannot be hosted on CDNs. But they can include resources via importScripts(). The reason for this restriction is that service workers create the opportunity for a bad actor to turn a bad day into a bad eternity.

6.3.2. `importScripts(urls)`

When the ***`importScripts(urls)`*** method is called on a `ServiceWorkerGlobalScope` object, the user agent *must* import scripts into worker global scope, given this `ServiceWorkerGlobalScope` object and *urls*, and with the following steps to perform the fetch given the request *request*:

1. Let *serviceWorker* be *request*'s client's global object's service worker.
2. If *serviceWorker*'s script resource map[*request*'s url] exists, return the entry's value.
3. If *serviceWorker*'s state is not "`parsed`" or "`installing`" return a network error.
4. Let *registration* be *service-*

Worker's containing service worker registration.

5. Set *request*'s service-workers mode to "`none`".
6. Set *request*'s cache mode to "`no-cache`" if any of the following are true:
    - *registration*'s update via cache mode is "`none`".
    - The current global object's force bypass cache for import scripts flag is set.
    - *registration* is stale.
7. Let *response* be the result of fetching *request*.
8. Set *response* to *response*'s unsafe response.
9. If *response*'s cache state is not "`local`", set *registration*'s last update check time to the current time.
10. Extract a MIME type from the *response*'s header list. If this MIME type (ignoring parameters) is not a JavaScript MIME type, return a network error.
11. If *response*'s type is not "`error`", and *response*'s status is an ok status, then:
    1. Set *serviceWorker*'s script resource map[*request*'s url] to *response*.
    2. Set *serviceWorker*'s classic

scripts imported flag.

12. Return *response*.

§ 6.4. Cross-Origin Resources and CORS

This section is non-normative.

Applications tend to cache items that come from a CDN or other origin. It is possible to request many of them directly using `<script>`, `<img>`, `<video>` and `<link>` elements. It would be hugely limiting if this sort of runtime collaboration broke when offline. Similarly, it is possible to fetch many sorts of off-origin resources when appropriate CORS headers are set. Service workers enable this by allowing `Caches` to fetch and cache off-origin items. Some restrictions apply, however. First, unlike same-origin resources which are managed in the `Cache` as `Response` objects whose corresponding responses are basic filtered response, the objects stored are `Response` objects whose corresponding responses are either CORS

filtered responses or opaque filtered responses. They can be passed to `event.respondWith(r)` method in the same manner as the `Response` objects whose corresponding responses are basic filtered responses, but cannot be meaningfully created programmatically. These limitations are necessary to preserve the security invariants of the platform. Allowing `Caches` to store them allows applications to avoid re-architecting in most cases.

§ 6.5. Path restriction

This section is non-normative.

In addition to the origin restriction, service workers are restricted by the path of the service worker script. For example, a service worker script at `https://www.example.com/~bob/sw.js` can be registered for the scope url `https://www.example.com/~bob/` but not for the scope `https://www.example.com/` or `https://www.example.com/~alice/`. This provides

some protection for sites that host multiple-user content in separated directories on the same origin. However, the path restriction is not considered a hard security boundary, as only origins are. Sites are encouraged to use different origins to securely isolate segments of the site if appropriate.

Servers can remove the path restriction by setting a Service-Worker-Allowed header on the service worker script.

§ 6.6. Service worker script request

This section is non-normative.

To further defend against malicious registration of a service worker on a site, this specification requires that:

- The Service-Worker header is present on service worker script requests, and
- Service worker scripts are served with a JavaScript MIME type.

§ 6.7. Implementer Concerns

This section is non-normative.

The implementers are encouraged to note:

- Plug-ins should not load via service workers. As plug-ins may get their security origins from their own urls, the embedding service worker cannot handle it. For this reason, the Handle Fetch algorithm makes the potential-navigation-or-subresource request (whose context is either `<embed>` or `<object>`) immediately fallback to the network without dispatching `fetch` event.
- Some of the legacy networking stack code may need to be carefully audited to understand the ramifications of interactions with service workers.

§ 6.8. Privacy

Service workers introduce new persistent storage features including scope to registration map (for service worker registrations and their service workers), request response list and name to cache map (for caches), and script resource

map (for script resources). In order to protect users from any potential unsanctioned tracking threat, these persistent storages *should* be cleared when users intend to clear them and *should* maintain and interoperate with existing user controls e.g. purging all existing persistent storages.

§ 7. Extensibility

Service Workers specification is extensible from other specifications.

§ 7.1. Define API bound to Service Worker Registration

Specifications *may* define an API tied to a service worker registration by using partial interface definition to the `ServiceWorkerRegistration` interface where it *may* define the specification specific attributes and methods:

```
EXAMPLE 5
partial interface ServiceWorkerRegistration {
  // e.g. define an API namespace
  readonly attribute APISpaceType APISpace;
  // e.g. define a method
  Promise<T> methodName(/* list of arguments */);
};
```

7.2. Define Functional Event

Specifications *may* define a functional event by extending `ExtendableEvent` interface:

```
EXAMPLE 6
// e.g. define FunctionalEvent interface
interface FunctionalEvent : ExtendableEvent {
  // add a functional event's own attributes and methods
};
```

7.3. Define Event Handler

Specifications *may* define an event handler attribute for the corresponding functional event using partial interface definition to the `ServiceWorkerGlobalScope` interface:

```
EXAMPLE 7
partial interface ServiceWorkerGlobalScope {
  attribute EventHandler onfunctionalevent;
};
```

7.4. Firing Functional Events

To request a functional event dispatch to the active worker of a service worker registration, specifications *should* invoke Fire Functional Event.

Appendix A: Algorithms

The following definitions are

the user agent's internal data structures used throughout the specification.

A ***scope to registration map*** is an ordered map where the keys are scope urls, serialized, and the values are service worker registrations.

A ***job*** is an abstraction of one of register, update, and unregister request for a service worker registration.

A job has a ***job type***, which is one of *register*, *update*, and *unregister*.

A job has a ***scope url*** (a URL).

A job has a ***script url*** (a URL).

A job has a ***worker type*** ("`classic`" or "`module`").

A job has an ***update via cache mode***, which is "`imports`", "`all`", or "`none`".

A job has a ***client*** (a service worker client). It is initially null.

A job has a ***job promise*** (a promise). It is initially null.

A job has a ***containing job queue*** (a job queue or null). It is initially null.

A job has a ***list of equivalent jobs*** (a list of jobs). It is initially the empty list.

A job has a ***force bypass cache flag***. It is initially unset.

Two jobs are ***equivalent*** when their job type is the same and:

- For *register* and *update* jobs, both their scope url and the script url are the same.
- For *unregister* jobs, their scope url is the same.

A ***job queue*** is a thread safe queue used to synchronize the set of concurrent jobs. The job queue contains jobs as its items. A job queue is initially empty.

A ***scope to job queue map*** is an ordered map where the keys are scope urls, serialized, and the values are job queues.

§ *Create Job*

Input

jobType, a job type

scopeURL, a URL

scriptURL, a URL

promise, a promise

client, a service worker client

Output

job, a job

1. Let *job* be a new job.

2. Set *job*'s job type to *jobType*.
3. Set *job*'s scope url to *scopeURL*.
4. Set *job*'s script url to *scriptURL*.
5. Set *job*'s job promise to *promise*.
6. Set *job*'s client to *client*.
7. Return *job*.

Schedule Job

Input
job, a job

Output
none

1. Let *jobQueue* be null.
2. Let *jobScope* be *job*'s scope url, serialized.
3. If scope to job queue map[*jobScope*] does not exist, set scope to job queue map[*jobScope*] to a new job queue.
4. Set *jobQueue* to scope to job queue map[*jobScope*].
5. If *jobQueue* is empty, then:
    1. Set *job*'s containing job queue to *jobQueue*, and enqueue *job* to *jobQueue*.
    2. Invoke Run Job with *jobQueue*.
6. Else:
    1. Let *lastJob* be the element at the back of *jobQueue*.
    2. If *job* is equivalent to *lastJob* and *lastJob*'s job promise has not settled, append *job* to *lastJob*'s

list of equivalent jobs.

3. Else, set *job*'s containing job queue to *jobQueue*, and enqueue *job* to *jobQueue*.

§ *Run Job*

Input
jobQueue, a job queue

Output
none

1. Assert: *jobQueue* is not empty.
2. Queue a task to run these steps:

    1. Let *job* be the first item in *jobQueue*.
    2. If *job*'s job type is *register*, run Register with *job* in parallel.
    3. Else if *job*'s job type is *update*, run Update with *job* in parallel.

        Note: For a register job and an update job, the user agent delays queuing a task for running the job until after a `DOMContentLoaded` event has been dispatched to the document that initiated the job.

    4. Else if *job*'s job type is *unregister*, run Unregister with *job* in parallel.

§ *Finish Job*

Input
job, a job

Output
none

1. Let *jobQueue* be *job*'s containing job queue.
2. Assert: the first item in *jobQueue* is *job*.
3. Dequeue from *jobQueue*.
4. If *jobQueue* is not empty, invoke Run Job with *jobQueue*.

