

```
1 <html><head><meta name="viewport" content="width=device-width"><title>jar:file:///C:/Program%20Files/Mozilla%20Firefox/browser/omni.ja!/chrome/devt
2 * License, v. 2.0. If a copy of the MPL was not distributed with this
3 * file, You can obtain one at http://mozilla.org/MPL/2.0/. */
4
5 /* global clearConsoleEvents */
6
7 "use strict";
8
9 const { Actor } = require("resource://devtools/shared/protocol.js");
10 const {
11   webconsoleSpec,
12 } = require("resource://devtools/shared/specs/webconsole.js");
13
14 const { ThreadActor } = require("resource://devtools/server/actors/thread.js");
15 const {
16   LongStringActor,
17 } = require("resource://devtools/server/actors/string.js");
18 const {
19   createValueGrip,
20   isArray,
21   stringIsLong,
22 } = require("resource://devtools/server/actors/object/utils.js");
23 const DevToolsUtils = require("resource://devtools/shared/DevToolsUtils.js");
24 const ErrorDocs = require("resource://devtools/server/actors/errordocs.js");
25 const Targets = require("resource://devtools/server/actors/targets/index.js");
26
27 loader.lazyRequireGetter(
28   this,
29   "evalWithDebugger",
30   "resource://devtools/server/actors/webconsole/eval-with-debugger.js",
31   true
32 );
33 loader.lazyRequireGetter(
34   this,
35   "ConsoleFileActivityListener",
36   "resource://devtools/server/actors/webconsole/listeners/console-file-activity.js",
37   true
38 );
39 loader.lazyRequireGetter(
40   this,
41   "jsPropertyProvider",
42   "resource://devtools/shared/webconsole/js-property-provider.js",
43   true
44 );
45 loader.lazyRequireGetter(
46   this,
47   ["isCommand"],
48   "resource://devtools/server/actors/webconsole/commands/parser.js",
49   true
50 );
51 loader.lazyRequireGetter(
```

```
52     this,
53     ["CONSOLE_WORKER_IDS", "WebConsoleUtils"],
54     "resource://devtools/server/actors/webconsole/utils.js",
55     true
56 );
57 loader.lazyRequireGetter(
58     this,
59     ["WebConsoleCommandsManager"],
60     "resource://devtools/server/actors/webconsole/commands/manager.js",
61     true
62 );
63 loader.lazyRequireGetter(
64     this,
65     "EventEmitter",
66     "resource://devtools/shared/event-emitter.js"
67 );
68 loader.lazyRequireGetter(
69     this,
70     "MESSAGE_CATEGORY",
71     "resource://devtools/shared/constants.js",
72     true
73 );
74
75 // Generated by /devtools/shared/webconsole/GenerateReservedWordsJS.py
76 loader.lazyRequireGetter(
77     this,
78     "RESERVED_JS_KEYWORDS",
79     "resource://devtools/shared/webconsole/reserved-js-words.js"
80 );
81
82 // Overwrite implemented listeners for workers so that we don't attempt
83 // to load an unsupported module.
84 if (isWorker) {
85     loader.lazyRequireGetter(
86         this,
87         ["ConsoleAPIListener", "ConsoleServiceListener"],
88         "resource://devtools/server/actors/webconsole/worker-listeners.js",
89         true
90     );
91 } else {
92     loader.lazyRequireGetter(
93         this,
94         "ConsoleAPIListener",
95         "resource://devtools/server/actors/webconsole/listeners/console-api.js",
96         true
97     );
98     loader.lazyRequireGetter(
99         this,
100        "ConsoleServiceListener",
101        "resource://devtools/server/actors/webconsole/listeners/console-service.js",
102        true
```

```

103 );
104 loader.lazyRequireGetter(
105     this,
106     "ConsoleReflowListener",
107     "resource://devtools/server/actors/webconsole/listeners/console-reflow.js",
108     true
109 );
110 loader.lazyRequireGetter(
111     this,
112     "DocumentEventsListener",
113     "resource://devtools/server/actors/webconsole/listeners/document-events.js",
114     true
115 );
116 }
117 loader.lazyRequireGetter(
118     this,
119     "ObjectUtils",
120     "resource://devtools/server/actors/object/utils.js"
121 );
122
123 function isObject(value) {
124     return Object(value) === value;
125 }
126
127 /**
128  * The WebConsoleActor implements capabilities needed for the Web Console
129  * feature.
130  *
131  * @constructor
132  * @param object connection
133  *       The connection to the client, DevToolsServerConnection.
134  * @param object [targetActor]
135  *       Optional, the parent actor.
136  */
137 class WebConsoleActor extends Actor {
138     constructor(connection, targetActor) {
139         super(connection, webconsoleSpec);
140
141         this.targetActor = targetActor;
142
143         this.dbg = this.targetActor.dbg;
144
145         this._gripDepth = 0;
146         this._evalCounter = 0;
147         this._listeners = new Set();
148         this._lastConsoleInputEvaluation = undefined;
149
150         this._onWillNavigate = this._onWillNavigate.bind(this);
151         this._onChangedToplevelDocument =
152             this._onChangedToplevelDocument.bind(this);
153         this.onConsoleServiceMessage = this.onConsoleServiceMessage.bind(this);

```

```

154     this.onConsoleAPICall = this.onConsoleAPICall.bind(this);
155     this.onDocumentEvent = this.onDocumentEvent.bind(this);
156
157     EventEmitter.on(
158         this.targetActor,
159         "changed-toplevel-document",
160         this._onChangedToplevelDocument
161     );
162 }
163
164 /**
165  * Debugger instance.
166  *
167  * @see jsdebugger.sys.mjs
168  */
169 dbg = null;
170
171 /**
172  * This is used by the ObjectActor to keep track of the depth of grip() calls.
173  * @private
174  * @type number
175  */
176 _gripDepth = null;
177
178 /**
179  * Holds a set of all currently registered listeners.
180  *
181  * @private
182  * @type Set
183  */
184 _listeners = null;
185
186 /**
187  * The global we work with (this can be a Window, a Worker global or even a Sandbox
188  * for processes and addons).
189  *
190  * @type nsIDOMWindow, WorkerGlobalScope or Sandbox
191  */
192 get global() {
193     if (this.targetActor.isRootActor) {
194         return this._getWindowForBrowserConsole();
195     }
196     return this.targetActor.targetGlobal;
197 }
198
199 /**
200  * Get a window to use for the browser console.
201  *
202  * (note that is is also used for browser toolbox and webextension
203  * i.e. all targets flagged with isRootActor=true)
204  *

