Blink Workers [public]

Current architecture, implementation and future project ideas for Blink Workers.
kinuko@chromium.org
Last updated: Jul 2016

Web Workers in Blink

“Web Workers” is a Web platform feature that provides a background Javascript context and allows any scripts to run in the background, usually on a separate thread from the main UI thread.

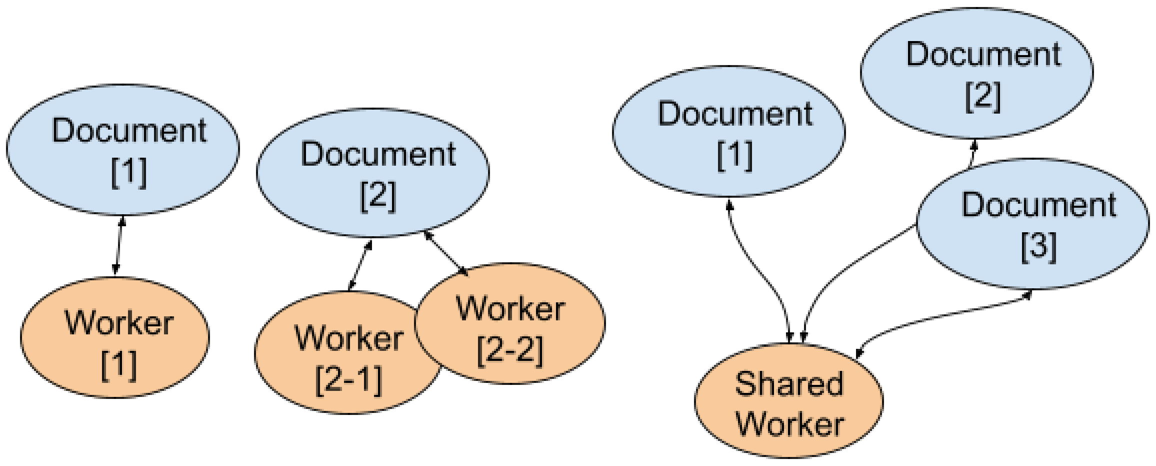
There are several types of Web Workers implemented (and planned to be implemented) in Blink. This document doesn’t provide details for each worker, but as of 2015 Oct Blink has implementation for: **dedicated worker** [spec tutorial], **shared worker** [spec tutorial], **service worker** [spec], and **compositor worker** [design doc]. There are also a few worker-like things (or ***worklet***s) being designed and considered to be implemented, like **isolated worker** [spec] (or Houdini worklet) and **audio worker** [spec].

HTML5 Workers: Dedicated and Shared Workers

Dedicated worker and shared worker are the two types of workers that are considered “standard” workers, and are specified as a part of HTML5 [spec]. Most other workers are modeled after these workers.

Dedicated workers, or just “workers”, are the most regular workers that run a script in the background context that is tied (or “dedicated”) to their creator document. The lifetime of a worker is basically tied to that of creator document.

Shared workers are identified by the URL of the script, and can be shared by multiple documents running in the same origin. When a document calls the constructor of SharedWorker, it may create a new worker or may connect to an existing worker if the worker for the same script URL is already running. The worker context is kept alive while there are more than one associated active document.



(Dedicated) Workers　　**Shared Workers**

Service Workers

Service worker is a powerful new Web platform feature that allows a script to run in the background as a network proxy, i.e. to intercept network requests from a set of associated documents [spec]. A service worker can be shared by multiple documents running in the same origin like shared workers, but its lifetime is not tied to that of any documents. Instead service workers run in an event-driven fashion, and can be started (and then killed) whenever an event is sent to the worker.

Compositor Workers

Compositor worker is yet another emerging Web platform feature that allows a script to do some UI work, e.g. respond to input and update visual effects, but on a different thread from the main UI thread, therefore without being janked by the main thread. More details about compositor workers can be found in the design doc.

“Nested” Workers

HTML5 Workers (and some other workers like service workers) can be actually created and initiated by another worker, and they are called “nested workers”. They are currently not supported in Chromium/Blink as of 2015 Oct (tracking bug: crbug.com/31666). Throughout this document, a javascript context that is responsible for a worker context (e.g. a creator document of the worker) is often simply noted as “parent document” or “associated document”, while it can also be a worker in nested worker cases (which do not happen in the current Blink code base).

Terminology

When one says “worker” in Blink context there could be several possible meanings.