§ *Resolve Job Promise*

Input
job, a job
value, any

Output
none

1. If *job*'s client is not null, queue a task, on *job*'s client's responsible event loop using the DOM manipulation task source, to run the following substeps:
    1. Let *convertedValue* be null.
    2. If *job*'s job type is either *register* or *update*, set *convertedValue* to the result of getting the service worker registration object that represents *value* in *job*'s client.
    3. Else, set *convertedValue* to *value*, in *job*'s client's Realm.
    4. Resolve *job*'s job promise with *convertedValue*.
2. For each *equivalentJob* in *job*'s list of equivalent jobs:
    1. If *equivalentJob*'s client is null,

continue to the next iteration of the loop.

2. Queue a task, on *equivalentJob*'s client's responsible event loop using the DOM manipulation task source, to run the following substeps:

    1. Let *convertedValue* be null.
    2. If *equivalentJob*'s job type is either *register* or *update*, set *convertedValue* to the result of getting the service worker registration object that represents *value* in *equivalentJob*'s client.
    3. Else, set *convertedValue* to *value*, in *equivalentJob*'s client's Realm.
    4. Resolve *equivalentJob*'s job promise with *convertedValue*.

§ *Reject Job Promise*

Input
job, a job

errorData, the information necessary to create an exception

Output
none

1. If *job*'s client is not null, queue a task, on *job*'s client's responsible event loop using the DOM manipulation task source, to reject *job*'s job promise with a new exception with *errorData* and a

user agent-defined message, in *job*'s client's Realm.

2. For each *equivalentJob* in *job*'s list of equivalent jobs:

    1. If *equivalentJob*'s client is null, continue.
    2. Queue a task, on *equivalentJob*'s client's responsible event loop using the DOM manipulation task source, to reject *equivalentJob*'s job promise with a new exception with *errorData* and a user agent-defined message, in *equivalentJob*'s client's Realm.

§ *Register*

Input
job, a job

Output
none

1. If the result of running potentially trustworthy origin with the origin of *job*'s script url as the argument is `Not Trusted`, then:

    1. Invoke Reject Job Promise with *job* and "`SecurityError`" `DOMException`.
    2. Invoke Finish Job with *job* and abort these steps.

2. If the origin of *job*'s script url is not *job*'s client's origin, then:

1. Invoke Reject Job Promise with *job* and "`SecurityError`" `DOMException`.
2. Invoke Finish Job with *job* and abort these steps.

3. If the origin of *job*'s scope url is not *job*'s client's origin, then:

    1. Invoke Reject Job Promise with *job* and "`SecurityError`" `DOMException`.
    2. Invoke Finish Job with *job* and abort these steps.

4. Let *registration* be the result of running the Get Registration algorithm passing *job*'s scope url as the argument.
5. If *registration* is not null, then:

    1. Let *newestWorker* be the result of running the Get Newest Worker algorithm passing *registration* as the argument.
    2. If *newestWorker* is not null, *job*'s script url equals *newestWorker*'s script url, and *job*'s update via cache mode's value equals *registration*'s update via cache mode's value, then:

        1. Invoke Resolve Job Promise with *job* and *registration*.

2. Invoke Finish Job with *job* and abort these steps.

6. Else:

    1. Invoke Set Registration algorithm with *job*'s scope url and *job*'s update via cache mode.

7. Invoke Update algorithm passing *job* as the argument.

Update

Input
job, a job

Output
none

1. Let *registration* be the result of running the Get Registration algorithm passing *job*'s scope url as the argument.
2. If *registration* is null, then:

    1. Invoke Reject Job Promise with *job* and `TypeError`.
    2. Invoke Finish Job with *job* and abort these steps.

3. Let *newestWorker* be the result of running Get Newest Worker algorithm passing *registration* as the argument.
4. If *job*'s job type is *update*, and *newestWorker*'s script url does not equal *job*'s script url, then:

    1. Invoke Reject Job Promise with *job* and `TypeError`.

2. Invoke Finish Job with *job* and abort these steps.

5. Let *httpsState* be "none".
6. Let *referrerPolicy* be the empty string.
7. Let *hasUpdatedResources* be false.
8. Let *updatedResourceMap* be an ordered map where the keys are URLs and the values are responses.
9. Switching on *job*'s worker type, run these substeps with the following options:

"classic"
: Fetch a classic worker script given *job*'s serialized script url, *job*'s client, "serviceworker", and the to-be-created environment settings object for this service worker.

"module"
: Fetch a module worker script graph given *job*'s serialized script url, *job*'s client, "serviceworker", "omit", and the to-be-created environment settings object for this service worker.

> Note: This step has two known issues.
>
> First, using the to-be-created environment settings object rather than a concrete environment settings object. This is used due to the unique processing model of service workers compared to the processing model of other web workers. The script fetching algorithms of HTML standard originally designed for other web workers require an environment settings object of the execution environment, but service workers fetch a script separately in the Update algorithm before the script later runs multiple times through the Run Service Worker algorithm.
>
> Second, the fetch a classic worker script algorithm and the fetch a module worker script graph algorithm in HTML take *job*'s client as an argument. *job*'s client is null when passed from the Soft Update algorithm.
>
> These issues are tracked in the issue #1013 of the Service

> Workers GitHub repository. We will address these issues in Service Workers Nightly.

To perform the fetch given *request*, run the following steps:

> 1. Append `Service-Worker`/`script` to *request*'s header list.
>
>   > Note: See the definition of the Service-Worker header in Appendix B: Extended HTTP headers.
>
> 2. Set *request*'s cache mode to "no-cache" if any of the following are true:
>
>   - *registration*'s update via cache mode is not "all".
>   - *job*'s force bypass cache flag is set.
>   - *newestWorker* is not null and *registration* is stale.
>
>   > Note: Even if the cache mode is not set to "no-cache", the user agent obeys Cache-Control header's max-age value in the network layer to determine if it should bypass the browser cache.
>
> 3. Set *request*'s service-workers mode to "none".

4. If the is top-level flag is unset, then return the result of fetching *request*.
5. Set *request*'s redirect mode to "`error`".
6. Fetch *request*, and asynchronously wait to run the remaining steps as part of fetch's process response for the response *response*.
7. Extract a MIME type from the *response*'s header list. If this MIME type (ignoring parameters) is not a JavaScript MIME type, then:

    1. Invoke Reject Job Promise with *job* and "`SecurityError`" `DOMException`.
    2. Asynchronously complete these steps with a network error.

8. Let *serviceWorkerAllowed* be the result of extracting header list values given `` `Service-Worker-Allowed` `` and *response*'s header list.

    > Note: See the definition of the Service-Worker-Allowed header in Appendix B: Extended HTTP headers.

9. Set *httpsState* to *response*'s HTTPS state.

10. Set *referrerPolicy* to the result of parse a referrer policy from a `Referrer-Policy` header of *response*.
11. If *serviceWorkerAllowed* is failure, then:
    1. Asynchronously complete these steps with a network error.
12. Let *scopeURL* be *registration*'s scope url.
13. Let *maxScopeString* be null.
14. If *serviceWorkerAllowed* is null, then:
    1. Let *resolvedScope* be the result of parsing "`./`" using *job*'s script url as the base URL.
    2. Set *maxScopeString* to "`/`", followed by the strings in *resolvedScope*'s path (including empty strings), separated from each other by "`/`".

       Note: The final item in *resolvedScope*'s path will always be an empty string, so *maxScopeString* will have a trailing "`/`".
15. Else:
    1. Let *maxScope* be the result of parsing *serviceWorkerAllowed* using *job*'s script url as the base URL.

2. If *maxScope*'s origin is *job*'s script url's origin, then:

    1. Set *maxScopeString* to "`/`", followed by the strings in *maxScope*'s path (including empty strings), separated from each other by "`/`".

16. Let *scopeString* be "`/`", followed by the strings in *scopeURL*'s path (including empty strings), separated from each other by "`/`".
17. If *maxScopeString* is null or *scopeString* does not start with *maxScopeString*, then:

    1. Invoke Reject Job Promise with *job* and "`SecurityError`" `DOMException`.
    2. Asynchronously complete these steps with a network error.

18. Set *updatedResourceMap*[*request*'s url] to *response*.
19. If *response*'s cache state is not "`local`", set *registration*'s last update check time to the current time.
20. If *newestWorker* is null, or *newestWorker*'s script resource map[*request*'s url]'s body is not byte-for-byte identical with

response's body, set *hasUpdatedResources* to true.

21. Else if *newestWorker*'s classic scripts imported flag is set, then:

    1. For each *url* → *storedResponse* of *newestWorker*'s script resource map:

        1. Let *request* be a new request whose url is *url*, client is *job*'s client, destination is "`script`", parser metadata is "`not parser-inserted`", synchronous flag is set, and whose use-URL-credentials flag is set.
        2. Set *request*'s cache mode to "`no-cache`" if any of the following are true:

            - *registration*'s update via cache mode is "`none`".
            - *job*'s force bypass cache flag is set.
            - *registration* is stale.

        3. Let *fetchedResponse* be the result of fetching *request*.
        4. Set *fetchedResponse* to *fetchedResponse*'s unsafe response.
        5. Set *updatedResourceMap*[*request*'s url] to *fetchedResponse*.
        6. If *fetchedResponse*'s cache state is not "`local`", set *registration*'s last

update check time to the current time.

