```
<html><head><meta name="viewport" content="width=device-width"><title>jar:file:///C:/Program%20Files/Mozilla%20Firefox/browser/omni.ja!/chrome/devt
 * License, v. 2.0. If a copy of the MPL was not distributed with this
 * file, You can obtain one at http://mozilla.org/MPL/2.0/. */

/* global clearConsoleEvents */

"use strict";

const { Actor } = require("resource://devtools/shared/protocol.js");
const {
  webconsoleSpec,
} = require("resource://devtools/shared/specs/webconsole.js");

const { ThreadActor } = require("resource://devtools/server/actors/thread.js");
const {
  LongStringActor,
} = require("resource://devtools/server/actors/string.js");
const {
  createValueGrip,
  isArray,
  stringIsLong,
} = require("resource://devtools/server/actors/object/utils.js");
const DevToolsUtils = require("resource://devtools/shared/DevToolsUtils.js");
const ErrorDocs = require("resource://devtools/server/actors/errordocs.js");
const Targets = require("resource://devtools/server/actors/targets/index.js");

loader.lazyRequireGetter(
  this,
  "evalWithDebugger",
  "resource://devtools/server/actors/webconsole/eval-with-debugger.js",
  true
);
loader.lazyRequireGetter(
  this,
  "ConsoleFileActivityListener",
  "resource://devtools/server/actors/webconsole/listeners/console-file-activity.js",
  true
);
loader.lazyRequireGetter(
  this,
  "jsPropertyProvider",
  "resource://devtools/shared/webconsole/js-property-provider.js",
  true
);
loader.lazyRequireGetter(
  this,
  ["isCommand"],
  "resource://devtools/server/actors/webconsole/commands/parser.js",
  true
);
loader.lazyRequireGetter(
  this,
  ["CONSOLE_WORKER_IDS", "WebConsoleUtils"],
  "resource://devtools/server/actors/webconsole/utils.js",
  true
);
loader.lazyRequireGetter(
  this,
  ["WebConsoleCommandsManager"],
  "resource://devtools/server/actors/webconsole/commands/manager.js",
  true
);
loader.lazyRequireGetter(
  this,
  "EventEmitter",
  "resource://devtools/shared/event-emitter.js"
);
loader.lazyRequireGetter(
  this,
  "MESSAGE_CATEGORY",
  "resource://devtools/shared/constants.js",
  true
);

// Generated by /devtools/shared/webconsole/GenerateReservedWordsJS.py
loader.lazyRequireGetter(
  this,
  "RESERVED_JS_KEYWORDS",
  "resource://devtools/shared/webconsole/reserved-js-words.js"
);

// Overwrite implemented listeners for workers so that we don't attempt
// to load an unsupported module.
if (isWorker) {
  loader.lazyRequireGetter(
    this,
    ["ConsoleAPIListener", "ConsoleServiceListener"],
    "resource://devtools/server/actors/webconsole/worker-listeners.js",
    true
  );
} else {
  loader.lazyRequireGetter(
    this,
    "ConsoleAPIListener",
    "resource://devtools/server/actors/webconsole/listeners/console-api.js",
    true
  );
  loader.lazyRequireGetter(
    this,
    "ConsoleServiceListener",
    "resource://devtools/server/actors/webconsole/listeners/console-service.js",
    true
  );
  loader.lazyRequireGetter(
    this,
    "ConsoleReflowListener",
    "resource://devtools/server/actors/webconsole/listeners/console-reflow.js",
    true
  );
  loader.lazyRequireGetter(
    this,
    "DocumentEventsListener",
    "resource://devtools/server/actors/webconsole/listeners/document-events.js",
    true
  );
}
loader.lazyRequireGetter(
  this,
  "ObjectUtils",
  "resource://devtools/server/actors/object/utils.js"
);

function isObject(value) {
  return Object(value) === value;
}

/**
 * The WebConsoleActor implements capabilities needed for the Web Console
 * feature.
 *
 * @constructor
 * @param object connection
 *        The connection to the client, DevToolsServerConnection.
 * @param object [targetActor]
 *        Optional, the parent actor.
