Show Complete Stack Info in SW Fetch Handler

Attention: Externally visible, non-confidential

Author: soxia@microsoft.com

erica.draud@microsoft.com

(based on work by dgozman@chromium.org)

Status: Inception

Created: 2020-07-03 / **Last Updated**: 2020-07-10

One-page overview

Summary

This explainer explores a new feature for debugging a fetch handler: when a debugger pauses in the fetch handler, a user wants to see the call stack information of the page script that leads to the fetch event (currently this part of stack is missing)

Platforms

All

Team

Songtao (Stan) Xia, soxia@microsoft.com

Erica Draud, erica.draud@microsoft.com

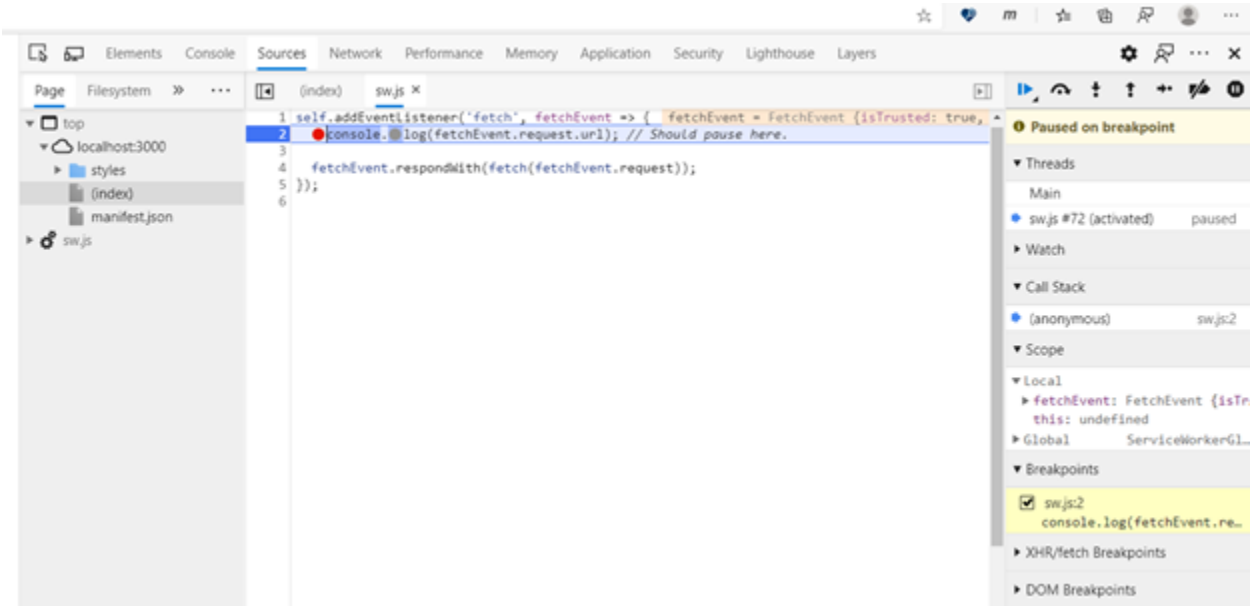
Tracking issue

Cr bug link to be added.