7. Extract a MIME type from the *fetchedResponse*'s header list. If this MIME type (ignoring parameters) is not a JavaScript MIME type, asynchronously complete these steps with a network error.
8. If *fetchedResponse*'s type is `"error"`, or *fetchedResponse*'s status is not an ok status, asynchronously complete these steps with a network error.
9. If *fetchedResponse*'s body is not byte-for-byte identical with *storedResponse*'s body, set *hasUpdatedResources* to true.

   > Note: The control does not break the loop in this step to continue with all the imported scripts to populate the cache.

22. Return true.

If the algorithm asynchronously completes with null, then:

1. Invoke Reject Job Promise with *job* and `TypeError`.

> Note: This will do nothing if Reject Job Promise was previously invoked with "`SecurityError`" `DOMException`.

2. If *newestWorker* is null, then remove scope to registration map[*scopeURL*, serialized].
3. Invoke Finish Job with *job* and abort these steps.

Else, continue the rest of these steps after the algorithm's asynchronous completion, with *script* being the asynchronous completion value.

10. If *hasUpdatedResources* is false, then:

    1. Invoke Resolve Job Promise with *job* and *registration*.
    2. Invoke Finish Job with *job* and abort these steps.

11. Let *worker* be a new service worker.
12. Set *worker*'s script url to *job*'s script url, *worker*'s script resource to *script*, and *worker*'s type to *job*'s worker type.
13. For each *url* → *response* of *updatedResourceMap*:

    1. Set *worker*'s script resource

map[*url*] to *response*.

14. Set *worker*'s script resource's HTTPS state to *httpsState*.
15. Set *worker*'s script resource's referrer policy to *referrerPolicy*.
16. Let *forceBypassCache* be true if *job*'s force bypass cache flag is set, and false otherwise.
17. Let *runResult* be the result of running the Run Service Worker algorithm with *worker* and *forceBypassCache*.
18. If *runResult* is *failure* or an abrupt completion, then:

    1. Invoke Reject Job Promise with *job* and `TypeError`.
    2. If *newestWorker* is null, then remove scope to registration map[*registration*'s scope url, [serialized].
    3. Invoke Finish Job with *job*.

19. Else, invoke Install algorithm with *job*, *worker*, and *registration* as its arguments.

§ *Soft Update*

The user agent *may* call this as often as it likes to check for updates.

Input
registration, a service worker registration

forceBypassCache, an optional boolean, false by default

> Note: Implementers may use *forceBypassCache* to aid debugging (e.g. invocations from developer tools), and other specifications that extend service workers may also use the flag on their own needs.

Output
None

1. Let *newestWorker* be the result of running Get Newest Worker algorithm passing *registration* as its argument.
2. If *newestWorker* is null, abort these steps.
3. Let *job* be the result of running Create Job with *update*, *registration*'s scope url, *newestWorker*'s script url, null, and null.
4. Set *job*'s worker type to *newestWorker*'s type.
5. Set *job*'s force bypass cache flag if *forceBypassCache* is true.
6. Invoke Schedule Job with *job*.

§ *Install*

Input
job, a job

worker, a service worker

registration, a service worker registration

Output
none

1. Let *installFailed* be false.
2. Let *newestWorker* be the result of running Get Newest Worker algorithm passing *registration* as its argument.
3. Run the Update Registration State algorithm passing *registration*, `"installing"` and *worker* as the arguments.
4. Run the Update Worker State algorithm passing *registration*'s installing worker and `"installing"` as the arguments.
5. > Assert: *job*'s job promise is not null.
6. Invoke Resolve Job Promise with *job* and *registration*.
7. Let *settingsObjects* be all environment settings objects whose origin is *registration*'s scope url's origin.
8. For each *settingsObject* of *settingsObjects*, queue a task on *settingsObject*'s responsible event loop in the DOM manipulation task source to run the following steps:

   > 1. Let *registrationObjects* be

every `ServiceWorkerRegistration` object in *settingsObject*'s realm, whose service worker registration is *registration*.

2. For each *registrationObject* of *registrationObjects*, fire an event on *registrationObject* named `updatefound`.

9. Let *installingWorker* be *registration*'s installing worker.
10. If the result of running the Should Skip Event algorithm with *installingWorker* and "install" is false, then:

    1. Let *forceBypassCache* be true if *job*'s force bypass cache flag is set, and false otherwise.
    2. If the result of running the Run Service Worker algorithm with *installingWorker* and *forceBypassCache* is *failure*, then:

        1. Set *installFailed* to true.

    3. Else:

        1. Queue a task *task* on *installingWorker*'s event loop using the DOM manipulation task source to run the following steps:

            1. Let *e* be the result of creating an event with

`ExtendableEvent`.

2. Initialize *e*'s `type` attribute to `install`.
3. Dispatch *e* at *installingWorker*'s global object.
4. *WaitForAsynchronousExtensions*: Run the following substeps in parallel:

    1. Wait until *e* is not active.
    2. If *e*'s timed out flag is set, or the result of waiting for all of *e*'s extend lifetime promises rejected, set *installFailed* to true.

If *task* is discarded, set *installFailed* to true.

2. Wait for *task* to have executed or been discarded.
3. Wait for the step labeled *WaitForAsynchronousExtensions* to complete.

11. If *installFailed* is true, then:

    1. Run the Update Worker State algorithm passing *registration*'s installing worker and `"redundant"` as the arguments.
    2. Run the Update Registration State algorithm passing *registration*, `"installing"` and null as the arguments.
    3. If *newestWorker* is null, then remove scope to

registration map[*registration*'s scope url, [serialized].

4. Invoke Finish Job with *job* and abort these steps.

12. If *registration*'s waiting worker is not null, then:

    1. Terminate *registration*'s waiting worker.
    2. Run the Update Worker State algorithm passing *registration*'s waiting worker and "`redundant`" as the arguments.

13. Run the Update Registration State algorithm passing *registration*, "`waiting`" and *registration*'s installing worker as the arguments.
14. Run the Update Registration State algorithm passing *registration*, "`installing`" and null as the arguments.
15. Run the Update Worker State algorithm passing *registration*'s waiting worker and "`installed`" as the arguments.
16. Invoke Finish Job with *job*.
17. Wait for all the tasks queued by Update Worker State invoked in this algorithm to have executed.
18. Invoke Try Activate with *registration*.

> Note: If Try Activate does not trigger Activate here, Activate is tried again when the last client controlled by the existing active worker is unloaded, `skipWaiting()` is asynchronously called, or the extend lifetime promises for the existing active worker settle.

§ *Activate*

Input
registration, a service worker registration

Output
None

1. If *registration*'s waiting worker is null, abort these steps.
2. If *registration*'s active worker is not null, then:
    1. Terminate *registration*'s active worker.
    2. Run the Update Worker State algorithm passing *registration*'s active worker and `"redundant"` as the arguments.
3. Run the Update Registration State algorithm passing *registration*, `"active"` and *registration*'s waiting worker as the arguments.
4. Run the Update Registration State algorithm passing *registra-*

tion, "`waiting`" and null as the arguments.

5. Run the Update Worker State algorithm passing *registration*'s active worker and "`activating`" as the arguments.

   > Note: Once an active worker is activating, neither a runtime script error nor a force termination of the active worker prevents the active worker from getting activated.

6. Let *matchedClients* be a list of service worker clients whose creation URL matches *registration*'s scope url.

7. For each *client* of *matchedClients*, queue a task on *client*'s responsible event loop, using the DOM manipulation task source, to run the following substeps:

   > 1. Let *readyPromise* be *client*'s global object's `ServiceWorkerContainer` object's ready promise.
   > 2. If *readyPromise* is pending, resolve *readyPromise* with the the result of getting the service worker registration object that represents *registration* in

readyPromise's relevant settings object.

8. For each *client* of *matchedClients*:

    1. If *client* is a window client, unassociate *client*'s responsible document from its application cache, if it has one.
    2. Else if *client* is a shared worker client, unassociate *client*'s global object from its application cache, if it has one.

    Note: Resources will now use the service worker registration instead of the existing application cache.

9. For each service worker client *client* who is using *registration*:

    1. Set *client*'s active worker to *registration*'s active worker.
    2. Invoke Notify Controller Change algorithm with *client* as the argument.

10. Let *activeWorker* be *registration*'s active worker.
11. If the result of running the Should Skip Event algorithm with *activeWorker* and "activate" is false, then:

    1. If the result of running the Run Service Worker algorithm with

activeWorker is not *failure*, then:

1. Queue a task *task* on *activeWorker*'s event loop using the DOM manipulation task source to run the following steps:

    1. Let *e* be the result of creating an event with `ExtendableEvent`.
    2. Initialize *e*'s `type` attribute to `activate`.
    3. Dispatch *e* at *activeWorker*'s global object.
    4. *WaitForAsynchronousExtensions*: Wait, in parallel, until *e* is not active.

2. Wait for *task* to have executed or been discarded.
3. Wait for the step labeled *WaitForAsynchronousExtensions* to complete.

12. Run the Update Worker State algorithm passing *registration*'s active worker and "`activated`" as the arguments.

§ *Try Activate*

Input
registration, a service worker registration

Output
None

1. If *registration*'s waiting worker is null, return.
2. If *registration*'s active worker is not null and *re-*

gistration's active worker's state is "activating", return.

> Note: If the existing active worker is still in activating state, the activation of the waiting worker is delayed.

