Better Web Development with WebKit Remote Debugging

Ashutosh Jagdish Sharma

Senior Computer Scientist (Web Platform & Authoring)

Adobe Systems Inc.
I-1A Sector 25A
NOIDA - 201301, India
+91-9958797592

ashutosh@adobe.com

ABSTRACT

The WebKit Remote Debugging API can be used to build custom tools, such as Web Development IDEs to aid in web design and development. In this presentation, source code walkthroughs and demos are presented to highlight the power of this API and show how it can be used in one's own tools - 1) Pausing the debugger when an uncaught exception is thrown, 2) Inspecting the computed style of a node that is visually selected by the user. Developers are encouraged to create their own tools based on this API.

Keywords

WebKit, Debugging, Remote Debugging, Chrome, Chromium, Development Tool, Web Development, Web Design.

1. INTRODUCTION

The WebKit Remote Debugging API is extremely powerful in helping developers inspect, modify and measure metrics on web pages and applications. The API is currently supported by the Chrome and Chromium web browsers and the Web Inspector developer tool in these browsers makes use of it. Third-party tools can also use the API, even when the third-party code lives in a separate process.

Tutorials and documentation available about the API have been scarce. The examples in this presentation should help remedy this by demonstrating how the API can be used in one's own tools.

2. REMOTE DEBUGGING PROTOCOL

The WebKit Remote Debugging Protocol is based on the JSON-RPC 2.0 specification. With remote debugging enabled, the browser, that displays the page being debugged, acts as a server and allows a client (such as a web development IDE) to interact with the page and debug it. The client sends asynchronous requests that the server responds to over a websocket. The protocol is specified by Inspector.json [1], a file in WebKit's source code.

The entire API surface is divided into several categories (called *domains*). Each domain contains [2]:

- *commands* to allow clients to send requests to the browser:

  Example: *DOM.querySelectorAll* is a command that requests the set of nodes that match a given selector

- *events* that are sent from the server to the client:

  Example: *DOM.childNodeRemoved* is an event dispatched when a child node is removed from its parent

In addition to being used for asynchronous notifications, events are also used as responses to some commands (e.g. when requested, the child nodes for a given parent node are sent back as events).

The following domains in the WebKit Remote Debugging Protocol version 1.0 are supported by the current versions of Chrome/Chromium (version 17):

- *CSS* - CSS read/write operations
- *Console* - Interaction with the JavaScript console
- *DOM* - DOM read/write operations
- *DOMDebugger* - Breakpoints on DOM events and operations
- *Debugger* - JavaScript debugging capabilities
- *Network* - Tracking network activities of the page
- *Page* - Actions and events related to the inspected page
- *Runtime* - JavaScript runtime
- *Timeline* - Instrumentation records for the page run-time

In addition to these, there are several domains (*ApplicationCache*, *DOMStorage*, *Database*, *FileSystem*, *IndexedDB*, *Inspector*, *Memory*, *Profiler*, *Worker*) that are marked as *hidden* in the protocol and do not show up in Google's official online documentation [3]. They are not guaranteed to be backwards-compatible, and are internally used by Chrome's Web Inspector developer tool. Some visible domains also have specific *commands* and *events* that are marked as *hidden*.

The network-based WebKit Remote Debugging Protocol also enables tools that use remote debugging API to run on an entirely different machine from that of the browser being debugged. This can be very useful for mobile devices that have a limited screen area. However, mobile implementations of WebKit do not currently support the protocol, but that's likely to change in the future.

3. DEVELOPMENT SETUP

3.1 Browser Setup

A stable build of Chrome or Chromium can be used to access the WebKit Remote Debugging functionality. A *Dev Channel* build of Chrome [4] or a nightly build of Chromium [5] can also be used. Chrome/Chromium acts as a server responding to remote debugging clients over websockets. To launch Chrome in this mode, an additional argument needs to be specified when invoking it:

{Path to Chrome} --remote-debugging-port=9222

This causes Chrome to listen on port 9222 (on localhost) for incoming connections from remote debugging clients [6].

3.2 Connecting to the Browser

With the browser listening for incoming connections, one can navigate to http://localhost:9222/ to see a list of pages that can be remotely inspected and debugged. This is an HTML-rendered list of pages currently open in the browser. To lookup the websocket URL to debug a specific page, one can navigate to http://localhost:9222/json. The entry for an open webpage here has the following properties:

- devtoolsFrontendUrl
- faviconUrl
- thumbnailUrl
- title
- url
- webSocketDebuggerUrl

Of interest to us are *url* and *webSocketDebuggerUrl*, which specify the URL of the web page and that for the websocket for inspecting/debugging the page remotely, respectively. Do note that remote debugging connections are not available while the browser's Web Inspector is open on a given page.

Instead of navigating to http://localhost:9222/json in the browser, one can also fetch that URL over HTTP in one's own standalone application (such as a web development IDE) and fetch details about the various inspectable web pages.

4. CODE WALKTHROUGHS

For our demos, we'll have a remote debugging client running inside the browser itself, to minimize the required setup. Do note that one can implement a client in a separate process/application as well – e.g. to integrate debugging functionality in one's web development IDE.