```

```

205 * @private
206 * @return nsIDOMWindow
207 *     The window to use, or null if no window could be found.
208 */
209 _getWindowForBrowserConsole() {
210     // Check if our last used chrome window is still live.
211     let window = this._lastChromeWindow && this._lastChromeWindow.get();
212     // If not, look for a new one.
213     // In case of WebExtension reload of the background page, the last
214     // chrome window might be a dead wrapper, from which we can't check for window.closed.
215     if (!window || Cu.isDeadWrapper(window) || window.closed) {
216         window = this.targetActor.window;
217         if (!window) {
218             // Try to find the Browser Console window to use instead.
219             window = Services.wm.getMostRecentWindow("devtools:webconsole");
220             // We prefer the normal chrome window over the console window,
221             // so we'll look for those windows in order to replace our reference.
222             const onChromeWindowOpened = () => {
223                 // We'll look for this window when someone next requests window()
224                 Services.obs.removeObserver(onChromeWindowOpened, "domwindowopened");
225                 this._lastChromeWindow = null;
226             };
227             Services.obs.addObserver(onChromeWindowOpened, "domwindowopened");
228         }
229         this._handleNewWindow(window);
230     }
231     return window;
232 }
233
234 /**
235  * Store a newly found window on the actor to be used in the future.
236  *
237  * @private
238  * @param nsIDOMWindow window
239  *     The window to store on the actor (can be null).
240  */
241 _handleNewWindow(window) {
242     if (window) {
243         if (this._hadChromeWindow) {
244             Services.console.logStringMessage("Webconsole context has changed");
245         }
246         this._lastChromeWindow = Cu.getWeakReference(window);
247         this._hadChromeWindow = true;
248     } else {
249         this._lastChromeWindow = null;
250     }
251 }
252
253 /**

```

```

256     * Whether we've been using a window before.
257     *
258     * @private
259     * @type boolean
260     */
261     _hadChromWindow = false;
262
263     /**
264     * A weak reference to the last chrome window we used to work with.
265     *
266     * @private
267     * @type nsIWeakReference
268     */
269     _lastChromWindow = null;
270
271     // The evalGlobal is used at the scope for JS evaluation.
272     _evalGlobal = null;
273     get evalGlobal() {
274         return this._evalGlobal || this.global;
275     }
276
277     set evalGlobal(global) {
278         this._evalGlobal = global;
279
280         if (!this._progressListenerActive) {
281             EventEmitter.on(this.targetActor, "will-navigate", this._onWillNavigate);
282             this._progressListenerActive = true;
283         }
284     }
285
286     /**
287     * Flag used to track if we are listening for events from the progress
288     * listener of the target actor. We use the progress listener to clear
289     * this.evalGlobal on page navigation.
290     *
291     * @private
292     * @type boolean
293     */
294     _progressListenerActive = false;
295
296     /**
297     * The ConsoleServiceListener instance.
298     * @type object
299     */
300     consoleServiceListener = null;
301
302     /**
303     * The ConsoleAPIListener instance.
304     */
305     consoleAPIListener = null;
306

```

```

307  /**
308   * The ConsoleFileActivityListener instance.
309   */
310  consoleFileActivityListener = null;
311
312  /**
313   * The ConsoleReflowListener instance.
314   */
315  consoleReflowListener = null;
316
317  grip() {
318      return { actor: this.actorID };
319  }
320
321  _findProtoChain = ThreadActor.prototype._findProtoChain;
322  _removeFromProtoChain = ThreadActor.prototype._removeFromProtoChain;
323
324  /**
325   * Destroy the current WebConsoleActor instance.
326   */
327  destroy() {
328      this.stopListeners();
329      super.destroy();
330
331      EventEmitter.off(
332          this.targetActor,
333          "changed-toplevel-document",
334          this._onChangedToplevelDocument
335      );
336
337      this._lastConsoleInputEvaluation = null;
338      this._evalGlobal = null;
339      this.dbg = null;
340  }
341
342  /**
343   * Create a grip for the given value.
344   *
345   * @param mixed value
346   * @return object
347   */
348  createValueGrip(value) {
349      return createValueGrip(
350          this.targetActor.threadActor,
351          value,
352          this.targetActor.objectsPool
353      );
354  }
355
356  /**
357   * Make a debuggee value for the given value.

```

```

358 *
359 * @param mixed value
360 *     The value you want to get a debuggee value for.
361 * @param boolean useObjectGlobal
362 *     If |true| the object global is determined and added as a debuggee,
363 *     otherwise |this.global| is used when makeDebuggeeValue() is invoked.
364 * @return object
365 *     Debuggee value for |value|.
366 */
367 makeDebuggeeValue(value, useObjectGlobal) {
368     if (useObjectGlobal && isObject(value)) {
369         try {
370             const global = Cu.getGlobalForObject(value);
371             const dbgGlobal = this.dbg.makeGlobalObjectReference(global);
372             return dbgGlobal.makeDebuggeeValue(value);
373         } catch (ex) {
374             // The above can throw an exception if value is not an actual object
375             // or 'Object in compartment marked as invisible to Debugger'
376         }
377     }
378     const dbgGlobal = this.dbg.makeGlobalObjectReference(this.global);
379     return dbgGlobal.makeDebuggeeValue(value);
380 }
381
382 /**
383 * Create a grip for the given string.
384 *
385 * @param string string
386 *     The string you want to create the grip for.
387 * @param object pool
388 *     A Pool where the new actor instance is added.
389 * @return object
390 *     A LongStringActor object that wraps the given string.
391 */
392 longStringGrip(string, pool) {
393     const actor = new LongStringActor(this.conn, string);
394     pool.manage(actor);
395     return actor.form();
396 }
397
398 /**
399 * Create a long string grip if needed for the given string.
400 *
401 * @private
402 * @param string string
403 *     The string you want to create a long string grip for.
404 * @return string|object
405 *     A string is returned if |string| is not a long string.
406 *     A LongStringActor grip is returned if |string| is a long string.
407 */
408 _createStringGrip(string) {

```



```

409     if (string && stringIsLong(string)) {
410         return this.longStringGrip(string, this);
411     }
412     return string;
413 }
414
415 /**
416  * Returns the latest web console input evaluation.
417  * This is undefined if no evaluations have been completed.
418  *
419  * @return object
420  */
421 getLastConsoleInputEvaluation() {
422     return this._lastConsoleInputEvaluation;
423 }
424
425 /**
426  * Preprocess a debugger object (e.g. return the `boundTargetFunction`
427  * debugger object if the given debugger object is a bound function).
428  *
429  * This method is called by both the `inspect` binding implemented
430  * for the webconsole and the one implemented for the devtools API
431  * `browser.devtools.inspectedWindow.eval`.
432  */
433 preprocessDebuggerObject(dbgObj) {
434     // Returns the bound target function on a bound function.
435     if (dbgObj?.isBoundFunction && dbgObj?.boundTargetFunction) {
436         return dbgObj.boundTargetFunction;
437     }
438
439     return dbgObj;
440 }
441
442 /**
443  * This helper is used by the WebExtensionInspectedWindowActor to
444  * inspect an object in the developer toolbox.
445  *
446  * NOTE: shared parts related to preprocess the debugger object (between
447  * this function and the `inspect` webconsole command defined in
448  * "devtools/server/actor/webconsole/utils.js") should be added to
449  * the webconsole actors' `preprocessDebuggerObject` method.
450  */
451 inspectObject(dbgObj, inspectFromAnnotation) {
452     dbgObj = this.preprocessDebuggerObject(dbgObj);
453     this.emit("inspectObject", {
454         objectActor: this.createValueGrip(dbgObj),
455         inspectFromAnnotation,
456     });
457 }
458
459 // Request handlers for known packet types.