3. Invoke Activate with *registration* if either of the following is true:
    - *registration*'s active worker is null.
    - The result of running Service Worker Has No Pending Events with *registration*'s active worker is true, and no service worker client is using *registration* or *registration*'s waiting worker's skip waiting flag is set.

§ *Run Service Worker*

Input
serviceWorker, a service worker

forceBypassCache, an optional boolean, false by default

Output
a Completion or *failure*

> Note: This algorithm blocks until the service worker is running or fails to start.

1. If *serviceWorker* is running, then

return *serviceWorker*'s start status.

2. If *serviceWorker*'s state is `"redundant"`, then return *failure*.
3. > Assert: *serviceWorker*'s start status is null.
4. Let *script* be *serviceWorker*'s script resource.
5. > Assert: *script* is not null.
6. Let *startFailed* be false.
7. Create a separate parallel execution environment (i.e. a separate thread or process or equivalent construct), and run the following steps in that context:

   1. Call the JavaScript Initialize HostDefinedRealm() abstract operation with the following customizations:

      - For the global object, create a new `ServiceWorkerGlobalScope` object. Let *workerGlobalScope* be the created object.
      - Let *realmExecutionContext* be the created JavaScript execution context.

   2. Set *serviceWorker*'s global object to *workerGlobalScope*.

3. Let *workerEventLoop* be a newly created event loop.
4. Let *settingsObject* be a new environment settings object whose algorithms are defined as follows:

   The realm execution context
   Return *realmExecutionContext*.

   The global object
   Return *workerGlobalScope*.

   The responsible event loop
   Return *workerEventLoop*.

   The referrer policy
   Return *workerGlobalScope*'s referrer policy.

   The API URL character encoding
   Return UTF-8.

   The API base URL
   Return *serviceWorker*'s script url.

   The origin
   Return its registering service worker client's origin.

   The creation URL
   Return *workerGlobalScope*'s url.

   The HTTPS state
   Return *workerGlobalScope*'s HTTPS state.
5. Set *workerGlobalScope*'s url to *serviceWorker*'s script url.
6. Set *workerGlobalScope*'s HTTPS state to *serviceWorker*'s script

resource's HTTPS state.

7. Set *workerGlobalScope*'s referrer policy to *serviceWorker*'s script resource's referrer policy.
8. Set *workerGlobalScope*'s type to *serviceWorker*'s type.
9. Set *workerGlobalScope*'s force bypass cache for import scripts flag if *forceBypassCache* is true.
10. Create a new `WorkerLocation` object and associate it with *workerGlobalScope*.
11. If *serviceWorker* is an active worker, and there are any tasks queued in *serviceWorker*'s containing service worker registration's task queues, queue them to *serviceWorker*'s event loop's task queues in the same order using their original task sources.
12. Let *evaluationStatus* be the result of running the classic script *script* if *script* is a classic script, otherwise, the result of running the module script *script* if *script* is a module script.
13. If *evaluationStatus*.[[Value]] is empty, this means the script was not evaluated. Set

startFailed to true and abort these steps.

14. If the script was aborted by the Terminate Service Worker algorithm, set *startFailed* to true and abort these steps.
15. Set *serviceWorker*'s start status to *evaluationStatus*.
16. If *script*'s has ever been evaluated flag is unset, then:

    1. For each *eventType* of *settingsObject*'s global object's associated list of event listeners' event types:

       1. Append *eventType* to *workerGlobalScope*'s associated service worker's set of event types to handle.

       Note: If the global object's associated list of event listeners does not have any event listener added at this moment, the service worker's set of event types to handle remains an empty set. The user agents are encouraged to show a warning that the event listeners must be added on the very first evaluation of the worker script.

    2. Set *script*'s has ever been evaluated flag.

17. Run the responsible event loop specified by *settingsObject* until it is destroyed.
18. Empty *workerGlobalScope*'s list of active timers.

8. Wait for *serviceWorker* to be running, or for *startFailed* to be true.
9. If *startFailed* is true, then return *failure*.
10. Return *serviceWorker*'s start status.

§ *Terminate Service Worker*

Input
serviceWorker, a service worker

Output
None

1. Run the following steps in parallel with *serviceWorker*'s main loop:

    1. Let *serviceWorkerGlobalScope* be *serviceWorker*'s global object.
    2. Set *serviceWorkerGlobalScope*'s closing flag to true.
    3. Remove all the items from *serviceWorker*'s set of extended events.
    4. If there are any tasks, whose task source is either the handle fetch task source or the handle functional event task source, queued in *serviceWorkerGlobalScope*'s event loop's task

queues, queue them to *serviceWorker*'s containing service worker registration's corresponding task queues in the same order using their original task sources, and discard all the tasks (including tasks whose task source is neither the handle fetch task source nor the handle functional event task source) from *serviceWorkerGlobalScope*'s event loop's task queues without processing them.

> Note: This effectively means that the fetch events and the other functional events such as push events are backed up by the registration's task queues while the other tasks including message events are discarded.

5. Abort the script currently running in *serviceWorker*.
6. Set *serviceWorker*'s start status to null.

§ *Handle Fetch*

The Handle Fetch algorithm is the entry point for the fetch handling handed to the service worker context.

Input

request, a request

Output

response, a response

1. Let *handleFetch-Failed* be false.
2. Let *respond-WithEntered* be false.
3. Let *event-Canceled* be false.
4. Let *response* be null.
5. Let *registration* be null.
6. Let *client* be *request*'s client.
7. Let *reservedClient* be *request*'s reserved client.
8. Assert: *request*'s destination is not `"serviceworker"`.
9. If *request* is a potential-navigation-or-subresource request, then:

    1. Return null.

10. Else if *request* is a non-subresource request, then:

    Note: If the non-subresource request is under the scope of a service worker registration, application cache is completely bypassed regardless of whether the non-subresource request uses the service worker registration.

    1. If *reservedClient* is not null and is an environment settings object, then:

1. If *reservedClient* is not a secure context, return null.

2. Else:

    1. If *request*'s url is not a potentially trustworthy URL, return null.

3. If *request* is a navigation request and the navigation triggering it was initiated with a shift+reload or equivalent, return null.

4. Set *registration* to the result of running Match Service Worker Registration algorithm passing *request*'s url as the argument.

5. If *registration* is null or *registration*'s active worker is null, return null.

6. If *request*'s destination is not `"report"`, set *reservedClient*'s active service worker to *registration*'s active worker.

Note: From this point, the service worker client starts to use its active service worker's containing service worker registration.

11. Else if *request* is a subresource request, then:

    1. If *client*'s active service worker is non-null, set *registration* to *client*'s active service worker's containing ser-

vice worker registration.

2. Else, return null.

12. Let *activeWorker* be *registration*'s active worker.
13. Let *shouldSoftUpdate* be true if any of the following are true, and false otherwise:
    - *request* is a non-subresource request.
    - *request* is a subresource request and *registration* is stale.
14. If the result of running the Should Skip Event algorithm with "fetch" and *activeWorker* is true, then:

    1. If *shouldSoftUpdate* is true, then in parallel run the Soft Update algorithm with *registration*.
    2. Return null.

15. If *activeWorker*'s state is "`activating`", wait for *activeWorker*'s state to become "`activated`".
16. If the result of running the Run Service Worker algorithm with *activeWorker* is *failure*, then set *handleFetchFailed* to true.
17. Else queue a task *task* to run the following substeps:

    1. Let *e* be the result of creating an event with `FetchEvent`.

2. Initialize *e*'s `type` attribute to `fetch`.
3. Initialize *e*'s `cancelable` attribute to true.
4. Initialize *e*'s `request` attribute to a new `Request` object associated with *request* and a new associated `Headers` object whose guard is `"immutable"`.
5. If *request* is a non-subresource request and *request*'s destination is not `"report"`, initialize *e*'s `clientId` attribute to the empty string, and to *client*'s id otherwise.
6. Dispatch *e* at *activeWorker*'s global object.
7. Invoke Update Service Worker Extended Events Set with *activeWorker* and *e*.
8. If *e*'s respond-with entered flag is set, set *respondWithEntered* to true.
9. If *e*'s wait to respond flag is set, then:

    1. Wait until *e*'s wait to respond flag is unset.
    2. If *e*'s respond-with error flag is set, set *handleFetchFailed* to true.
    3. Else, set *response* to *e*'s potential response.

10. If *e*'s canceled flag is set, set

eventCanceled to true.

If *task* is discarded, set *handleFetchFailed* to true.

The *task must* use *activeWorker*'s event loop and the handle fetch task source.

18. Wait for *task* to have executed or for *handleFetchFailed* to be true.
19. If *shouldSoftUpdate* is true, then in parallel run the Soft Update algorithm with *registration*.
20. If *respondWithEntered* is false, then return a network error if *eventCanceled* is true and null otherwise.
21. If *handleFetchFailed* is true, then return a network error.
22. Return *response*.

§ *Should Skip Event*

Input

eventName, a string

serviceWorker, a service worker

Output

a boolean

> Note: To avoid unnecessary delays, this specification permits skipping event dispatch when no event listeners for the event have been deterministically added in the service worker's global during the very first script execution.

1. If *serviceWorker*'s set of event types to handle does not contain *eventName*, then the user agent *may* return true.
2. Return false.