For our demos, one can navigate to http://localhost:9222/json and then invoke a bookmarklet [7] for each of the code walkthroughs that follow. A bookmarklet is saved as a bookmark (e.g. in the Bookmarks toolbar in the browser) and essentially runs a JavaScript snippet in the context of the current page. To invoke its functionality, one needs to launch the bookmark after navigating to the URL http://localhost:9222/json. The bookmarklet for each demo has the following code, with *{path-to-javascript-file.js}* replaced with the corresponding URL:

Snippet 1. Code for remote debugging client bookmarklets

```
javascript:(function(){
   function loadScript(scriptURL) {
      var scriptElem = document.createElement("SCRIPT");
      scriptElem.setAttribute("language", "JavaScript");
      scriptElem.setAttribute("src", scriptURL);
      document.body.appendChild(scriptElem);
   }
   loadScript("{path-to-javascript-file.js}");
})()
```

In the code walkthroughs that follow, each bookmarklet iterates over the various entries at http://localhost:9222/json (in the *init* function), transforming each into a clickable HTML element, so that the functionality can be invoked on a specific page. (In the case of a standalone remote debugging client, one can use the URL specified by the *webSocketDebuggerUrl* property for inspecting/debugging a page.) While iterating, entries for the page http://localhost:9222/json itself and any Web Inspector windows that are currently open in the browser are ignored.

The tools that follow have source code that corresponds to and works with builds of Chrome or Chromium that support version 1.0 of the WebKit Remote Debugging Protocol. Chromium Build 127895 [8] is used for the live demos.

Each of the tools that follow has a helper class (*Debugger*) that manages the connection with the remote debugging server. This class utilizes jQuery's Deferred functionality [9] to manage and chain asynchronous callbacks.

4.1 Catching Uncaught JavaScript Exceptions

This tool will enable the user to receive alerts when uncaught exceptions are thrown by a target page. One will also be able to obtain the call stack for when the exception is thrown.

One will need to add a new bookmarklet that has the code from Snippet 1 with *{path-to-javascript-file.js}* replaced with the URL *http://marple.host.adobe.com/webkit/demo/exceptions.js* [10].

After navigating to http://localhost:9222/json and invoking the bookmarklet, one can select a target page by clicking on its entry. This calls *process()* which connects to the target page's *websocketDebuggerUrl*, and enables the JavaScript Debugger with the *Debugger.enable* command in the *Debugger* domain. When the Debugger is enabled, it is instructed to pause on uncaught exceptions with the *Debugger.setPauseOnExceptions* method:

Snippet 2. Pausing on uncaught exceptions

```
function process(webSocketDebuggerUrl, pageUrl) {
   var dbg = Debugger.getDebugger(webSocketDebuggerUrl);
   dbg.connect().done(function() {
      dbg.sendCommand("Debugger.enable").done(function() {
         dbg.sendCommand("Debugger.setPauseOnExceptions",
            { state: "uncaught" } );
      });
   });
}
```

When an exception is thrown in the target page's JavaScript and is not caught therein, a *Debugger.paused* event is dispatched by the browser. This is received by the client which then extracts information about the exception and displays the callstack at that time (Snippet 3). In the client, the event is received as a message on the debugger websocket – the JSON data packet will have a *method* property with the value *Debugger.paused*.

In order to avoid completely halting the target page, we resume the Debugger with the *Debugger.resume* command after extracting appropriate information.

To try the bookmarklet out, one can use the sample web page at http://marple.host.adobe.com/webkit/demo/exception.html [11]. This page has a button which, when clicked, invokes JavaScript code that refers to an undefined variable, thereby causing an exception to be thrown.

Snippet 3. Obtaining information about an uncaught exception

```
if(json.params.reason === "exception") {
   var errorName = json.params.data.className;
   var callFrames = json.params.callFrames;
   var callStack = "";
   for(var i = callFrames.length - 1; i >= 0; i--) {
      if(callStack !== "")
         callStack += " -> ";
      callStack += callFrames[i].functionName + "()";
   }
   alert("Exception: " + errorName + "\n" + "Callstack: " +
      callStack);
   self.sendCommand("Debugger.resume");
}
```

4.2 Inspecting the Computed Style for a Visually-Selected Node

This tool will enable a user to visually select a node on the target page and then inspect its computed style.

One will need to add a new bookmarklet that has the code from Snippet 1 with *{path-to-javascript-file.js}* replaced with the URL *http://marple.host.adobe.com/webkit/demo/computedStyle.js* [12].

After navigating to http://localhost:9222/json and invoking the bookmarklet, one can select a target page by clicking on its entry. This calls *process()* which connects to the target page's *websocketDebuggerUrl*, and enables events in the *Inspector* domain with the *Inspector.enable* command. The browser is then instructed to enter a mode where the user can move the mouse cursor over elements to highlight them and see their details, and then select one by clicking on it. This is achieved with the *DOM.setInspectModeEnabled* command in the *DOM* domain. When issuing this command, one can control the highlighting that the browser will use on elements. When the user selects a node on the target page, the remote debugging client receives an *Inspector.inspect* event with a reference to a runtime object corresponding to the node. An additional command *DOM.getDocument* (at the beginning of the Snippet 4) is required to prepare the DOM to be able to resolve this runtime object to its actual DOM node.