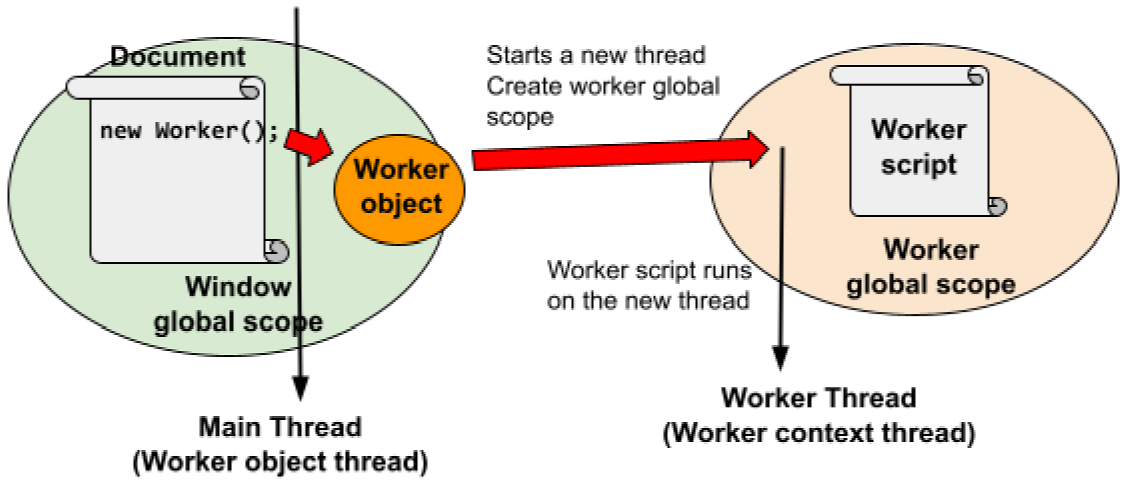
Worker context usually means a background javascript context where worker script runs. Typically a worker context runs on a different thread from the main thread, and its thread is called “worker thread”.

Worker thread refers to a thread where a worker’s javascript context, or “worker context” runs. In most cases a worker thread is dynamically created when a new worker is created, and is shut down when the worker is closed or killed. “Worker thread” also sometimes refers to WorkerThread class in Blink, which usually corresponds to one underlying platform thread (represented by WebThread in Blink) but sometimes not (i.e. in compositor worker case). Worker’s different threading models are explained in the following section.

Worker object usually refers to a javascript “Worker” object, with which its associated document(s) talk(s) to the worker. Note that worker object is instantiated in a worker’s parent context (or associated context), which means that it usually lives on the parent context’s thread, i.e. main thread in most cases.

Worker global scope refers to a global scope for a worker javascript context (while “window” is a global scope for document javascripts). Worker global scope is defined as WorkerGlobalScope interface in the Web Workers specification. APIs that are available to Web workers are exposed in WorkerGlobalScope. Not all APIs available to window scope are available in worker global scope, while some APIs are only available to workers [list of Web Workers API].

Following figure shows conceptual relationships between worker context, worker thread, worker object and worker global scope:


Document
new Worker();
Worker object
Window global scope
Main Thread (Worker object thread)
Starts a new thread Create worker global scope
Worker script
Worker global scope
Worker script runs on the new thread
Worker Thread (Worker context thread)

Life of a Worker

Worker lifetime and its processing model differ across different worker types, but typically a worker is started in a sequence like following:

1. A document creates a new **worker object** to instantiate a new worker.
2. Worker object then loads the worker script.
3. Once script is loaded, a new **worker thread** is created, whose main function then sets up a new **worker global scope** as the global scope for the worker script.
4. Worker's run loop is started on the worker thread and the worker script is evaluated.

When and how workers are shut down is also different across different worker types, but for HTML5 workers worker termination happens when:

- Worker explicitly calls WorkerGlobalScope#close method,
- Parent document calls Worker#terminate method (dedicated worker only),
- All associated documents are closed and become inactive, or
- If document drops a reference to the worker object, the worker context has no pending activity and GC starts and collects the worker object. (This part is currently being discussed in https://crrev.com/1133713008)

All termination sequences except for the first one (WorkerGlobalScope#close case) are triggered from the main thread, and therefore need to stop worker thread execution and join the worker thread too.

Explanation here is abridged and may not be really accurate, more formal lifetime definition can be found in the spec text.