```

```

460 /**
461  * Handler for the "startListeners" request.
462  *
463  * @param array listeners
464  *       An array of events to start sent by the Web Console client.
465  * @return object
466  *       The response object which holds the startedListeners array.
467  */
468 // eslint-disable-next-line complexity
469 async startListeners(listeners) {
470   const startedListeners = [];
471   const global = !this.targetActor.isRootActor ? this.global : null;
472   const isTargetActorContentProcess =
473     this.targetActor.targetType === Targets.TYPES.PROCESS;
474
475   for (const event of listeners) {
476     switch (event) {
477       case "PageError":
478         // Workers don't support this message type yet
479         if (isWorker) {
480           break;
481         }
482       case "ConsoleAPI":
483         if (!this.consoleServiceListener) {
484           this.consoleServiceListener = new ConsoleServiceListener(
485             global,
486             this.onConsoleServiceMessage,
487             {
488               matchExactWindow: this.targetActor.ignoreSubFrames,
489             }
490           );
491           this.consoleServiceListener.init();
492         }
493         startedListeners.push(event);
494         break;
495       case "ConsoleAPI":
496         if (!this.consoleAPIListener) {
497           // Create the consoleAPIListener
498           // (and apply the filtering options defined in the parent actor).
499           this.consoleAPIListener = new ConsoleAPIListener(
500             global,
501             this.onConsoleAPICall,
502             {
503               matchExactWindow: this.targetActor.ignoreSubFrames,
504             }
505           );
506           this.consoleAPIListener.init();
507         }
508         startedListeners.push(event);
509         break;
510       case "NetworkActivity":

```

```
511 // Workers don't support this message type
512 if (isWorker) {
513     break;
514 }
515 // Bug 1807650 removed this in favor of the new Watcher/Resources APIs
516 const errorMessage =
517     "NetworkActivity is no longer supported. " +
518     "Instead use Watcher actor's watchResources and listen to NETWORK_EVENT resource";
519 dump(errorMessage + "\n");
520 throw new Error(errorMessage);
521 case "FileActivity":
522     // Workers don't support this message type
523     if (isWorker) {
524         break;
525     }
526     if (this.global instanceof Ci.nsIDOMWindow) {
527         if (!this.consoleFileActivityListener) {
528             this.consoleFileActivityListener =
529                 new ConsoleFileActivityListener(this.global, this);
530         }
531         this.consoleFileActivityListener.startMonitor();
532         startedListeners.push(event);
533     }
534     break;
535 case "ReflowActivity":
536     // Workers don't support this message type
537     if (isWorker) {
538         break;
539     }
540     if (!this.consoleReflowListener) {
541         this.consoleReflowListener = new ConsoleReflowListener(
542             this.global,
543             this
544         );
545     }
546     startedListeners.push(event);
547     break;
548 case "DocumentEvents":
549     // Workers don't support this message type
550     if (isWorker || isTargetActorContentProcess) {
551         break;
552     }
553     if (!this.documentEventsListener) {
554         this.documentEventsListener = new DocumentEventsListener(
555             this.targetActor
556         );
557     }
558     this.documentEventsListener.on("dom-loading", data =>
559         this.onDocumentEvent("dom-loading", data)
560     );
561     this.documentEventsListener.on("dom-interactive", data =>
```

```

562         this.onDocumentEvent("dom-interactive", data)
563     );
564     this.documentEventsListener.on("dom-complete", data =&gt;
565         this.onDocumentEvent("dom-complete", data)
566     );
567
568     this.documentEventsListener.listen();
569 }
570 startedListeners.push(event);
571 break;
572 }
573 }
574
575 // Update the live list of running listeners
576 startedListeners.forEach(this._listeners.add, this._listeners);
577
578 return {
579     startedListeners,
580 };
581 }
582
583 /**
584  * Handler for the "stopListeners" request.
585  *
586  * @param array listeners
587  *     An array of events to stop sent by the Web Console client.
588  * @return object
589  *     The response packet to send to the client: holds the
590  *     stoppedListeners array.
591  */
592 stopListeners(listeners) {
593     const stoppedListeners = [];
594
595     // If no specific listeners are requested to be detached, we stop all
596     // listeners.
597     const eventsToDetach = listeners || [
598         "PageError",
599         "ConsoleAPI",
600         "FileActivity",
601         "ReflowActivity",
602         "DocumentEvents",
603     ];
604
605     for (const event of eventsToDetach) {
606         switch (event) {
607             case "PageError":
608                 if (this.consoleServiceListener) {
609                     this.consoleServiceListener.destroy();
610                     this.consoleServiceListener = null;
611                 }
612                 stoppedListeners.push(event);

```

```

613         break;
614     case "ConsoleAPI":
615         if (this.consoleAPIListener) {
616             this.consoleAPIListener.destroy();
617             this.consoleAPIListener = null;
618         }
619         stoppedListeners.push(event);
620         break;
621     case "FileActivity":
622         if (this.consoleFileActivityListener) {
623             this.consoleFileActivityListener.stopMonitor();
624             this.consoleFileActivityListener = null;
625         }
626         stoppedListeners.push(event);
627         break;
628     case "ReflowActivity":
629         if (this.consoleReflowListener) {
630             this.consoleReflowListener.destroy();
631             this.consoleReflowListener = null;
632         }
633         stoppedListeners.push(event);
634         break;
635     case "DocumentEvents":
636         if (this.documentEventsListener) {
637             this.documentEventsListener.destroy();
638             this.documentEventsListener = null;
639         }
640         stoppedListeners.push(event);
641         break;
642     }
643 }
644
645 // Update the live list of running listeners
646 stoppedListeners.forEach(this._listeners.delete, this._listeners);
647
648 return { stoppedListeners };
649 }
650
651 /**
652  * Handler for the "getCachedMessages" request. This method sends the cached
653  * error messages and the window.console API calls to the client.
654  *
655  * @param array messageTypes
656  *     An array of message types sent by the Web Console client.
657  * @return object
658  *     The response packet to send to the client: it holds the cached
659  *     messages array.
660  */
661 getCachedMessages(messageTypes) {
662     if (!messageTypes) {
663         return {