§ *Fire Functional Event*

Input

eventName, a string

eventConstructor, an event constructor that extends `ExtendableEvent`

registration, a service worker registration

initialization, optional property initialization for *event*, constructed from *eventConstructor*

postDispatchSteps, optional steps to run on the active worker's event loop, with *dispatchedEvent* set to the instance of *eventConstructor* that was dispatched.

Output

None

1. Assert: *registration*'s active worker is not null.
2. Let *activeWorker* be *registration*'s active worker.
3. If the result of running Should Skip Event with *eventName* and *activeWorker* is true, then:
    1. If *registration* is stale, then in parallel run the Soft Update algorithm with *registration*.
    2. Return.
4. If *activeWorker*'s state is `"activating"`, wait for *activeWorker*'s state to become `"activated"`.
5. If the result of running the Run Service Worker algorithm with *activeWorker* is *failure*, then:
    1. If *registration* is stale, then in parallel run the Soft Update algorithm with *registration*.
    2. Return.
6. Queue a task *task* to run these substeps:
    1. Let *event* be the result of creating an event with *eventConstructor* and the relevant realm of *activeWorker*'s global object.
    2. If *initialization* is not null, then initialize *event* with *initialization*.

3. Dispatch *event* on *activeWorker*'s global object.
4. Invoke Update Service Worker Extended Events Set with *activeWorker* and *event*.
5. If *postDispatchSteps* is not null, then run *postDispatchSteps* passing *event* as *dispatchedEvent*.

The *task must* use *activeWorker*'s event loop and the handle functional event task source.

7. Wait for *task* to have executed or been discarded.
8. If *registration* is stale, then in parallel run the Soft Update algorithm with *registration*.

EXAMPLE 8

To fire an "amazingthing" event (which is of type AmazingThingEvent) on a particular *serviceWorkerRegistration*, and initialize the event object's properties, the prose would be:

1. Fire Functional Event "amazingthing" using AmazingThingEvent on *serviceWorkerRegistration* with the following properties:

   propertyName
   : value

   anotherPropertyName
   : anotherValue

   Then run the following steps with *dispatchedEvent*:

   1. Do whatever you need to with *dispatchedEvent* on the service worker's event loop.

Note that the initialization steps and post-dispatch steps are optional. If they aren't needed, the prose would be:

1. Fire Functional Event "whatever" using ExtendableEvent on *serviceWorkerRegistration*.

§ *Handle Service Worker Client Unload*

The user agent *must* run these steps when a service worker client unloads by unloading or terminating.

Input
client, a service worker client

Output
None

1. Run the following steps atomically.
2. Let *registration* be the service worker registration used by *client*.
3. If *registration* is null, abort these steps.
4. If any other service worker client is using *registration*, abort these steps.
5. If *registration* is unregistered, invoke Try Clear Registration with *registration*.
6. Invoke Try Activate with *registration*.

§ *Handle User Agent Shutdown*

Input
None

Output
None

1. For each *scope* → *registration* of scope to registration map:

    1. If *registration*'s installing

worker *installingWorker* is not null, then:

1. If *registration*'s waiting worker is null and *registration*'s active worker is null, invoke Clear Registration with *registration* and continue to the next iteration of the loop.
2. Else, set *installingWorker* to null.

2. If *registration*'s waiting worker is not null, run the following substep in parallel:

1. Invoke Activate with *registration*.

§ *Update Service Worker Extended Events Set*

Input

worker, a service worker

event, an event

Output

None

1. Assert: *event*'s dispatch flag is unset.
2. For each *item* of *worker*'s set of extended events:

1. If *item* is not active, remove *item* from *worker*'s set of extended events.

3. If *event* is active, append *event* to *worker*'s set of extended events.

§ *Unregister*

Input

job, a job

Output

none

1. If the origin of *job*'s scope url is not *job*'s client's origin, then:

    1. Invoke Reject Job Promise with *job* and "`SecurityError`" `DOMException`.
    2. Invoke Finish Job with *job* and abort these steps.

2. Let *registration* be the result of running Get Registration algorithm passing *job*'s scope url as the argument.
3. If *registration* is null, then:

    1. Invoke Resolve Job Promise with *job* and false.
    2. Invoke Finish Job with *job* and abort these steps.

4. Remove scope to registration map[*job*'s scope url].
5. Invoke Resolve Job Promise with *job* and true.
6. Invoke Try Clear Registration with *registration*.

> Note: If Try Clear Registration does not trigger Clear Registration here, Clear Registration is tried again when the last client using the registration is unloaded or the extend lifetime promises for the registration's service workers settle.

7. Invoke Finish Job with *job*.

§ Set Registration

Input

scope, a URL

updateViaCache, an update via cache mode

Output

registration, a service worker registration

1. Run the following steps atomically.
2. Let *scopeString* be serialized *scope* with the *exclude fragment flag* set.
3. Let *registration* be a new service worker registration whose scope url is set to *scope* and update via cache mode is set to *updateViaCache*.
4. Set scope to registration map[*scopeString*] to *registration*.
5. Return *registration*.

§ Clear Registration

Input
registration, a service worker registration

Output
None

1. Run the following steps atomically.
2. If *registration*'s installing worker is not null, then:
    1. Terminate *registration*'s installing worker.
    2. Run the Update Worker State algorithm passing *registration*'s installing worker and `"redundant"` as the arguments.
    3. Run the Update Registration State algorithm passing *registration*, `"installing"` and null as the arguments.
3. If *registration*'s waiting worker is not null, then:
    1. Terminate *registration*'s waiting worker.
    2. Run the Update Worker State algorithm passing *registration*'s waiting worker and `"redundant"` as the arguments.
    3. Run the Update Registration State algorithm passing *registration*, `"waiting"` and null as the arguments.
4. If *registration*'s active worker is not null, then:
    1. Terminate *registration*'s active

worker.

2. Run the Update Worker State algorithm passing *registration*'s active worker and "`redundant`" as the arguments.
3. Run the Update Registration State algorithm passing *registration*, "`active`" and null as the arguments.

Try Clear Registration

Input
registration, a service worker registration

Output
None

1. Invoke Clear Registration with *registration* if no service worker client is using *registration* and all of the following conditions are true:
    - *registration*'s installing worker is null or the result of running Service Worker Has No Pending Events with *registration*'s installing worker is true.
    - *registration*'s waiting worker is null or the result of running Service Worker Has No Pending Events with *registration*'s waiting worker is true.
    - *registration*'s active worker is null or the result of running Service Worker Has

No Pending Events with *registration*'s active worker is true.

§ *Update Registration State*

Input

registration, a service worker registration

target, a string (one of "`installing`", "`waiting`", and "`active`")

source, a service worker or null

Output

None

1. Let *registrationObjects* be an array containing all the `ServiceWorkerRegistration` objects associated with *registration*.
2. If *target* is "`installing`", then:

    1. Set *registration*'s installing worker to *source*.
    2. For each *registrationObject* in *registrationObjects*:

        1. Queue a task to set the `installing` attribute of *registrationObject* to null if *registration*'s installing worker is null, or the result of getting the service worker object that represents *registration*'s installing worker in *registra-*

tionObject's relevant settings object.

3. Else if *target* is "`waiting`", then:

    1. Set *registration*'s waiting worker to *source*.
    2. For each *registrationObject* in *registrationObjects*:

        1. Queue a task to set the `waiting` attribute of *registrationObject* to null if *registration*'s waiting worker is null, or the result of getting the service worker object that represents *registration*'s waiting worker in *registrationObject*'s relevant settings object.

4. Else if *target* is "`active`", then:

    1. Set *registration*'s active worker to *source*.
    2. For each *registrationObject* in *registrationObjects*:

        1. Queue a task to set the `active` attribute of *registrationObject* to null if *registration*'s active worker is null, or the result of getting the service worker object that represents *registration*'s active worker in *registrationObject*'s relevant settings object.

    The task *must* use *registra-*

tionObject's relevant settings object's responsible event loop and the DOM manipulation task source.

§ *Update Worker State*

Input

worker, a service worker

state, a service worker state

Output

None

1. Set *worker*'s state to *state*.
2. Let *settingsObjects* be all environment settings objects whose origin is *worker*'s script url's origin.
3. For each *settingsObject* of *settingsObjects*, queue a task on *settingsObject*'s responsible event loop in the DOM manipulation task source to run the following steps:

    1. Let *objectMap* be *settingsObject*'s service worker object map.
    2. If *objectMap*[*worker*] does not exist, then abort these steps.
    3. Let *workerObj* be *objectMap*[*worker*].
    4. Set *workerObj*'s `state` to *state*.
    5. Fire an event named `statechange` at *workerObj*.

§ Notify Controller Change

Input
client, a service worker client

Output
None

1. Assert: *client* is not null.
2. If *client* is an environment settings object, queue a task to fire an event named `controllerchange` at the `ServiceWorkerContainer` object that *client* is associated with.

The task *must* use *client*'s responsible event loop and the DOM manipulation task source.

§ Match Service Worker Registration

Input
clientURL, a URL

Output
A service worker registration

1. Run the following steps atomically.
2. Let *clientURLString* be serialized *clientURL*.
3. Let *matchingScopeString* be the empty string.
4. Let *scopeStringSet* be the result of getting the keys from scope to registration map.
5. Set *matchingScopeString* to the longest value in *scopeStringSet*

which the value of *clientURL-String* starts with, if it exists.