 */
class WebConsoleActor extends Actor {
  constructor(connection, targetActor) {
    super(connection, webconsoleSpec);

    this.targetActor = targetActor;

    this.dbg = this.targetActor.dbg;

    this._gripDepth = 0;
    this._evalCounter = 0;
    this._listeners = new Set();
    this._lastConsoleInputEvaluation = undefined;

    this._onWillNavigate = this._onWillNavigate.bind(this);
    this._onChangedToplevelDocument =
      this._onChangedToplevelDocument.bind(this);
    this.onConsoleServiceMessage = this.onConsoleServiceMessage.bind(this);
    this.onConsoleAPICall = this.onConsoleAPICall.bind(this);
    this.onDocumentEvent = this.onDocumentEvent.bind(this);

    EventEmitter.on(
      this.targetActor,
      "changed-toplevel-document",
      this._onChangedToplevelDocument
    );
  }

  /**
   * Debugger instance.
   *
   * @see jsdebugger.sys.mjs
   */
  dbg = null;

  /**
   * This is used by the ObjectActor to keep track of the depth of grip() calls.
   * @private
   * @type number
   */
  _gripDepth = null;

  /**
   * Holds a set of all currently registered listeners.
   *
   * @private
   * @type Set
   */
  _listeners = null;

  /**
   * The global we work with (this can be a Window, a Worker global or even a Sandbox
   * for processes and addons).
   *
   * @type nsIDOMWindow, WorkerGlobalScope or Sandbox
   */
  get global() {
    if (this.targetActor.isRootActor) {
      return this._getWindowForBrowserConsole();
    }
    return this.targetActor.targetGlobal;
  }

  /**
   * Get a window to use for the browser console.
   *
   * (note that is is also used for browser toolbox and webextension
   *  i.e. all targets flagged with isRootActor=true)
   *
   * @private
   * @return nsIDOMWindow
   *         The window to use, or null if no window could be found.
   */
  _getWindowForBrowserConsole() {
    // Check if our last used chrome window is still live.
    let window = this._lastChromeWindow &amp;&amp; this._lastChromeWindow.get();
    // If not, look for a new one.
    // In case of WebExtension reload of the background page, the last
    // chrome window might be a dead wrapper, from which we can't check for window.closed.
    if (!window || Cu.isDeadWrapper(window) || window.closed) {
      window = this.targetActor.window;
      if (!window) {
        // Try to find the Browser Console window to use instead.
        window = Services.wm.getMostRecentWindow("devtools:webconsole");
        // We prefer the normal chrome window over the console window,
        // so we'll look for those windows in order to replace our reference.
        const onChromeWindowOpened = () =&gt; {
          // We'll look for this window when someone next requests window()
          Services.obs.removeObserver(onChromeWindowOpened, "domwindowopened");
          this._lastChromeWindow = null;
        };
        Services.obs.addObserver(onChromeWindowOpened, "domwindowopened");
      }

      this._handleNewWindow(window);
    }

    return window;
  }

  /**
   * Store a newly found window on the actor to be used in the future.
   *
   * @private
   * @param nsIDOMWindow window
   *        The window to store on the actor (can be null).
   */
  _handleNewWindow(window) {
    if (window) {
      if (this._hadChromeWindow) {
        Services.console.logStringMessage("Webconsole context has changed");
      }
      this._lastChromeWindow = Cu.getWeakReference(window);
      this._hadChromeWindow = true;
    } else {
      this._lastChromeWindow = null;
    }
  }

  /**
```

```
   * Whether we've been using a window before.
   *
   * @private
   * @type boolean
   */
  _hadChromeWindow = false;

  /**
   * A weak reference to the last chrome window we used to work with.
   *
   * @private
   * @type nsIWeakReference
   */
  _lastChromeWindow = null;

  // The evalGlobal is used at the scope for JS evaluation.
  _evalGlobal = null;
  get evalGlobal() {
    return this._evalGlobal || this.global;
  }

  set evalGlobal(global) {
    this._evalGlobal = global;

    if (!this._progressListenerActive) {
      EventEmitter.on(this.targetActor, "will-navigate", this._onWillNavigate);
      this._progressListenerActive = true;
    }
  }

  /**
   * Flag used to track if we are listening for events from the progress
   * listener of the target actor. We use the progress listener to clear
   * this.evalGlobal on page navigation.
   *
   * @private
   * @type boolean
   */
  _progressListenerActive = false;

  /**
   * The ConsoleServiceListener instance.