```

```
664     error: "missingParameter",
665     message: "The messageTypes parameter is missing.",
666   };
667 }
668
669 const messages = [];
670
671 const consoleServiceCachedMessages =
672   messageTypes.includes("PageError") || messageTypes.includes("LogMessage")
673     ? this.consoleServiceListener?.getCachedMessages(
674       !this.targetActor.isRootActor
675     )
676     : null;
677
678 for (const type of messageTypes) {
679   switch (type) {
680     case "ConsoleAPI": {
681       if (!this.consoleAPIListener) {
682         break;
683       }
684
685       // this.global might not be a window (can be a worker global or a Sandbox),
686       // and in such case performance isn't defined
687       const winStartTime =
688         this.global?.performance?.timing?.navigationStart;
689
690       const cache = this.consoleAPIListener.getCachedMessages(
691         !this.targetActor.isRootActor
692       );
693       cache.forEach(cachedMessage => {
694         // Filter out messages that came from a ServiceWorker but happened
695         // before the page was requested.
696         if (
697           cachedMessage.innerID === "ServiceWorker" &&&
698           winStartTime > cachedMessage.timeStamp
699         ) {
700           return;
701         }
702
703         messages.push({
704           message: this.prepareConsoleMessageForRemote(cachedMessage),
705           type: "consoleAPICall",
706         });
707       });
708       break;
709     }
710
711     case "PageError": {
712       if (!consoleServiceCachedMessages) {
713         break;
714       }
```

```

715
716     for (const cachedMessage of consoleServiceCachedMessages) {
717         if (!(cachedMessage instanceof Ci.nsIScriptError)) {
718             continue;
719         }
720
721         messages.push({
722             pageError: this.preparePageErrorForRemote(cachedMessage),
723             type: "pageError",
724         });
725     }
726     break;
727 }
728
729 case "LogMessage": {
730     if (!consoleServiceCachedMessages) {
731         break;
732     }
733
734     for (const cachedMessage of consoleServiceCachedMessages) {
735         if (cachedMessage instanceof Ci.nsIScriptError) {
736             continue;
737         }
738
739         messages.push({
740             message: this._createStringGrip(cachedMessage.message),
741             timeStamp: cachedMessage.microSecondTimeStamp / 1000,
742             type: "logMessage",
743         });
744     }
745     break;
746 }
747 }
748 }
749
750 return {
751     messages,
752 };
753 }
754
755 /**
756  * Handler for the "evaluateJSAsync" request. This method evaluates a given
757  * JavaScript string with an associated `resultID`.
758  *
759  * The result will be returned later as an unsolicited `evaluationResult`,
760  * that can be associated back to this request via the `resultID` field.
761  *
762  * @param object request
763  *       The JSON request object received from the Web Console client.
764  * @return object
765  *       The response packet to send to with the unique id in the

```

```

766     * `resultID` field.
767     */
768     async evaluateJSAsync(request) {
769         const startTime = ChromeUtils.dateNow();
770         // Use a timestamp instead of a UUID as this code is used by workers, which
771         // don't have access to the UUID XPCOM component.
772         // Also use a counter in order to prevent mixing up response when calling
773         // at the exact same time.
774         const resultID = startTime + "-" + this._evalCounter++;
775
776         // Execute the evaluation in the next event loop in order to immediately
777         // reply with the resultID.
778         //
779         // The console input should be evaluated with micro task level != 0,
780         // so that microtask checkpoint isn't performed while evaluating it.
781         DevToolsUtils.executeSoonWithMicroTask(async () => {
782             try {
783                 // Execute the script that may pause.
784                 let response = await this.evaluateJS(request);
785                 // Wait for any potential returned Promise.
786                 response = await this._maybeWaitForResponseResult(response);
787
788                 // Set the timestamp only now, so any messages logged in the expression (e.g. console.log)
789                 // can be appended before the result message (unlike the evaluation result, other
790                 // console resources are throttled before being handled by the webconsole client,
791                 // which might cause some ordering issue).
792                 // Use ChromeUtils.dateNow() as it gives us a higher precision than Date.now().
793                 response.timestamp = ChromeUtils.dateNow();
794                 // Finally, emit an unsolicited evaluationResult packet with the evaluation result.
795                 this.emit("evaluationResult", {
796                     type: "evaluationResult",
797                     resultID,
798                     startTime,
799                     ...response,
800                 });
801             } catch (e) {
802                 const message = `Encountered error while waiting for Helper Result: ${e}\n${e.stack}`;
803                 DevToolsUtils.reportException("evaluateJSAsync", Error(message));
804             }
805         });
806         return { resultID };
807     }
808
809     /**
810     * In order to support async evaluations (e.g. top-level await, ...),
811     * we have to be able to handle promises. This method handles waiting for the promise,
812     * and then returns the result.
813     *
814     * @private
815     * @param object response
816     * The response packet to send to with the unique id in the

```



```

817 * `resultID` field, and potentially a promise in the `helperResult` or in the
818 * `awaitResult` field.
819 *
820 * @return object
821 *     The updated response object.
822 */
823 async _maybeWaitForResponseResult(response) {
824     if (!response?.awaitResult) {
825         return response;
826     }
827
828     let result;
829     try {
830         result = await response.awaitResult;
831
832         // `createValueGrip` expect a debuggee value, while here we have the raw object.
833         // We need to call `makeDebuggeeValue` on it to make it work.
834         const dbgResult = this.makeDebuggeeValue(result);
835         response.result = this.createValueGrip(dbgResult);
836     } catch (e) {
837         // The promise was rejected. We let the engine handle this as it will report a
838         // `uncaught exception` error.
839         response.topLevelAwaitRejected = true;
840     }
841
842     // Remove the promise from the response object.
843     delete response.awaitResult;
844
845     return response;
846 }
847
848 /**
849 * Handler for the "evaluateJS" request. This method evaluates the given
850 * JavaScript string and sends back the result.
851 *
852 * @param object request
853 *     The JSON request object received from the Web Console client.
854 * @return object
855 *     The evaluation response packet.
856 */
857 evaluateJS(request) {
858     const input = request.text;
859
860     const evalOptions = {
861         frameActor: request.frameActor,
862         url: request.url,
863         innerWindowID: request.innerWindowID,
864         selectedNodeActor: request.selectedNodeActor,
865         selectedObjectActor: request.selectedObjectActor,
866         eager: request.eager,
867         bindings: request.bindings,