Process and Thread Model

By definition all (or most of) workers basically run on a different thread from the main thread, but different workers employ different process and thread model. Following table summarizes how each worker is implemented and run in Blink:

	Process model	**Thread model**
Dedicated Worker	In-process	Run on its own thread (Worker context : thread = 1:1)
Shared Worker	Out-of-process	Run on its own thread (Worker context : thread = 1:1)
Service Worker	Out-of-process	Run on its own thread (Worker context : thread = 1:1)
Compositor Worker	In-process	Share a per-process single thread within a process (worker context : thread = N:1)
Isolated Worker	In-process	May run on the same thread as document thread

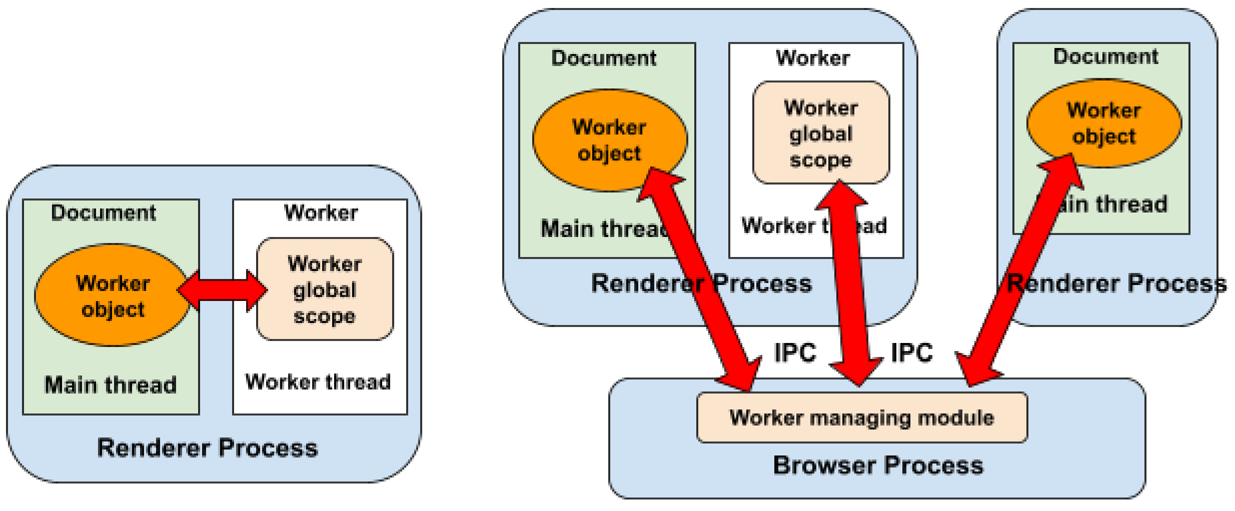
Process Model

Workers can be roughly classified into two types by process model: in-process workers and out-of-process workers.

In-process workers run in the same process as their corresponding document(s), therefore they basically just add "additional threads" to the document(s).

Out-of-process workers may run in a different process from that of their corresponding document(s). Typically when a worker needs to be shared by multiple documents Blink/Chromium implements it as an out-of-process worker, since different documents in different processes may need to talk to a single worker.

Implementation-wise in-process workers can talk to their documents within the renderer process usually by posting tasks between worker thread and main thread, while out-of-process workers needs to talk to their documents over IPC regardless of whether the workers are actually on a different process or on the same process.



In-process worker

Out-of-process worker

Thread Model

There are also differences across different types of workers in threading. Almost all workers run on their own thread (i.e. worker context : worker thread = 1:1 relationship), while there are exceptions: Compositor Worker runs on a per-process singleton thread regardless of the # of contexts, therefore worker context : worker thread relationship is N:1. Also Houdini Worker (Isolated Worker) is planned to run on the same thread as the document thread (so worker and document share the same thread).

Worker Code in Blink

Common worker code that is used by multiple workers and code for “standard” workers (i.e. dedicated and shared workers) are placed under core/workers, while other code that is specific to a particular worker type lives in their own directories, e.g. modules/serviceworker for service workers and modules/compositorworker for compositor workers.

Out-of-process workers (e.g. service worker and shared worker) need to talk to their associated document(s) via IPC, therefore they also need code in chromium. For these workers glue classes that talk to / from chromium are placed under Source/web and public/web directories. (Note: these glue classes are likely to be refactored as blink code repository is now merged into chromium code as of Sep 2015.)