Snippet 4. Visually selecting an element on the target page

```
dbg.sendCommand("DOM.getDocument")
.done(function(response) {
   dbg.sendCommand("Inspector.enable")
   .done(function(response) {
      var config = {
         showInfo: true,
         contentColor: { r: 255, g: 0, b: 0, a: 0.5 },
         paddingColor: { r: 255, g: 204, b: 153, a: 0.5 },
         marginColor: { r: 255, g: 255, b: 204, a: 0.5 }
      };
      dbg.sendCommand("DOM.setInspectModeEnabled", {
         enabled: true, highlightConfig: config });
   });
});
```

The resolution of the runtime object to a DOM node is achieved with the *DOM.requestNode* command (Snippet 5).

Snippet 5. Handler for the *Inspect.inspect* event

```
self.sendCommand("DOM.requestNode", { objectId: objectId })
.done(function(response) {
   var nodeId = response.result.nodeId;
   self.sendCommand("CSS.getComputedStyleForNode", {
      nodeId: nodeId }).done(function(response) {
         var result = response.result.computedStyle;
         var computedStyle = {};
            for(var i = 0; i < result.length; i++) {
               var s = result[i];
               computedStyle[s["name"]] = s["value"];
            }
         alert("margin-bottom: " + computedStyle["margin-
bottom"]);
      });
});
```

Once the DOM node is resolved, the command *CSS.getComputedStyleForNode* is issued to fetch information about computed style of the selected node. Similarly, *CSS.getInlineStylesForNode* can be used to fetch information about the inline styles of the selected node.

5. CONCLUSION

In this presentation, we have only scratched the surface of the WebKit Remote Debugging API. It has a large set of commands and events that define a very clean interface with the browser. This API surface enables the Web Inspector (Dev Tools Frontend) in the Chrome browser to be implemented using the remote debugging API.

The support for the WebSocket API in modern browsers makes it possible and easy to create tools that use this powerful API and run within the browser itself, without requiring one to create complex native applications for this purpose.

Chrome also exposes the remote debugging protocol to browser extensions via the chrome.debugger extension API [13].

6. OTHER BROWSERS

Similar to WebKit's Web Inspector, Firebug [14] is a popular Firefox extension that provides similar functionality and is itself extensible. Crossfire [15] is a Firebug extension which has a similar JSON protocol to allow remote debugging clients to connect to Firebug.

7. ACKNOWLEDGMENTS

My thanks to the WebKit team that developed the WebKit Remote Debugging Protocol, and to Narciso Jaramillo (@rictus on Twitter) who wrote the *Debugger* helper class that manages the remote debugging connections.

8. REFERENCES

[1] JSON schema for the remote debugging protocol. http://trac.webkit.org/browser/trunk/Source/WebCore/inspector/Inspector.json

[2] Chrome Developer Tools: Remote Debugging. http://code.google.com/chrome/devtools/docs/remote-debugging.html

[3] Remote Debugging Protocol 1.0

http://code.google.com/chrome/devtools/docs/protocol/1.0/index.html

[4] Chrome's *Dev Channel* builds

http://www.chromium.org/getting-involved/dev-channel

[5] Nightly Chromium builds

http://commondatastorage.googleapis.com/chromium-browser-continuous/index.html?path=Mac/

http://commondatastorage.googleapis.com/chromium-browser-continuous/index.html?path=Win/

http://commondatastorage.googleapis.com/chromium-browser-continuous/index.html?path=Linux/

[6] WebKit Remote Debugging (WebKit Blog). http://www.webkit.org/blog/1620/webkit-remote-debugging/

[7] Bookmarklet

http://en.wikipedia.org/wiki/Bookmarklet

[8] Chromium Build 127895 used for the live demos

http://commondatastorage.googleapis.com/chromium-browser-continuous/Mac/127895/chrome-mac.zip

http://commondatastorage.googleapis.com/chromium-browser-continuous/Win/127895/chrome-win32.zip

http://commondatastorage.googleapis.com/chromium-browser-continuous/Linux/127895/chrome-linux.zip

[9] jQuery's Deferred Object

http://api.jquery.com/category/deferred-object/

[10] Source code for the tool to alert the user on uncaught JavaScript exceptions

http://marple.host.adobe.com/webkit/demo/exceptions.js

[11] Sample web page which has code that throws an uncaught exception http://marple.host.adobe.com/webkit/demo/exception.html

[12] Source code for the tool to inspect the computed style of a visually-selected node

http://marple.host.adobe.com/webkit/demo/computedStyle.js

[13] Chrome extension API to expose the remote debugging protocol to browser extensions

http://code.google.com/chrome/extensions/trunk/debugger.html

[14] Firebug

http://getfirebug.com/whatisfirebug

[15] Crossfire

http://getfirebug.com/wiki/index.php/Crossfire