> Note: The URL string matching in this step is prefix-based rather than path-structural. E.g. a client URL string with "https://example.com/prefix-of/resource.html" will match a registration for a scope with "https://example.com/prefix". The URL string comparison is safe for the same-origin security as HTTP(S) URLs are always serialized with a trailing slash at the end of the origin part of the URLs.

6. Let *matching-Scope* be null.
7. If *matching-ScopeString* is not the empty string, then:
    1. Set *matching-Scope* to the result of parsing *matching-ScopeString*.
    2. Assert: *matchingScope*'s origin and *clientURL*'s origin are same origin.
8. Return the result of running Get Registration algorithm passing *matchingScope* as the argument.

§ *Get Registration*

Input

scope, a URL

Output
A service worker registration

1. Run the following steps atomically.
2. Let *scopeString* be the empty string.
3. If *scope* is not null, set *scopeString* to serialized *scope* with the *exclude fragment flag* set.
4. For each *key* → *value* of scope to registration map:
   1. If *scopeString* matches *key*, then return *value*.
5. Return null.

§ *Get Newest Worker*

Input
registration, a service worker registration

Output
newestWorker, a service worker

1. Run the following steps atomically.
2. Let *newestWorker* be null.
3. If *registration*'s installing worker is not null, set *newestWorker* to *registration*'s installing worker.
4. Else if *registration*'s waiting worker is not null, set *newestWorker* to *registration*'s waiting worker.
5. Else if *registration*'s active worker is not null, set *newest-*

Worker to *registration*'s active worker.

6. Return *newestWorker*.

Service Worker Has No Pending Events

Input

worker, a service worker

Output

True or false, a boolean

1. For each *event* of *worker*'s set of extended events:
    1. If *event* is active, return false.
2. Return true.

Create Client

Input

client, a service worker client

Output

clientObject, a `Client` object

1. Let *clientObject* be a new `Client` object.
2. Set *clientObject*'s service worker client to *client*.
3. Return *clientObject*.

Create Window Client

Input

client, a service worker client

frameType, a string

visibilityState, a string

focusState, a boolean

ancestorOriginsList, a list

Output

windowClient, a `WindowClient` object

1. Let *windowClient* be a new `WindowClient` object.
2. Set *windowClient*'s service worker client to *client*.
3. Set *windowClient*'s frame type to *frameType*.
4. Set *windowClient*'s visibility state to *visibilityState*.
5. Set *windowClient*'s focus state to *focusState*.
6. Set *windowClient*'s ancestor origins array to a frozen array created from *ancestorOriginsList*.
7. Return *windowClient*.

§ *Get Frame Type*

Input

browsingContext, a browsing context

Output

frameType, a string

1. Return the value by switching on the type of *browsingContext*:

   Nested browsing context
   "nested"

   Auxiliary browsing context
   "auxiliary"

   Otherwise
   "top-level"

§ *Resolve Get Client Promise*

Input

client, a service worker client

promise, a promise

Output

none

1. If *client* is an environment settings object, then:

    1. If *client* is not a secure context, queue a task to reject *promise* with a "`SecurityError`" `DOMException`, on *promise*'s relevant settings object's responsible event loop using the DOM manipulation task source, and abort these steps.

2. Else:

    1. If *client*'s creation URL is not a potentially trustworthy URL, queue a task to reject *promise* with a "`SecurityError`" `DOMException`, on *promise*'s relevant settings object's responsible event loop using the DOM manipulation task source, and abort these steps.

3. If *client* is an environment settings object and is not a window client, then:

    1. Let *clientObject* be the result of

running Create Client algorithm with *client* as the argument.

2. Queue a task to resolve *promise* with *clientObject*, on *promise*'s relevant settings object's responsible event loop using the DOM manipulation task source, and abort these steps.

4. Else:

    1. Let *browsingContext* be null.
    2. If *client* is an environment settings object, set *browsingContext* to *client*'s global object's browsing context.
    3. Else, set *browsingContext* to *client*'s target browsing context.
    4. Queue a task to run the following steps on *browsingContext*'s event loop using the user interaction task source:

        1. Let *frameType* be the result of running Get Frame Type with *browsingContext*.
        2. Let *visibilityState* be *browsingContext*'s active document's `visibilityState` attribute value.
        3. Let *focusState* be the result of running the has focus steps with *browsingContext*'s active docu-

ment as the argument.

4. Let *ancestorOriginsList* be the empty list.
5. If *client* is a window client, set *ancestorOriginsList* to *browsingContext*'s active document's relevant global object's `Location` object's ancestor origins list's associated list.
6. Queue a task to run the following steps on *promise*'s relevant settings object's responsible event loop using the DOM manipulation task source:

    1. If *client*'s discarded flag is set, resolve *promise* with undefined and abort these steps.
    2. Let *windowClient* be the result of running Create Window Client with *client*, *frameType*, *visibilityState*, *focusState*, and *ancestorOriginsList*.
    3. Resolve *promise* with *windowClient*.

§ *Query Cache*

Input

requestQuery, a request

options, a `CacheQueryOptions` object, optional

targetStorage, a request re-

sponse list, optional

Output

resultList, a request response list

1. Let *resultList* be an empty list.
2. Let *storage* be null.
3. If the optional argument *targetStorage* is omitted, set *storage* to the relevant request response list.
4. Else, set *storage* to *targetStorage*.
5. For each *requestResponse* of *storage*:
    1. Let *cachedRequest* be *requestResponse*'s request.
    2. Let *cachedResponse* be *requestResponse*'s response.
    3. If Request Matches Cached Item with *requestQuery*, *cachedRequest*, *cachedResponse*, and *options* returns true, then:
        1. Let *requestCopy* be a copy of *cachedRequest*.
        2. Let *responseCopy* be a copy of *cachedResponse*.
        3. Add *requestCopy*/*responseCopy* to *resultList*.
6. Return *resultList*.

§ *Request Matches Cached Item*

Input

requestQuery, a request

request, a request

response, a response or null, optional, defaulting to null

options, a `CacheQueryOptions` object, optional

Output
a boolean

1. If *options*.`ignoreMethod` is false and *request*'s method is not `` `GET` ``, return false.
2. Let *queryURL* be *requestQuery*'s url.
3. Let *cachedURL* be *request*'s url.
4. If *options*.`ignoreSearch` is true, then:

    1. Set *cachedURL*'s query to the empty string.
    2. Set *queryURL*'s query to the empty string.

5. If *queryURL* does not equal *cachedURL* with the *exclude fragment flag* set, then return false.
6. If *response* is null, *options*.`ignoreVary` is true, or *response*'s header list does not contain `` `Vary` ``, then return true.
7. Let *fieldValues* be the list containing the elements corresponding to the field-values of

the Vary header for the value of the header with name `Vary`.

8. For each *fieldValue* in *fieldValues*:

    1. If *fieldValue* matches "*", or the combined value given *fieldValue* and *request*'s header list does not match the combined value given *fieldValue* and *requestQuery*'s header list, then return false.

9. Return true.

§ Batch Cache Operations

Input
operations, a list of cache batch operation objects

Output
resultList, a request response list

1. Let *cache* be the relevant request response list.
2. Let *backupCache* be a new request response list that is a copy of *cache*.
3. Let *addedItems* be an empty list.
4. Try running the following substeps atomically:

    1. Let *resultList* be an empty list.
    2. For each *operation* in *operations*:

        1. If *operation*'s type matches neither "`delete`" nor "`put`", throw a `TypeError`.

2. If *operation*'s type matches "`delete`" and *operation*'s response is not null, throw a `TypeError`.
3. If the result of running Query Cache with *operation*'s request, *operation*'s options, and *addedItems* is not empty, throw an "`InvalidStateError`" `DOMException`.
4. Let *requestResponses* be an empty list.
5. If *operation*'s type matches "`delete`", then:
    1. Set *requestResponses* to the result of running Query Cache with *operation*'s request and *operation*'s options.
    2. For each *requestResponse* in *requestResponses*:
        1. Remove the item whose value matches *requestResponse* from *cache*.
6. Else if *operation*'s type matches "`put`", then:
    1. If *operation*'s response is null, throw a `TypeError`.
    2. Let *r* be *operation*'s request's associated request.
    3. If *r*'s url's scheme is not one of "`http`" and "`https`",

throw a `TypeError`.

4. If *r*'s method is not `GET`, throw a `TypeError`.
5. If *operation*'s options is not null, throw a `TypeError`.
6. Set *requestResponses* to the result of running Query Cache with *operation*'s request.
7. For each *requestResponse* of *requestResponses*:

    1. Remove the item whose value matches *requestResponse* from *cache*.

8. Append *operation*'s request/*operation*'s response to *cache*.
9. If the cache write operation in the previous two steps failed due to exceeding the granted quota limit, throw a "`QuotaExceededError`" `DOMException`.
10. Append *operation*'s request/*operation*'s response to *addedItems*.

7. Append *operation*'s request/*operation*'s response to *resultList*.

3. Return *resultList*.

5. And then, if an exception was thrown, then:

    1. Remove all the items from the

relevant request response list.

2. For each *requestResponse* of *backupCache*:

    1. Append *requestResponse* to the relevant request response list.

3. Throw the exception.

> Note: When an exception is thrown, the implementation does undo (roll back) any changes made to the cache storage during the batch operation job.

