

Recording user actions using console utilities APIs

Attention: Externally visible, non-confidential

Author: alexrudenko@chromium.org

Status: Inception | Draft | Accepted | [Done](#)

Created: 2021-04-20 / Last Updated: 2021-04-21

One-page overview

Summary

We change the way user actions are recorded to achieve more reliable recordings.

Platforms

All

Team

alexrudenko@chromium.org janscheffler@chromium.org

Tracking issue

[1200705](#)

Value proposition

Currently, the debugger API is used to record user actions on a page, i.e., via breakpoints set on DOM event listeners. This approach has problems outlined in the document [Reliable Recording](#) (as well as alternative solutions). We propose using additional console utilities APIs and in-page recording to improve stability while allowing for more flexibility.

Code affected

DevTools front-end, backend and V8.

Signed off by

Name	Write (not) LGTM in this row
bmeurer@chromium.org	LGTM
sigurds@chromium.org	LGTM
mathias@chromium.org	LGTM not blocking on the pop-up issue sgtm
caseq@chromium.org	LGTM
...	

Core user stories

As a user I want to record scenarios on a page so that I can replay it later. The recording should be reliable (capture all events consistently).

Design

High-level description

To record user interactions we do the following when a recording is started:

1. We evaluate a script in the target page that sets up event listeners for events that are relevant for recordings.
 - a. We use [Runtime.evaluate](#) and [Page.addScriptToEvaluateOnNewDocument](#) to do that.
 - b. We set the `includeCommandLineAPI` attribute to allow the evaluated scripts to access additional DevTools APIs that are also exposed in the DevTools console.
2. We add a binding using [Runtime.addBinding](#) to allow the injected scripts to send data back to the recorder.
3. We expose two additional functions via the console utilities API (`getAccessibleName(node)` and `getAccessibleRole(node)`) which allow reading the a11y data synchronously. This step is what allows us to stop using event listener breakpoints and reliably record element selectors.

Front-end changes

On the frontend, we create a new TS file called `RecordingClient.ts` that contains the code that will be injected into the inspected pages. The file exports a single function called `setupRecordingClient` that contains the recording logic in-line. It allows us to evaluate the function content using `setupRecordingClient.toString()` and have the advantages of authoring the logic using TypeScript. The implementation when simplified does the following, for example, when recording clicks:

```
const recorderEventListener = (event: Event): void => {  
  const target = event.target;
```

```

window.addStep(
    JSON.stringify({type: event.type, selector: getSelector(target), value:
target.value}));
};
window.addEventListener('click', recorderEventListener, true);

```

Here the addStep is the binding that the front-end adds during recording. The bindings and evaluated scripts are added to a separate isolated world that has access to the page's DOM but the page cannot see it.

CDP changes

Currently, [Page.addScriptToEvaluateOnNewDocument](#) does not include an option to install console utilities APIs in the evaluated script. We propose adding the same parameter as [Runtime.evaluate](#) already has to [Page.addScriptToEvaluateOnNewDocument](#): `includeCommandLineAPI`. Setting this parameter to true would expose all command line APIs to the script.

Console utilities API changes

We add two more functions to the set of command line APIs in [third_party/blink/renderer/core/inspector/thread_debugger.cc](#). The following would be an implementation of these functions:

```

CreateFunctionProperty(
    context, object, "getAccessibleName",
    ThreadDebugger::GetAccessibleNameCallback,
    "function getAccessibleName(node) { [Command Line API] }",
    v8::SideEffectType::kHasNoSideEffect);

void ThreadDebugger::GetAccessibleNameCallback(
    const v8::FunctionCallbackInfo<v8::Value>& info) {
    if (info.Length() < 1)
        return;

    v8::Isolate* isolate = info.GetIsolate();
    v8::Local<v8::Value> value = info[0];

    Node* node = V8Node::ToImplWithTypeCheck(isolate, value);
    if (auto* element = DynamicTo<Element>(node)) {
        v8::Local<v8::String> result =
            v8::String::NewFromUtf8(isolate, element->computedName().Utf8().c_str())
                .ToLocalChecked();
        info.GetReturnValue().Set(result);
    }
}

```

V8 changes

In order to add the `includeCommandLineAPI` flag to [Page.addScriptToEvaluateOnNewDocument](#), we need to export the `CommandLineAPIScope` from V8 and the function that creates the API definition.

The proposed changes are listed in this CL

<https://chromium-review.googlesource.com/c/v8/v8/+2835712>

Problem with Pop-ups

The recorder has to record user actions in any pop-up that is opened by the inspected page. For this, the frontend listens for the target creation event and installs the bindings and scripts into the new target. Since the pop-up targets are not considered to be related to the main target, the auto-attach mechanism that allows pausing the navigation until the setup is complete is not working for pop-up windows. It leads to a possibility that some user actions happen in the pop-up before the recorder is able to set up the listeners. In order to solve this problem, the DevTools frontend has to have access to the browser target in order to set up the auto-attach mechanism. Currently, DevTools does not have access to it and it's not clear how it can be exposed.

We propose to tackle this problem as a follow up since the impact of this problem might be minimal because:

- Not all web sites rely on pop-ups.
- In most cases, DevTools will still set up listeners before any user actions in the pop-up. So far the issue could only be demonstrated by using automation tools like Puppeteer which are capable of triggering events immediately after the navigation is over.

Rollout plan

Waterfall (The Recorder is an experiment at the moment).

Core principle considerations

Speed

The change will not have an impact on overall Chrome performance.

Security

The change is based on existing features such as evaluation of scripts in the inspected page, bindings and exposing additional APIs (a.k.a. Console Utilities APIs). In particular, the additional APIs are exposed only to our scripts, not to page scripts.

Simplicity

We are able to migrate away from using breakpoints by using two additional APIs (for getting accessible roles and name for an element). With these APIs the recording can be done inside the inspected page which allows for complete flexibility when implementing the selector logic.

Accessibility

No UI changes are required for this change so no changes in A11y.

Alternatives considered

Alternatively, we could use `V8RuntimeAgent::runScript` instead of the `ClassicScript` class that `InspectorPageAgent` is currently using. We are not sure if there is a particular reason why `Page.addScriptToEvaluateOnNewDocument` is using `ClassicScripts` so we intend to explore the possibility of changing the implementation to use `V8RuntimeAgent::runScript` in the future.

Testing plan

Waterfall

Followup work

Solving the issue with pop-ups if required.