```

```

868     lineNumber: request.lineNumber,
869     // This flag is set to true in most cases as we consider most evaluations as internal and:
870     // * prevent any breakpoint from being triggered when evaluating the JS input
871     // * prevent spawning Debugger.Source for the evaluated JS and showing it in Debugger UI
872     // This is only set to false when evaluating the console input.
873     disableBreaks: !!request.disableBreaks,
874     // Optional flag, to be set to true when Console Commands should override local symbols with
875     // the same name. Like if the page defines `$`, the evaluated string will use the `$` implemented
876     // by the console command instead of the page's function.
877     preferConsoleCommandsOverLocalSymbols:
878         !!request.preferConsoleCommandsOverLocalSymbols,
879 };
880
881 const { mapped } = request;
882
883 // Set a flag on the thread actor which indicates an evaluation is being
884 // done for the client. This is used to disable all types of breakpoints for all sources
885 // via `disabledBreaks`. When this flag is used, `reportExceptionsWhenBreaksAreDisabled`
886 // allows to still pause on exceptions.
887 this.targetActor.threadActor.insideClientEvaluation = evalOptions;
888
889 let evalInfo;
890 try {
891     evalInfo = evalWithDebugger(input, evalOptions, this);
892 } finally {
893     this.targetActor.threadActor.insideClientEvaluation = null;
894 }
895
896 return new Promise((resolve, reject) => {
897     // Queue up a task to run in the next tick so any microtask created by the evaluated
898     // expression has the time to be run.
899     // e.g. in :
900     // ```
901     // const promiseThenCb = result => "result: " + result;
902     // new Promise(res => res("hello")).then(promiseThenCb)
903     // ```
904     // we want `promiseThenCb` to have run before handling the result.
905     DevToolsUtils.executeSoon(() => {
906         try {
907             const result = this.prepareEvaluationResult(
908                 evalInfo,
909                 input,
910                 request.eager,
911                 mapped,
912                 request.evalInTracer
913             );
914             resolve(result);
915         } catch (err) {
916             reject(err);
917         }
918     });

```

```

919     });
920 }
921
922 // eslint-disable-next-line complexity
923 prepareEvaluationResult(evalInfo, input, eager, mapped, evalInTracer) {
924     const evalResult = evalInfo.result;
925     const helperResult = evalInfo.helperResult;
926
927     let result,
928         errorDocURL,
929         errorMessage,
930         errorNotes = null,
931         errorGrip = null,
932         frame = null,
933         awaitResult,
934         errorMessageName,
935         exceptionStack;
936     if (evalResult) {
937         if ("return" in evalResult) {
938             result = evalResult.return;
939             if (
940                 mapped?.await &&&
941                 result &&&
942                 result.class === "Promise" &&&
943                 typeof result.unsafeDereference === "function"
944             ) {
945                 awaitResult = result.unsafeDereference();
946             }
947         } else if ("yield" in evalResult) {
948             result = evalResult.yield;
949         } else if ("throw" in evalResult) {
950             const error = evalResult.throw;
951             errorGrip = this.createValueGrip(error);
952
953             exceptionStack = this.prepareStackForRemote(evalResult.stack);
954
955             if (exceptionStack) {
956                 // Set the frame based on the topmost stack frame for the exception.
957                 const {
958                     filename: source,
959                     sourceId,
960                     lineNumber: line,
961                     columnNumber: column,
962                 } = exceptionStack[0];
963                 frame = { source, sourceId, line, column };
964
965                 exceptionStack =
966                     WebConsoleUtils.removeFramesAboveDebuggerEval(exceptionStack);
967             }
968
969             errorMessage = String(error);

```

```

970 if (typeof error === "object" && error !== null) {
971     try {
972         errorMessage = DevToolsUtils.callPropertyOnObject(
973             error,
974             "toString"
975         );
976     } catch (e) {
977         // If the debuggee is not allowed to access the "toString" property
978         // of the error object, calling this property from the debuggee's
979         // compartment will fail. The debugger should show the error object
980         // as it is seen by the debuggee, so this behavior is correct.
981         //
982         // Unfortunately, we have at least one test that assumes calling the
983         // "toString" property of an error object will succeed if the
984         // debugger is allowed to access it, regardless of whether the
985         // debuggee is allowed to access it or not.
986         //
987         // To accomodate these tests, if calling the "toString" property
988         // from the debuggee compartment fails, we rewrap the error object
989         // in the debugger's compartment, and then call the "toString"
990         // property from there.
991         if (typeof error.unsafeDereference === "function") {
992             const rawError = error.unsafeDereference();
993             errorMessage = rawError ? rawError.toString() : "";
994         }
995     }
996 }
997
998 // It is possible that we won't have permission to unwrap an
999 // object and retrieve its errorMessageName.
1000 try {
1001     errorDocURL = ErrorDocs.GetURL(error);
1002     errorMessageName = error.errorMessageName;
1003 } catch (ex) {
1004     // ignored
1005 }
1006
1007 try {
1008     const line = error.errorLineNumber;
1009     const column = error.errorColumnNumber;
1010
1011     if (typeof line === "number" && typeof column === "number") {
1012         // Set frame only if we have line/column numbers.
1013         frame = {
1014             source: "debugger eval code",
1015             line,
1016             column,
1017         };
1018     }
1019 } catch (ex) {
1020     // ignored

```

```

1021     }
1022
1023     try {
1024         const notes = error.errorNotes;
1025         if (notes?.length) {
1026             errorNotes = [];
1027             for (const note of notes) {
1028                 errorNotes.push({
1029                     messageBody: this._createStringGrip(note.message),
1030                     frame: {
1031                         source: note.fileName,
1032                         line: note.lineNumber,
1033                         column: note.columnNumber,
1034                     },
1035                 });
1036             }
1037         }
1038     } catch (ex) {
1039         // ignored
1040     }
1041 }
1042
1043 // If a value is encountered that the devtools server doesn't support yet,
1044 // the console should remain functional.
1045 let resultGrip;
1046 if (!awaitResult) {
1047     try {
1048         const objectActor =
1049             this.targetActor.threadActor.getThreadLifetimeObject(result);
1050         if (evalInTracer) {
1051             const tracerActor = this.targetActor.getTargetScopedActor("tracer");
1052             resultGrip = tracerActor.createValueGrip(result);
1053         } else if (objectActor) {
1054             resultGrip = this.targetActor.threadActor.createValueGrip(result);
1055         } else {
1056             resultGrip = this.createValueGrip(result);
1057         }
1058     } catch (e) {
1059         errorMessage = e;
1060     }
1061 }
1062
1063 // Don't update _lastConsoleInputEvaluation in eager evaluation, as it would interfere
1064 // with the $_ command.
1065 if (!eager) {
1066     if (!awaitResult) {
1067         this._lastConsoleInputEvaluation = result;
1068     } else {
1069         // If we evaluated a top-level await expression, we want to assign its result to the
1070         // _lastConsoleInputEvaluation only when the promise resolves, and only if it
1071         // resolves. If the promise rejects, we don't re-assign _lastConsoleInputEvaluation,

```

```

1072 // it will keep its previous value.
1073
1074 const p = awaitResult.then(res => {
1075     this._lastConsoleInputEvaluation = this.makeDebuggeeValue(res);
1076 });
1077
1078 // If the top level await was already rejected (e.g. `await Promise.reject("bleh")`),
1079 // catch the resulting promise of awaitResult.then.
1080 // If we don't do that, the new Promise will also be rejected, and since it's
1081 // unhandled, it will generate an error.
1082 // We don't want to do that for pending promise (e.g. `await new Promise((res, rej) => { setTimeout(rej, 250) })`),
1083 // as the the Promise rejection will be considered as handled, and the "Uncaught (in promise)"
1084 // message wouldn't be emitted.
1085 const { state } = ObjectUtils.getPromiseState(evalResult.return);
1086 if (state === "rejected") {
1087     p.catch(() => {});
1088 }
1089 }
1090 }
1091
1092 return {
1093     input,
1094     result: resultGrip,
1095     awaitResult,
1096     exception: errorGrip,
1097     exceptionMessage: this._createStringGrip(errorMessage),
1098     exceptionDocURL: errorDocURL,
1099     exceptionStack,
1100     hasException: errorGrip !== null,
1101     errorMessageName,
1102     frame,
1103     helperResult,
1104     notes: errorNotes,
1105 };
1106 }
1107
1108 /**
1109  * The Autocomplete request handler.
1110  *
1111  * @param string text
1112  *     The request message - what input to autocomplete.
1113  * @param number cursor
1114  *     The cursor position at the moment of starting autocomplete.
1115  * @param string frameActor
1116  *     The frameactor id of the current paused frame.
1117  * @param string selectedNodeActor
1118  *     The actor id of the currently selected node.
1119  * @param array authorizedEvaluations
1120  *     Array of the properties access which can be executed by the engine.
1121  * @return object
1122  *     The response message - matched properties.