§ Appendix B: Extended HTTP headers

§ Service Worker Script Request

An HTTP request to fetch a service worker's script resource will include the following header:

`Service-Worker`
: Indicates this request is a service worker's script resource request.

> Note: This header helps administrators log the requests and detect threats.

§ Service Worker Script Response

An HTTP response to a service worker's script resource request can include the following header:

`*Service-Worker-Allowed*`
: Indicates the user agent will override the path restriction, which limits the maximum allowed scope url that the script can control, to the given value.

  > Note: The value is a URL. If a relative URL is given, it is parsed against the script's URL.

EXAMPLE 9
Default scope:

```
// Maximum allowed scope defaults to the path the script sits in
// "/js/" in this example
navigator.serviceWorker.register("/js/sw.js").then(() => {
  console.log("Install succeeded with the default scope '/js/'.");
});
```

EXAMPLE 10
Upper path without Service-Worker-Allowed header:

```
// Set the scope to an upper path of the script location
// Response has no Service-Worker-Allowed header
navigator.serviceWorker.register("/js/sw.js", { scope: "/" }).catch(() => {
  console.error("Install failed due to the path restriction violation.");
});
```

EXAMPLE 11
Upper path with Service-Worker-Allowed header:

```
// Set the scope to an upper path of the script location
// Response included "Service-Worker-Allowed : /"
navigator.serviceWorker.register("/js/sw.js", { scope: "/" }).then(() => {

  console.log("Install succeeded as the max allowed scope was overriden to '/'.");
});
```

EXAMPLE 12

A path restriction voliation even with Service-Worker-Allowed header:

```
// Set the scope to an upper path of the script location
// Response included "Service-Worker-Allowed : /foo"
navigator.serviceWorker.register("/foo/bar/sw.js", { scope: "/" }).catch(() => {
  console.error("Install failed as the scope is still out of the overriden maximum allow
});
```

§ Syntax

ABNF for the values of the headers used by the service worker's script resource requests and responses:

```
Service-Worker = %x73.63.72.69.70.74 ; "script", case-sensitive
```

Note: The validation of the Service-Worker-Allowed header's values is done by URL parsing algorithm (in Update algorithm) instead of using ABNF.

§ 8. Acknowledgements

Deep thanks go to Andrew Betts for organizing and hosting a small workshop of like-minded individuals including: Jake Archibald, Jackson Gabbard, Tobie Langel, Robin Berjon, Patrick Lauke, Christian Heilmann. From the clarity of the day's discussions and the use-

cases outlined there, much has become possible. Further thanks to Andrew for raising consciousness about the offline problem. His organization of EdgeConf and inclusion of Offline as a persistent topic there has created many opportunities and connections that have enabled this work to progress.

Anne van Kesteren has generously lent his encyclopedic knowledge of Web Platform arcana and standards development experience throughout the development of the service worker. This specification would be incomplete without his previous work in describing the real-world behavior of URLs, HTTP Fetch, Promises, and DOM. Similarly, this specification would not be possible without Ian Hickson's rigorous Web Worker spec. Much thanks to him.

In no particular order, deep gratitude for design guidance and discussion goes to: Jungkee Song, Alec Flett, David Barrett-Kahn, Aaron Boodman,

Michael Nordman, Tom Ashworth, Kinuko Yasuda, Darin Fisher, Jonas Sicking, Jesús Leganés Combarro, Mark Christian, Dave Hermann, Yehuda Katz, François Remy, Ilya Grigorik, Will Chan, Domenic Denicola, Nikhil Marathe, Yves Lafon, Adam Barth, Greg Simon, Devdatta Akhawe, Dominic Cooney, Jeffrey Yasskin, Joshua Bell, Boris Zbarsky, Matt Falkenhagen, Tobie Langel, Gavin Peters, Ben Kelly, Hiroki Nakagawa, Jake Archibald, Josh Soref, Jinho Bang, Yutaka Hirano, isonmad, Ali Alabbas, Philip Jägenstedt, Mike Pennisi, and Eric Willigers.

Jason Weber, Chris Wilson, Paul Kinlan, Ehsan Akhgari, and Daniel Austin have provided valuable, well-timed feedback on requirements and the standardization process.

The authors would also like to thank Dimitri Glazkov for his scripts and formatting tools which have been essential in the production of

this specification. The authors are also grateful for his considerable guidance.

Thanks also to Vivian Cromwell, Greg Simon, Alex Komoroske, Wonsuk Lee, and Seojin Kim for their considerable professional support.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-

normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

> EXAMPLE 13
>
> This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

> Note, this is an informative note.

§ Conformant Algorithms

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 spe-

cific steps can 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 understand and are not intended to be performant. Implementers are encouraged to optimize.

↥.

.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[csp-3] defines the following terms:

- enforced
- monitored

[DOM] defines the following terms:

- Document
- Event
- EventInit
- EventTarget
- cancelable
- canceled flag
- context object
- creating an event
- dispatch
- dispatch flag
- document
- event
- event listener
- fire an event
- isTrusted
- stop immediate propagation flag
- stop propagation flag
- type

[ECMASCRIPT] defines the following terms:

- detacharraybuffer
- execution context
- initialize-hostdefinedrealm
- map objects
- promise

[FETCH] defines the following terms:

- "report"
- Headers
- ReadableStream
- Request
- RequestInfo
- Response
- aborted flag
- basic filtered response
- body (for response)
- cache mode
- cache state
- cancel
- client
- clone
- closed
- construct a readablestream object
- contains
- cors filtered response
- destination
- disturbed
- empty
- enqueue
- errored
- extract a mime type
- extracting header list values
- fetch
- fetch(input, init)
- filtered response
- get a reader
- guard
- header
- header list (for response)
- http fetch
- https state
- https state value
- initiator
- locked
- method
- name
- navigation request
- network error
- non-subresource request
- ok status
- opaque filtered response
- origin
- parser metadata
- potential-navigation-or-subresource request
- process response
- process response end-of-body
- read a chunk
- read all bytes
- redirect mode
- request (for Request)
- reserved client
- response (for Response)
- service-workers mode
- status
- stream
- subresource request
- synchronous flag
- terminated
- type
- url
- use-url-credentials flag
- value

[FileAPI] defines the following terms:

- blob url
- environment

[HTML] defines the following terms:

- AbstractWorker
- DOMContentLoaded
- DedicatedWorkerGlobalScope
- EventHandler
- Location
- MessageEvent
- MessagePort
- Navigator
- SharedWorkerGlobalScope
- StructuredDeserializeWithTransfer
- StructuredSerializeWithTransfer
- Window
- WindowOrWorkerGlobalScope
- Worker
- WorkerGlobalScope
- WorkerLocation
- WorkerNavigator
- WorkerType
- active document
- active service worker
- ancestor origins list
- api base url
- api url character encoding
- application cache
- auxiliary browsing context
- base url
- browsing context
- classic script
- closing
- creation url
- current global object
- data
- discard a document
- dom manipulation task source
- environment discarding steps
- environment settings object
- event handler
- event handler event type
- event handler idl attribute
- event loop
- exceptions enabled flag
- execution ready flag
- fetch a classic worker script
- fetch a module worker script graph
- focusing steps
- global object (for environment settings object)
- has focus steps
- https state (for environment settings object)
- id
- import scripts into worker global scope
- importScripts(urls)
- in parallel
- incumbent settings object
- is top-level
- list of active timers
- message
- module script
- navigate
- nested browsing context
- opaque origin
- origin (for environment settings object)
- owner set
- perform the fetch
- ports
- queue a microtask
- queue a task
- realm (for global object)
- realm execution context
- referrer policy (for environment settings object)
- relevant global object
- relevant realm
- relevant settings object
- replacement enabled
- responsible document
- responsible event loop
- run a classic script
- run a module script
- run a worker
- same origin
- script
- serialization of an origin
- shared worker
- source
- source browsing context
- target browsing context
- task
- task queues
- task source
- top-level browsing context
- triggered by user activation
- type
- unload a document
- unsafe response
- url
- user interaction task source
- web worker

[INFRA] defines the following terms:

- append (for set)
- ascii case-insensitive
- break
- contain
- continue
- dequeue
- enqueue
- entry
- exist
- for each (for map)
- get the keys
- is not empty
- item
- key
- list
- map
- ordered map
- ordered set
- pair
- queue
- remove (for map)
- set
- struct
- value

[MIMESNIFF] defines the following terms:

- javascript mime type

[page-visibility] defines the following terms:

- VisibilityState
- visibilityState

[promises-guide] defines the following terms:
- a new promise
- a promise rejected with
- a promise resolved with
- transforming
- upon fulfillment
- upon rejection
- waiting for all

[push] defines the following terms:
- push

[referrer-policy] defines the following terms:
- referrer policy

[rfc7230] defines the following terms:
- field-value

[rfc7231] defines the following terms:
- vary

[secure-contexts] defines the following terms:
- potentially trustworthy origin
- potentially trustworthy url
- secure contexts

[STREAMS] defines the following terms:
- chunk

[URL] defines the following terms:
- equal
- fragment
- origin
- path
- query
- scheme
- url
- url parser
- url serializer

[webappsec-referrer-policy] defines the following terms:
- parse a referrer policy from a referrer-policy header