Worker Code Locations

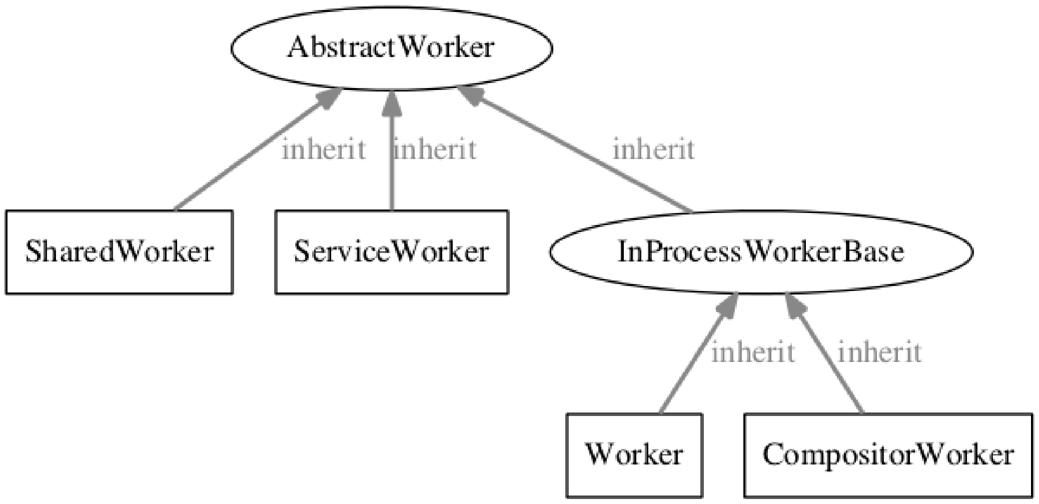
- third_party/WebKit/**Source/core/workers/***
    - Common worker code (e.g. WorkerThread.*, WorkerGlobalScope.*)

    - Dedicated worker code (e.g. DedicatedWorker*)
    - Shared worker code (e.g. SharedWorker*)
- third_party/WebKit/**Source/modules/serviceworkers**/*
    - Service worker code
- third_party/WebKit/**Source/modules/compositorworker**/*
    - Compositor worker code
- third_party/WebKit/**Source/web**/*Worker*
    - Classes that implement public/web/ classes and are used by / talked from chromium (e.g. ServiceWorkerGlobalScopeProxy, WebEmbeddedWorkerImpl, WebSharedWorkerImpl)
    - Classes that need to talk to chromium via public/web/* classes (e.g. WorkerGlobalScopeProxyProviderImpl, ServiceWorkerGlobalScopeClientImpl)
- third_party/WebKit/**public/web**/*Worker*
    - Header files for classes that are are implemented by Blink and used by / talked from chromium (e.g. WebEmbeddedWorker, WebSharedWorker, WebServiceWorkerContextProxy)
    - Header files for classes that are implemented by chromium and used by Blink (e.g. WebServiceWorkerContextClient)
- content/{browser,child,common,renderer}/service_worker/*
    - Chromium code for service worker
- content/{browser,renderer}/shared_worker/*
    - Chromium code for shared worker
- content/child/worker_*
    - Chromium code for handling worker thread. They are used mainly for sending and receiving IPCs to/from worker thread. For example storage API classes that are available on Worker context and need to talk to browser process commonly use these classes.

Important Worker Classes

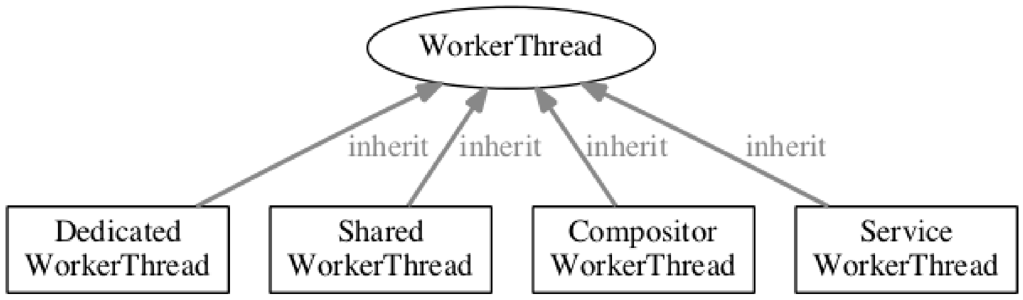
Worker object classes, worker thread classes and worker global scope classes are the three main important class families in common worker code. (What “worker object”, “worker thread” and “worker global scope” refer to is summarized in the Terminology section.)