```

```
1123 */
1124 autocomplete(
1125     text,
1126     cursor,
1127     frameActorId,
1128     selectedNodeActor,
1129     authorizedEvaluations,
1130     expressionVars = []
1131 ) {
1132     let dbgObject = null;
1133     let environment = null;
1134     let matches = [];
1135     let matchProp;
1136     let isElementAccess;
1137
1138     const reqText = text.substr(0, cursor);
1139
1140     if (isCommand(reqText)) {
1141         matchProp = reqText;
1142         matches = WebConsoleCommandsManager.getAllColonCommandNames()
1143             .filter(c => `:${c}`.startsWith(reqText))
1144             .map(c => `:${c}`);
1145     } else {
1146         // This is the case of the paused debugger
1147         if (frameActorId) {
1148             const frameActor = this.conn.getActor(frameActorId);
1149             try {
1150                 // Need to try/catch since accessing frame.environment
1151                 // can throw "Debugger.Frame is not live"
1152                 const frame = frameActor.frame;
1153                 environment = frame.environment;
1154             } catch (e) {
1155                 DevToolsUtils.reportException(
1156                     "autocomplete",
1157                     Error("The frame actor was not found: " + frameActorId)
1158                 );
1159             }
1160         } else {
1161             dbgObject = this.dbg.addDebuggee(this.evalGlobal);
1162         }
1163
1164         const result = jsPropertyProvider({
1165             dbgObject,
1166             environment,
1167             frameActorId,
1168             inputValue: text,
1169             cursor,
1170             webconsoleActor: this,
1171             selectedNodeActor,
1172             authorizedEvaluations,
1173             expressionVars,
```

```

1174 });
1175
1176 if (result === null) {
1177     return {
1178         matches: null,
1179     };
1180 }
1181
1182 if (result && result.isUnsafeGetter === true) {
1183     return {
1184         isUnsafeGetter: true,
1185         getterPath: result.getterPath,
1186     };
1187 }
1188
1189 matches = result.matches || new Set();
1190 matchProp = result.matchProp || "";
1191 isElementAccess = result.isElementAccess;
1192
1193 // We consider '$' as alphanumeric because it is used in the names of some
1194 // helper functions; we also consider whitespace as alphanum since it should not
1195 // be seen as break in the eval'd string.
1196 const lastNonAlphaIsDot = /[.][a-zA-Z0-9$\s]*$/ .test(reqText);
1197
1198 // We only return commands and keywords when we are not dealing with a property or
1199 // element access.
1200 if (matchProp && !lastNonAlphaIsDot && !isElementAccess) {
1201     const colonOnlyCommands =
1202         WebConsoleCommandsManager.getColonOnlyCommandNames();
1203     for (const name of WebConsoleCommandsManager.getAllCommandNames()) {
1204         // Filter out commands like `screenshot` as it is inaccessible without the `:` prefix
1205         if (
1206             !colonOnlyCommands.includes(name) &&
1207             name.startsWith(result.matchProp)
1208         ) {
1209             matches.add(name);
1210         }
1211     }
1212
1213     for (const keyword of RESERVED_JS_KEYWORDS) {
1214         if (keyword.startsWith(result.matchProp)) {
1215             matches.add(keyword);
1216         }
1217     }
1218 }
1219
1220 // Sort the results in order to display lowercased item first (e.g. we want to
1221 // display `document` then `Document` as we loosely match the user input if the
1222 // first letter was lowercase).
1223 const firstMeaningfulCharIndex = isElementAccess ? 1 : 0;
1224 matches = Array.from(matches).sort((a, b) => {

```



```

1225     const aFirstMeaningfulChar = a[firstMeaningfulCharIndex];
1226     const bFirstMeaningfulChar = b[firstMeaningfulCharIndex];
1227     const lA =
1228         aFirstMeaningfulChar.toLocaleLowerCase() === aFirstMeaningfulChar;
1229     const lB =
1230         bFirstMeaningfulChar.toLocaleLowerCase() === bFirstMeaningfulChar;
1231     if (lA === lB) {
1232         if (a === matchProp) {
1233             return -1;
1234         }
1235         if (b === matchProp) {
1236             return 1;
1237         }
1238         return a.localeCompare(b);
1239     }
1240     return lA ? -1 : 1;
1241 });
1242 }
1243
1244 return {
1245     matches,
1246     matchProp,
1247     isElementAccess: isElementAccess === true,
1248 };
1249 }
1250
1251 /**
1252  * The "clearMessagesCacheAsync" request handler.
1253  */
1254 clearMessagesCacheAsync() {
1255     if (isWorker) {
1256         // Defined on WorkerScope
1257         clearConsoleEvents();
1258         return;
1259     }
1260
1261     const windowId = !this.targetActor.isRootActor
1262         ? WebConsoleUtils.getInnerWindowId(this.global)
1263         : null;
1264
1265     const ConsoleAPIStorage = Cc[
1266         "@mozilla.org/consoleAPI-storage;1"
1267     ].getService(Ci.nsIConsoleAPIStorage);
1268     ConsoleAPIStorage.clearEvents(windowId);
1269
1270     CONSOLE_WORKER_IDS.forEach(id => {
1271         ConsoleAPIStorage.clearEvents(id);
1272     });
1273
1274     if (this.targetActor.isRootActor || !this.global) {
1275         // If were dealing with the root actor (e.g. the browser console), we want