[WebIDL] defines the following terms:
- AbortError
- DOMException
- DOMString
- Exposed
- Global
- InvalidAccessError
- InvalidStateError
- NewObject
- QuotaExceededError
- SameObject
- SecureContext
- SecurityError
- USVString
- boolean
- create a frozen array
- created
- exception
- frozen array type
- message
- object
- partial interface
- present
- throw

§ References

§ Normative References

[CSP-3]
Content Security Policy Level 3
URL: https://www.w3.org/TR/CSP3/

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

[ECMASCRIPT]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[FETCH]
Anne van Kesteren. Fetch Standard. Living Standard. URL:

https://fetch.spec.whatwg.org/

[FileAPI]
Marijn Kruisselbrink; Arun Ranganathan. File API. 31 May 2019. WD. URL: https://www.w3.org/TR/FileAPI/

[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/

[MIMESNIFF]
Gordon P. Hemsley. MIME Sniffing Standard. Living Standard. URL: https://mimesniff.spec.whatwg.org/

[PAGE-VISIBILITY]
Jatinder Mann; Arvind Jain. Page Visibility (Second Edition). 29 October 2013. REC. URL: https://www.w3.org/TR/page-visibility/

[PROMISES-GUIDE]
Domenic Denicola. Writing Promise-Using Specifications. 9 November 2018. TAG Finding. URL: https://www.w3.org/2001/tag/doc/promises-guide

[REFERRER-POLICY]
Jochen Eisinger; Emily Stark. Referrer Policy. 26 January 2017. CR. URL: https://www.w3.org/TR/referrer-policy/

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

[RFC5234]
D. Crocker, Ed.; P. Overell. Augmented BNF for Syntax Specifications: ABNF. January 2008. Internet Standard. URL: https://tools.ietf.org/html/rfc5234

[RFC7230]
R. Fielding, Ed.; J. Reschke, Ed.. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7230.html

[RFC7231]
R. Fielding, Ed.; J. Reschke, Ed.. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7231.html

[SECURE-CONTEXTS]
Mike West. Secure Contexts. 15 September 2016.

CR. URL: https://www.w3.org/TR/secure-contexts/

[STREAMS]
Adam Rice; Domenic Denicola; 吉野剛史 (Takeshi Yoshino). Streams Standard. Living Standard. URL: https://streams.spec.whatwg.org/

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

[WebIDL]
Boris Zbarsky. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

§ Informative References

[UNSANCTIONED-TRACKING]
Unsanctioned Web Tracking. 17 July 2015. Finding of the W3C TAG. URL: https://www.w3.org/2001/tag/doc/unsanctioned-tracking/

§ IDL Index

```
[SecureContext, Exposed=(Window,Worker)]
interface ServiceWorker : EventTarget {
  readonly attribute USVString scriptURL;
  readonly attribute ServiceWorkerState state;
  void postMessage(any message, optional sequence<object> transfer = []);

  // event
  attribute EventHandler onstatechange;
};
ServiceWorker includes AbstractWorker;

enum ServiceWorkerState {
  "installing",
  "installed",
  "activating",
  "activated",
  "redundant"
};

[SecureContext, Exposed=(Window,Worker)]
interface ServiceWorkerRegistration : EventTarget {
  readonly attribute ServiceWorker? installing;
  readonly attribute ServiceWorker? waiting;
  readonly attribute ServiceWorker? active;

  readonly attribute USVString scope;
  readonly attribute ServiceWorkerUpdateViaCache updateViaCache;

  [NewObject] Promise<void> update();
  [NewObject] Promise<boolean> unregister();

  // event
  attribute EventHandler onupdatefound;
};

enum ServiceWorkerUpdateViaCache {
  "imports",
  "all",
  "none"
};

partial interface Navigator {
  [SecureContext, SameObject] readonly attribute ServiceWorkerContainer serviceWorker;
};

partial interface WorkerNavigator {
  [SecureContext, SameObject] readonly attribute ServiceWorkerContainer serviceWorker;
};

[SecureContext, Exposed=(Window,Worker)]
interface ServiceWorkerContainer : EventTarget {
  readonly attribute ServiceWorker? controller;
  readonly attribute Promise<ServiceWorkerRegistration> ready;

  [NewObject] Promise<ServiceWorkerRegistration> register(USVString scriptURL, optional Re

  [NewObject] Promise<any> getRegistration(optional USVString clientURL = "");
  [NewObject] Promise<FrozenArray<ServiceWorkerRegistration>> getRegistrations();

  void startMessages();


  // events
  attribute EventHandler oncontrollerchange;
  attribute EventHandler onmessage; // event.source of message events is ServiceWorker obj
  attribute EventHandler onmessageerror;
};

dictionary RegistrationOptions {
  USVString scope;
  WorkerType type = "classic";
```

```
  ServiceWorkerUpdateViaCache updateViaCache = "imports";
};

[Global=(Worker,ServiceWorker), Exposed=ServiceWorker]
interface ServiceWorkerGlobalScope : WorkerGlobalScope {
  [SameObject] readonly attribute Clients clients;
  [SameObject] readonly attribute ServiceWorkerRegistration registration;

  [NewObject] Promise<void> skipWaiting();

  attribute EventHandler oninstall;
  attribute EventHandler onactivate;
  attribute EventHandler onfetch;

  attribute EventHandler onmessage;
  attribute EventHandler onmessageerror;
};

[Exposed=ServiceWorker]
interface Client {
  readonly attribute USVString url;
  readonly attribute FrameType frameType;
  readonly attribute DOMString id;
  readonly attribute ClientType type;
  void postMessage(any message, optional sequence<object> transfer = []);
};

[Exposed=ServiceWorker]
interface WindowClient : Client {
  readonly attribute VisibilityState visibilityState;
  readonly attribute boolean focused;
  [SameObject] readonly attribute FrozenArray<USVString> ancestorOrigins;
  [NewObject] Promise<WindowClient> focus();
  [NewObject] Promise<WindowClient?> navigate(USVString url);
};

enum FrameType {
  "auxiliary",
  "top-level",
  "nested",
  "none"
};

[Exposed=ServiceWorker]
interface Clients {
  // The objects returned will be new instances every time
  [NewObject] Promise<any> get(DOMString id);
  [NewObject] Promise<FrozenArray<Client>> matchAll(optional ClientQueryOptions options =
  [NewObject] Promise<WindowClient?> openWindow(USVString url);
  [NewObject] Promise<void> claim();
};

dictionary ClientQueryOptions {
  boolean includeUncontrolled = false;
  ClientType type = "window";
};

enum ClientType {
  "window",
  "worker",
  "sharedworker",
  "all"
};

[Constructor(DOMString type, optional ExtendableEventInit eventInitDict = {}), Exposed=Ser
interface ExtendableEvent : Event {
  void waitUntil(Promise<any> f);
};

dictionary ExtendableEventInit : EventInit {
  // Defined for the forward compatibility across the derived events
};
```

```
[Constructor(DOMString type, FetchEventInit eventInitDict), Exposed=ServiceWorker]
interface FetchEvent : ExtendableEvent {
  [SameObject] readonly attribute Request request;
  readonly attribute DOMString clientId;

  void respondWith(Promise<Response> r);
};

dictionary FetchEventInit : ExtendableEventInit {
  required Request request;
  DOMString clientId = "";
};

[Constructor(DOMString type, optional ExtendableMessageEventInit eventInitDict = {}), Expo
interface ExtendableMessageEvent : ExtendableEvent {
  readonly attribute any data;
  readonly attribute USVString origin;
  readonly attribute DOMString lastEventId;
  [SameObject] readonly attribute (Client or ServiceWorker or MessagePort)? source;
  readonly attribute FrozenArray<MessagePort> ports;
};

dictionary ExtendableMessageEventInit : ExtendableEventInit {
  any data = null;
  USVString origin = "";
  DOMString lastEventId = "";
  (Client or ServiceWorker or MessagePort)? source = null;
  sequence<MessagePort> ports = [];
};

partial interface WindowOrWorkerGlobalScope {
  [SecureContext, SameObject] readonly attribute CacheStorage caches;
};

[SecureContext, Exposed=(Window,Worker)]
interface Cache {
  [NewObject] Promise<any> match(RequestInfo request, optional CacheQueryOptions options =
  [NewObject] Promise<FrozenArray<Response>> matchAll(optional RequestInfo request, option
  [NewObject] Promise<void> add(RequestInfo request);
  [NewObject] Promise<void> addAll(sequence<RequestInfo> requests);
  [NewObject] Promise<void> put(RequestInfo request, Response response);
  [NewObject] Promise<boolean> delete(RequestInfo request, optional CacheQueryOptions optio
  [NewObject] Promise<FrozenArray<Request>> keys(optional RequestInfo request, optional Ca
};

dictionary CacheQueryOptions {
  boolean ignoreSearch = false;
  boolean ignoreMethod = false;
  boolean ignoreVary = false;
};

[SecureContext, Exposed=(Window,Worker)]
interface CacheStorage {
  [NewObject] Promise<any> match(RequestInfo request, optional MultiCacheQueryOptions optio
  [NewObject] Promise<boolean> has(DOMString cacheName);
  [NewObject] Promise<Cache> open(DOMString cacheName);
  [NewObject] Promise<boolean> delete(DOMString cacheName);
  [NewObject] Promise<sequence<DOMString>> keys();
};

dictionary MultiCacheQueryOptions : CacheQueryOptions {
  DOMString cacheName;
};
```