- **Worker object classes**
    - All worker object classes inherit from AbstractWorker class. This inheritance doesn’t give much in implementation but follows the specification where AbstractWorker interface is defined as a common base interface of HTML5 workers.
    - Worker object class for each worker type is implemented as: Worker for dedicated workers, SharedWorker for shared workers, ServiceWorker for service workers and CompositorWorker for compositor workers, respectively.



Worker object class hierarchy

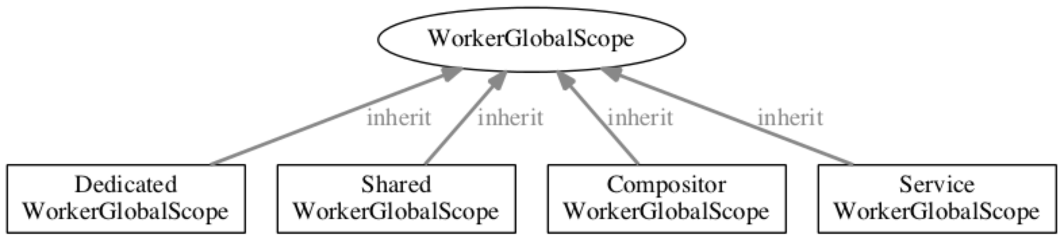
- **Worker thread classes**
  - WorkerThread represents a thread for workers, or a worker thread. An instance of this class usually owns one platform thread (via WebThreadSupportingGC class), which can be accessed via WorkerThread::backingThread() accessor. Each WorkerThread also initializes and uses one v8::Isolate for running worker script completely separately from main thread scripts. (Note: in Blink v8::Isolate is used to instantiate parallel multiple v8 environment for multiple threads, i.e. we have one v8::Isolate for each thread that runs javascript.)
  - WorkerBackingThread represents a thread for workers. A WorkerBackingThread owns a WebThreadSupportingGC and a v8::Isolate. Multiple workers can be attached to one WorkerBackingThread (e.g., CompositorWorkers).
  - WorkerThread can be started by calling WorkerThread::start() and can be terminated by calling WorkerThread::terminate().
  - Each worker type subclasses WorkerThread for its own type, e.g. DedicatedWorkerThread for dedicated workers, SharedWorkerThread for shared workers, ServiceWorkerThread for service workers and CompositorWorkerThread for compositor workers, mainly in order to override WorkerThread::createWorkerGlobalScope method.



Worker thread class hierarchy

- Most subclasses of WorkerThread only overrides createWorkerGlobalScope method, while CompositorWorkerThread also overrides bunch of thread- and v8::isolate-related methods, so that multiple CompositorWorkerThread's can share a single per-process platform thread and v8::Isolate.

- **Worker global scope classes**
    - WorkerGlobalScope is a common base class to represents a worker global scope.
    - Similar to other worker classes, each worker type subclasses WorkerGlobalScope for its own type, e.g. DedicatedWorkerGlobalScope for dedicated workers, SharedWorkerGlobalScope for shared workers, ServiceWorkerGlobalScope for service workers and CompositorWorkerGlobalScope for compositor workers.


WorkerGlobalScope
inherit
inherit
inherit
inherit
Dedicated WorkerGlobalScope
Shared WorkerGlobalScope
Compositor WorkerGlobalScope
Service WorkerGlobalScope

Worker global scope class hierarchy

Adding a New Worker

Currently adding a new worker type to Blink basically requires adding its own Worker object class, WorkerThread subclass and WorkerGlobalScope subclass. It'd be also necessary to add binding code for the new worker global scope, e.g. in bindings/idl.gni and bindings/scripts/v8_utilities.py.

If the new worker is **in-process worker**, it might also need to subclass WorkerMessagingProxy in order to override WorkerMessagingProxy::createWorkerThread() to make it return its own worker thread class.

If the new worker is **out-of-process worker**, starting and shutting down a worker will be likely going to be controlled by IPC come from browser process or from another renderer process that has its associated document. For example, shared workers do this by exposing WebSharedWorker interface to chromium code. The WebSharedWorker class is implemented by WebSharedWorkerImpl and has methods like startWorkerContext and terminateWorkerContext. The former method (startWorkerContext) loads a worker script, creates a new worker thread and eventually calls WorkerThread::start(), while the latter method (terminateWorkerContext) calls WorkerThread::terminate(). Service workers do similar via WebEmbeddedWorker interface.

The new worker also needs to be hooked up into inspector, and the necessary changes also vary depending on its process and thread model. In either way it will need to instantiate WorkerInspectorProxy and hooks to the instance.