```

```

1276 // to remove all cached messages, not only the ones specific to a window.
1277 Services.console.reset();
1278 } else if (this.targetActor.ignoreSubFrames) {
1279     Services.console.resetWindow(windowId);
1280 } else {
1281     WebConsoleUtils.getInnerWindowIDsForFrames(this.global).forEach(id =>
1282         Services.console.resetWindow(id)
1283     );
1284 }
1285 }
1286
1287 // End of request handlers.
1288
1289 // Event handlers for various listeners.
1290
1291 /**
1292  * Handler for messages received from the ConsoleServiceListener. This method
1293  * sends the nsIConsoleMessage to the remote Web Console client.
1294  *
1295  * @param nsIConsoleMessage message
1296  *       The message we need to send to the client.
1297  */
1298 onConsoleServiceMessage(message) {
1299     if (message instanceof Ci.nsIScriptError) {
1300         this.emit("pageError", {
1301             pageError: this.preparePageErrorForRemote(message),
1302         });
1303     } else {
1304         this.emit("logMessage", {
1305             message: this._createStringGrip(message.message),
1306             timeStamp: message.microSecondTimeStamp / 1000,
1307         });
1308     }
1309 }
1310
1311 getActorIdForInternalSourceId(id) {
1312     const actor =
1313         this.targetActor.sourcesManager.getSourceActorByInternalSourceId(id);
1314     return actor ? actor.actorID : null;
1315 }
1316
1317 /**
1318  * Prepare a SavedFrame stack to be sent to the client.
1319  *
1320  * @param SavedFrame errorStack
1321  *       Stack for an error we need to send to the client.
1322  * @return object
1323  *       The object you can send to the remote client.
1324  */
1325 prepareStackForRemote(errorStack) {
1326     // Convert stack objects to the JSON attributes expected by client code

```

```

1327 // Bug 1348885: If the global from which this error came from has been
1328 // nuked, stack is going to be a dead wrapper.
1329 if (!errorStack || (Cu && Cu.isDeadWrapper(errorStack))) {
1330     return null;
1331 }
1332 const stack = [];
1333 let s = errorStack;
1334 while (s) {
1335     stack.push({
1336         filename: s.source,
1337         sourceId: this.getActorIdForInternalSourceId(s.sourceId),
1338         lineNumber: s.line,
1339         columnNumber: s.column,
1340         functionName: s.functionDisplayName,
1341         asyncCause: s.asyncCause ? s.asyncCause : undefined,
1342     });
1343     s = s.parent || s.asyncParent;
1344 }
1345 return stack;
1346 }
1347
1348 /**
1349  * Prepare an nsIScriptError to be sent to the client.
1350  *
1351  * @param nsIScriptError pageError
1352  *       The page error we need to send to the client.
1353  * @return object
1354  *       The object you can send to the remote client.
1355  */
1356 preparePageErrorForRemote(pageError) {
1357     const stack = this.prepareStackForRemote(pageError.stack);
1358     let notesArray = null;
1359     const notes = pageError.notes;
1360     if (notes?.length) {
1361         notesArray = [];
1362         for (let i = 0, len = notes.length; i < len; i++) {
1363             const note = notes.queryElementAt(i, Ci.nsIScriptErrorNote);
1364             notesArray.push({
1365                 messageBody: this._createStringGrip(note.errorMessage),
1366                 frame: {
1367                     source: note.sourceName,
1368                     sourceId: this.getActorIdForInternalSourceId(note.sourceId),
1369                     line: note.lineNumber,
1370                     column: note.columnNumber,
1371                 },
1372             });
1373         }
1374     }
1375
1376     // If there is no location information in the error but we have a stack,
1377     // fill in the location with the first frame on the stack.

```

```
1378 let { sourceName, sourceId, lineNumber, columnNumber } = pageError;
1379 if (!sourceName &&& !sourceId &&& !lineNumber &&& !columnNumber &&& stack) {
1380     sourceName = stack[0].filename;
1381     sourceId = stack[0].sourceId;
1382     lineNumber = stack[0].lineNumber;
1383     columnNumber = stack[0].columnNumber;
1384 }
1385
1386 const isCSSMessage = pageError.category === MESSAGE_CATEGORY.CSS_PARSER;
1387
1388 const result = {
1389     errorMessage: this._createStringGrip(pageError.errorMessage),
1390     errorMessageName: isCSSMessage ? undefined : pageError.errorMessageName,
1391     exceptionDocURL: ErrorDocs.GetURL(pageError),
1392     sourceName,
1393     sourceId: this.getActorIdForInternalSourceId(sourceId),
1394     lineNumber,
1395     columnNumber,
1396     category: pageError.category,
1397     innerWindowID: pageError.innerWindowID,
1398     timeStamp: pageError.microSecondTimeStamp / 1000,
1399     warning: !(pageError.flags & pageError.warningFlag),
1400     error: !(pageError.flags & (pageError.warningFlag | pageError.infoFlag)),
1401     info: !(pageError.flags & pageError.infoFlag),
1402     private: pageError.isFromPrivateWindow,
1403     stacktrace: stack,
1404     notes: notesArray,
1405     chromeContext: pageError.isFromChromeContext,
1406     isPromiseRejection: isCSSMessage
1407         ? undefined
1408         : pageError.isPromiseRejection,
1409     isForwardedFromContentProcess: pageError.isForwardedFromContentProcess,
1410     cssSelectors: isCSSMessage ? pageError.cssSelectors : undefined,
1411 };
1412
1413 // If the pageError does have an exception object, we want to return the grip for it,
1414 // but only if we do manage to get the grip, as we're checking the property on the
1415 // client to render things differently.
1416 if (pageError.hasException) {
1417     try {
1418         const obj = this.makeDebuggeeValue(pageError.exception, true);
1419         if (obj?.class !== "DeadObject") {
1420             result.exception = this.createValueGrip(obj);
1421             result.hasException = true;
1422         }
1423     } catch (e) {}
1424 }
1425
1426 return result;
1427 }
1428
```

```

1429 /**
1430  * Handler for window.console API calls received from the ConsoleAPIListener.
1431  * This method sends the object to the remote Web Console client.
1432  *
1433  * @see ConsoleAPIListener
1434  * @param object message
1435  *      The console API call we need to send to the remote client.
1436  * @param object extraProperties
1437  *      an object whose properties will be folded in the packet that is emitted.
1438  */
1439 onConsoleAPICall(message, extraProperties = {}) {
1440     this.emit("consoleAPICall", {
1441         message: this.prepareConsoleMessageForRemote(message),
1442         ...extraProperties,
1443     });
1444 }
1445
1446 /**
1447  * Handler for the DocumentEventsListener.
1448  *
1449  * @see DocumentEventsListener
1450  * @param {String} name
1451  *      The document event name that either of followings.
1452  *      - dom-loading
1453  *      - dom-interactive
1454  *      - dom-complete
1455  * @param {Number} time
1456  *      The time that the event is fired.
1457  * @param {Boolean} hasNativeConsoleAPI
1458  *      Tells if the window.console object is native or overwritten by script in the page.
1459  *      Only passed when `name` is "dom-complete" (see devtools/server/actors/webconsole/listeners/document-events.js).
1460  */
1461 onDocumentEvent(name, { time, hasNativeConsoleAPI }) {
1462     this.emit("documentEvent", {
1463         name,
1464         time,
1465         hasNativeConsoleAPI,
1466     });
1467 }
1468
1469 /**
1470  * Handler for file activity. This method sends the file request information
1471  * to the remote Web Console client.
1472  *
1473  * @see ConsoleFileActivityListener
1474  * @param string fileURI
1475  *      The requested file URI.
1476  */
1477 onFileActivity(fileURI) {
1478     this.emit("fileActivity", {
1479         uri: fileURI,