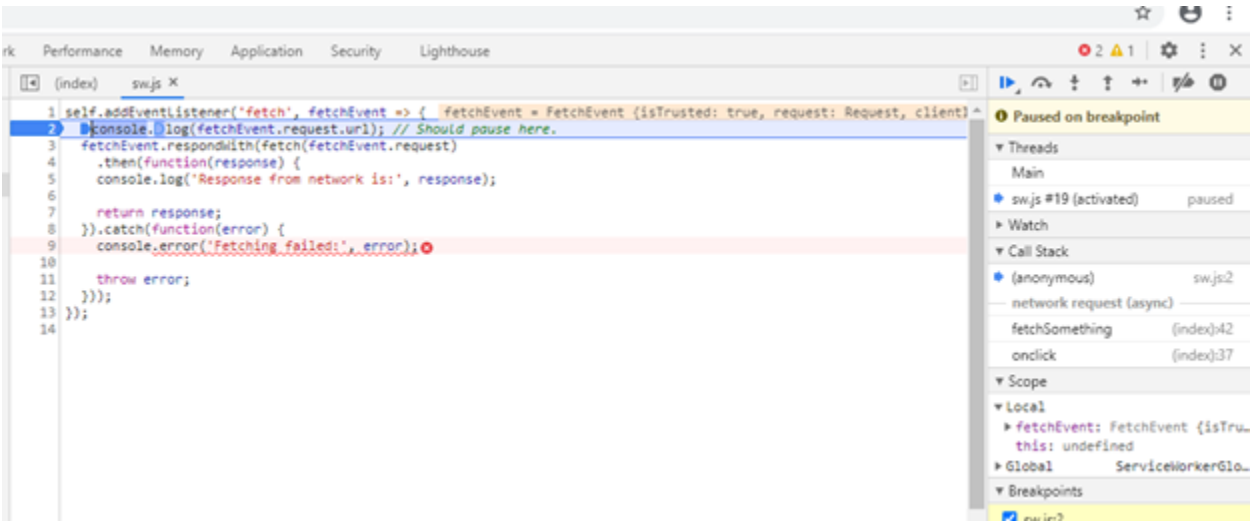
Value proposition

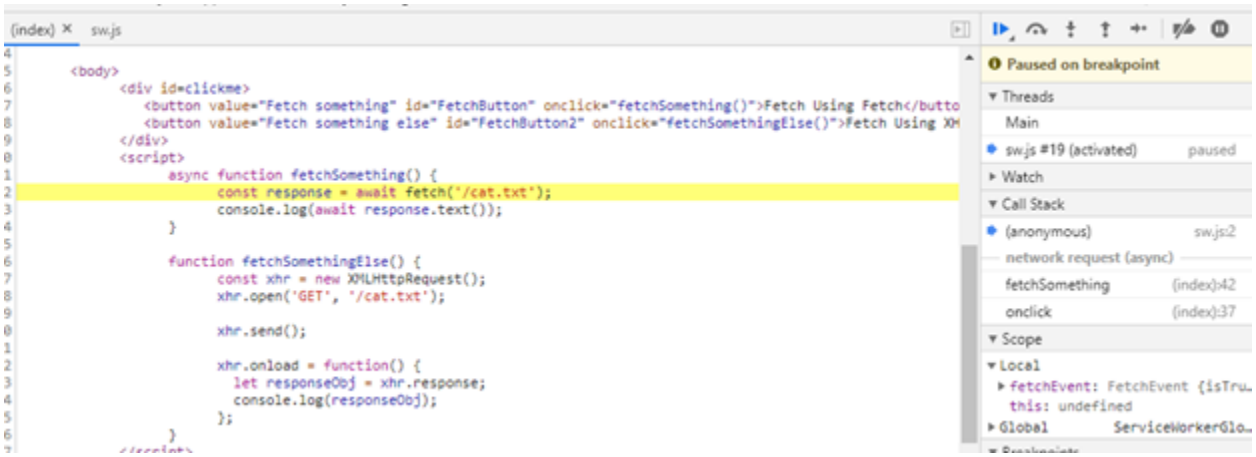
When a page script requests a resource that is controlled by a service worker, the fetch handler of the service worker is called. For a developer to understand how a service worker works, it is most helpful if the developer can step into and out of the fetch handler.

Currently, DevTools allows setting a break point in the fetch handler, but the stack information leading to where the resource is requested in the page script is not available, as shown below. This is due to the asynchronous nature of the handler.



Our proposed change is illustrated by the screenshots below, taken from a prototype implementation. When the debugger pauses inside a fetch handler, we show a combined stack information in the pane to the right. The other screenshot shows that we can move around in the stack frames.





Code affected

- DevTools frontend
- Backend
- CDP change

Chromium WIP PR

We have a prototype PR that implements the idea outlined below.

Signed off by

Name	Write (not) LGTM in this row
yangguo@chromium.org	
shanejc@microsoft.com	LGTM
szuend@chromium.org	
dgozman@chromium.org	LGTM
caseq@chromium.org	LGTM
brandm@microsoft.com	LGTM

Design

Prerequisite: V8 async debugging support

When a V8 debugger executes an async method, it is possible to stitch an existing stack (using a V8StackTraceId) to any stack trace generated by this execution. Current frontend can parse such combined stack traces (for example, when the stack trace is displayed).

Specifically, V8 method ExternalAsyncTaskStarted(stack_trace_id) adds a stack trace to the environment in which the next async method is executed; V8 method ExternalAsyncTaskFinished() removes that stack trace from the environment. Execution/debugging of any async method sandwiched in between these two calls will have an extra segment of an “ancestor” stack trace.

Approach

We extend the network CDP domain to have a method Network.setAttachDebugStack [SC6] [SX7] method. It has a Boolean flag `enabled`. When this flag is set, the DevTools will show a complete stack when debugging a fetch handler. The method is named using the mechanism we used instead of the frontend functionality because this mechanism also allows other future debugging features.

```
setAttachDebugStack(boolean enabled);
```

More Stack Info in Fetch Handler

There are two steps. The first is for the data structures for `ResourceRequest` and `FetchAPIRequest` to carry a stack id. The second step is to stitch the stack id to the V8 context when calling a fetch event handler.

Add stack-id to ResourceRequest

In `InspectorNetworkAgent::PrepareRequest`, if and only if the setAttachDebugHeader is on, we add to the resource request with a field that contains a V8StackTraceId. This field will be consumed later by the backend.

The stack trace id represents the stack trace when the resource is requested in the page script. The trace id is obtained using V8 inspector’s storeCurrentStackTrace method.

Stitch stack id

Currently, when a fetch handler is called, the stack known to the V8 debugger is empty. If the fetch api request has a stack id, we shall stitch the stack id to the empty stack. The implementation should ensure that the stack id, which we placed in a resource request, should be copied into the fetch api request.

This is done in `ServiceWorkerGlobalScope`, where the fetch handler is called. We shall wrap the fetch event handler with a pair of `ExtenalAsyncTaskStarted` and `ExternalAsyncTaskFinished` calls, as described above. The stack visible to the V8 debugger is then an empty stack (which may grow when the fetch handler is executed) with a link to the stack from page script (the stack id we pass to `ExternalAsyncTaskStarted`). The two stacks may belong to two different VMs, which is fine.

The stack trace with a link will be passed to the frontend, which can correctly show such a stack trace already. So change to the frontend is minimal.

Core user stories

When SW Developers step into a fetch handler, they want to see a stack that contains stack frames from the page script.

Rollout plan

Waterfall.

Core principle considerations

Security

It is possible for a malicious site to send a fetch request with a bogus X-Debug-Stack-Trace-Id field in the header. It is possible that such bogus stack id is stitched to a fetch handler stack. But the stack trace from the fetch handler is only reported to CDP. The risk of leaking a stack trace with or without the bogus field is the same. We may also elect to override the incoming field.

Testing plan

We shall implement some web_tests for the backend change, and e2e tests for the frontend.

Followup work

An immediate followup is when stepping over the end of a fetch handler, one should be able to step to the logical next statement, for example, the statement after await fetch() call, or call backs for the fetch result, or XHR onload methods. We have a prototype for this work and an explainer will follow.