Potential Future Projects

There are many areas in which we can improve Blink Worker architecture and code base. A list below shows excerpts from our backlog. Some are being planned and others are just discussed but no active development. Any feedback / more ideas are always welcome!

- [Reliability] **Worker termination should be graceful (Fixed in 2016 Q2)**
    - Currently shutting down a worker from the main thread basically starts with a sudden termination of v8 engine, which means that any code that touches v8 and could be running on the worker thread might start getting null v8 handles. The normal worker shutdown sequence could be probably made more graceful to avoid unexpected null dereference. (Tracking bug: crbug.com/487050)
    - **Fixed in 2016 Q2**: The main thread avoids sudden termination of v8 engine as much as possible. Instead, the main thread posts a delayed task to terminate the v8 engine in case that shutdown sequence does not start on a worker thread in a certain time period.

- [Performance] **Sending data over MessagePort is slow**
    - This is not only about worker code but about MessagePort implementation general. Currently any messaging over MessagePort go across browser process via IPC, regardless of whether the source and destination ports are in the same process or not. This could add noticeable overhead on messaging between document and dedicated workers, where transfer message often could be important. There is also a related bug that we don’t currently support sending transferable ArrayBuffers over MessagePort. (Related bug: crbug.com/334408, design doc)

- [Code health] **Worker is not WebFrameClient; Loader code should be factored out from Frame code**
    - In the current architecture, loading worker scripts needs a frame (or WebFrame) regardless of whether we actually need a frame or not, because loader code is deeply embedded into Frame and FrameLoader code. Because of this architecture currently all out-of-process workers (i.e. shared and service workers) are implemented as frame client (or WebFrameClient), and set up an almost empty WebFrame (called ‘shadow page’) only in order to have a loader for the

worker. This is making the code more complex than necessary, and should be redesigned and cleaned up. (Tracking bug: crbug.com/538751)
    - Currently worker team and loading team are considering working on this in a collaborative way.

- [Code health] **Adding a new worker is not easy, common worker code should be cleaned up**
    - As mentioned above currently adding a new worker requires a lot of copy-and-paste like workflow across multiple directories. This was good when we only had two or a few workers, but it starts to look cumbersome as we're adding more workers. Now that we have better idea about what parts are commonly used we could factor out and clean up common worker code more. Also we could probably add a bit more support in binding generator scripts.

- [Code health, Performance] **More thread isolation: Detaching worker loader/networking code from main thread**
    - Currently worker loader code and most of networking API code is going through the main thread for sending and receiving data to/from the browser process.
    - This does not only make the code complex but also could lead to performance issues: having worker rely on the main thread means that workers could be blocked when the main thread is busy, and also worker code could be a source of jank on the main thread.
    - To give more contexts, in the past most of API implementation for workers used to go through the main thread like loader and WebSocket do today, but we have been fixing this pattern and now most of worker API code directly send/receive IPC to/from browser process without going through the main thread.
    - We should probably do similar refactoring and make the loader and networking code more independent from the threading assumptions.
    - Related tracking bugs are: crbug.com/397403, crbug.com/443374

- [Feature] **Cross-origin messaging support**
    - There seem to be a need for supporting generic cross-origin messaging to/from out-of-process workers (i.e. shared and service workers) so that it becomes possible to run a single worker for caching and providing data to multiple documents. Currently a similar feature that focuses more on handling cross-origin network requests is being discussed and implemented as Foreign Fetch in the service worker context.

- [Reliability] **Better thread safety**
    - Majority of Blink code is single-threaded, but workers require multi-threaded code and it is not very easy to write thread-safe code in Blink, because fundamental classes like String is not thread-safe. Many crash/security bugs due to thread-safety issues have been fixed and we have better utility templates (like

  - threadSafeBind) for cross-thread task posting now, but the current architecture is still error-prone and writing new cross-thread can easily introduce new thread-safety bugs.
  - One way to tighten up this situation would be to annotate all thread-unsafe classes that could be transferred to another thread, and to statically and dynamically verify that they are correctly handled by the cross-thread object handling code. (Design doc)
  - Also now that we are considering using more base/ code in Blink, we should start thinking about how we could make the chromium's Bind and Callback work safely with racy Blink classes. (Discussion thread)

- [Feature] **Support nested workers**
  - As is mentioned in the "Nested" Workers subsection, Blink/Chromium currently doesn't support nested workers, or a worker creating another worker. This one has kept getting lower priority as we didn't have very strong use case for this, but we probably need to periodically review the priority so that we can start re-prioritizing this once it becomes more needed. (Tracking bug: crbug.com/31666)