```

```

1480 });
1481 }
1482
1483 // End of event handlers for various listeners.
1484
1485 /**
1486  * Prepare a message from the console API to be sent to the remote Web Console
1487  * instance.
1488  *
1489  * @param object message
1490  *       The original message received from the console storage listener.
1491  * @param boolean aUseObjectGlobal
1492  *       If |true| the object global is determined and added as a debuggee,
1493  *       otherwise |this.global| is used when makeDebuggeeValue() is invoked.
1494  * @return object
1495  *       The object that can be sent to the remote client.
1496  */
1497 prepareConsoleMessageForRemote(message, useObjectGlobal = true) {
1498     const result = {
1499         arguments: message.arguments
1500             ? message.arguments.map(obj => {
1501                 const dbgObj = this.makeDebuggeeValue(obj, useObjectGlobal);
1502                 return this.createValueGrip(dbgObj);
1503             })
1504         : [],
1505         chromeContext: message.chromeContext,
1506         columnNumber: message.columnNumber,
1507         filename: message.filename,
1508         level: message.level,
1509         lineNumber: message.lineNumber,
1510         // messages emitted from Console.sys.mjs don't have a microSecondTimeStamp property
1511         timeStamp: message.microSecondTimeStamp
1512             ? message.microSecondTimeStamp / 1000
1513             : message.timeStamp,
1514         sourceId: this.getActorIdForInternalSourceId(message.sourceId),
1515         category: message.category || "webdev",
1516         innerWindowID: message.innerID,
1517     };
1518
1519     // It only make sense to include the following properties in the message when they have
1520     // a meaningful value. Otherwise we simply don't include them so we save cycles in JSActor communication.
1521     if (message.counter) {
1522         result.counter = message.counter;
1523     }
1524     if (message.private) {
1525         result.private = message.private;
1526     }
1527     if (message.prefix) {
1528         result.prefix = message.prefix;
1529     }
1530

```

```

1531     if (message.stacktrace) {
1532         result.stacktrace = message.stacktrace.map(frame => {
1533             return {
1534                 ...frame,
1535                 sourceId: this.getActorIdForInternalSourceId(frame.sourceId),
1536             };
1537         });
1538     }
1539
1540     if (message.styles && message.styles.length) {
1541         result.styles = message.styles.map(string => {
1542             return this.createValueGrip(string);
1543         });
1544     }
1545
1546     if (message.timer) {
1547         result.timer = message.timer;
1548     }
1549
1550     if (message.level === "table") {
1551         const tableItems = this._getConsoleTableMessageItems(result);
1552         if (tableItems) {
1553             result.arguments[0].ownProperties = tableItems;
1554             result.arguments[0].preview = null;
1555         }
1556
1557         // Only return the 2 first params.
1558         result.arguments = result.arguments.slice(0, 2);
1559     }
1560
1561     return result;
1562 }
1563
1564 /**
1565  * Return the properties needed to display the appropriate table for a given
1566  * console.table call.
1567  * This function does a little more than creating an ObjectActor for the first
1568  * parameter of the message. When layout out the console table in the output, we want
1569  * to be able to look into sub-properties so the table can have a different layout (
1570  * for arrays of arrays, objects with objects properties, arrays of objects, ...).
1571  * So here we need to retrieve the properties of the first parameter, and also all the
1572  * sub-properties we might need.
1573  *
1574  * @param {Object} result: The console.table message.
1575  * @returns {Object} An object containing the properties of the first argument of the
1576  *                   console.table call.
1577  */
1578 _getConsoleTableMessageItems(result) {
1579     if (
1580         !result ||
1581         !Array.isArray(result.arguments) ||

```

```

1582     !result.arguments.length
1583 ) {
1584     return null;
1585 }
1586
1587 const [tableItemGrip] = result.arguments;
1588 const dataType = tableItemGrip.class;
1589 const needEntries = ["Map", "WeakMap", "Set", "WeakSet"].includes(dataType);
1590 const ignoreNonIndexedProperties = isArray(tableItemGrip);
1591
1592 const tableItemActor = this.targetActor.objectsPool.getActorByID(
1593     tableItemGrip.actor
1594 );
1595 if (!tableItemActor) {
1596     return null;
1597 }
1598
1599 // Retrieve the properties (or entries for Set/Map) of the console table first arg.
1600 const iterator = needEntries
1601     ? tableItemActor.enumEntries()
1602     : tableItemActor.enumProperties({
1603         ignoreNonIndexedProperties,
1604     });
1605 const { ownProperties } = iterator.all();
1606
1607 // The iterator returns a descriptor for each property, wherein the value could be
1608 // in one of those sub-property.
1609 const descriptorKeys = ["safeGetterValues", "getterValue", "value"];
1610
1611 Object.values(ownProperties).forEach(desc => {
1612     if (typeof desc !== "undefined") {
1613         descriptorKeys.forEach(key => {
1614             if (desc && desc.hasOwnProperty(key)) {
1615                 const grip = desc[key];
1616
1617                 // We need to load sub-properties as well to render the table in a nice way.
1618                 const actor =
1619                     grip && this.targetActor.objectsPool.getActorByID(grip.actor);
1620                 if (actor) {
1621                     const res = actor
1622                         .enumProperties({
1623                             ignoreNonIndexedProperties: isArray(grip),
1624                         })
1625                         .all();
1626                     if (res?.ownProperties) {
1627                         desc[key].ownProperties = res.ownProperties;
1628                     }
1629                 }
1630             }
1631         });
1632     }

```



```
1633     });
1634
1635     return ownProperties;
1636 }
1637
1638 /**
1639  * The "will-navigate" progress listener. This is used to clear the current
1640  * eval scope.
1641  */
1642 _onWillNavigate({ isTopLevel }) {
1643   if (isTopLevel) {
1644     this._evalGlobal = null;
1645     EventEmitter.off(this.targetActor, "will-navigate", this._onWillNavigate);
1646     this._progressListenerActive = false;
1647   }
1648 }
1649
1650 /**
1651  * This listener is called when we switch to another frame,
1652  * mostly to unregister previous listeners and start listening on the new document.
1653  */
1654 _onChangedToplevelDocument() {
1655   // Convert the Set to an Array
1656   const listeners = [...this._listeners];
1657
1658   // Unregister existing listener on the previous document
1659   // (pass a copy of the array as it will shift from it)
1660   this.stopListeners(listeners.slice());
1661
1662   // This method is called after this.global is changed,
1663   // so we register new listener on this new global
1664   this.startListeners(listeners);
1665
1666   // Also reset the cached top level chrome window being targeted
1667   this._lastChromeWindow = null;
1668 }
1669 }
1670
1671 exports.WebConsoleActor = WebConsoleActor;
1672 </pre></body></html>
```