   * @type object
   */
  consoleServiceListener = null;

  /**
   * The ConsoleAPIListener instance.
   */
  consoleAPIListener = null;

  /**
   * The ConsoleFileActivityListener instance.
   */
  consoleFileActivityListener = null;

  /**
   * The ConsoleReflowListener instance.
   */
  consoleReflowListener = null;

  grip() {
    return { actor: this.actorID };
  }

  _findProtoChain = ThreadActor.prototype._findProtoChain;
  _removeFromProtoChain = ThreadActor.prototype._removeFromProtoChain;

  /**
   * Destroy the current WebConsoleActor instance.
   */
  destroy() {
    this.stopListeners();
    super.destroy();

    EventEmitter.off(
      this.targetActor,
      "changed-toplevel-document",
      this._onChangedToplevelDocument
    );

    this._lastConsoleInputEvaluation = null;
    this._evalGlobal = null;
    this.dbg = null;
  }

  /**
   * Create a grip for the given value.
   *
   * @param mixed value
   * @return object
   */
  createValueGrip(value) {
    return createValueGrip(
      this.targetActor.threadActor,
      value,
      this.targetActor.objectsPool
    );
  }

  /**
   * Make a debuggee value for the given value.
   *
   * @param mixed value
   *        The value you want to get a debuggee value for.
   * @param boolean useObjectGlobal
   *        If |true| the object global is determined and added as a debuggee,
   *        otherwise |this.global| is used when makeDebuggeeValue() is invoked.
   * @return object
   *         Debuggee value for |value|.
   */
  makeDebuggeeValue(value, useObjectGlobal) {
    if (useObjectGlobal &amp;&amp; isObject(value)) {
      try {
        const global = Cu.getGlobalForObject(value);
        const dbgGlobal = this.dbg.makeGlobalObjectReference(global);
        return dbgGlobal.makeDebuggeeValue(value);
      } catch (ex) {
        // The above can throw an exception if value is not an actual object
        // or 'Object in compartment marked as invisible to Debugger'
      }
    }
    const dbgGlobal = this.dbg.makeGlobalObjectReference(this.global);
    return dbgGlobal.makeDebuggeeValue(value);
  }

  /**
   * Create a grip for the given string.
   *
   * @param string string
   *        The string you want to create the grip for.
   * @param object pool
   *        A Pool where the new actor instance is added.
   * @return object
   *         A LongStringActor object that wraps the given string.
   */
  longStringGrip(string, pool) {
    const actor = new LongStringActor(this.conn, string);
    pool.manage(actor);
    return actor.form();
  }

  /**
   * Create a long string grip if needed for the given string.
   *
   * @private
   * @param string string
   *        The string you want to create a long string grip for.
   * @return string|object
   *         A string is returned if |string| is not a long string.
   *         A LongStringActor grip is returned if |string| is a long string.
   */
  _createStringGrip(string) {
    if (string &amp;&amp; stringIsLong(string)) {
      return this.longStringGrip(string, this);
    }
    return string;
  }

  /**
   * Returns the latest web console input evaluation.
   * This is undefined if no evaluations have been completed.
   *
   * @return object
   */
  getLastConsoleInputEvaluation() {
    return this._lastConsoleInputEvaluation;
  }

  /**
   * Preprocess a debugger object (e.g. return the `boundTargetFunction`
   * debugger object if the given debugger object is a bound function).
   *
   * This method is called by both the `inspect` binding implemented
   * for the webconsole and the one implemented for the devtools API
   * `browser.devtools.inspectedWindow.eval`.
   */
  preprocessDebuggerObject(dbgObj) {
    // Returns the bound target function on a bound function.
    if (dbgObj?.isBoundFunction &amp;&amp; dbgObj?.boundTargetFunction) {
      return dbgObj.boundTargetFunction;
    }

    return dbgObj;
  }

  /**
   * This helper is used by the WebExtensionInspectedWindowActor to
   * inspect an object in the developer toolbox.
   *
   * NOTE: shared parts related to preprocess the debugger object (between
   * this function and the `inspect` webconsole command defined in
   * "devtools/server/actor/webconsole/utils.js") should be added to
   * the webconsole actors' `preprocessDebuggerObject` method.
   */
  inspectObject(dbgObj, inspectFromAnnotation) {
    dbgObj = this.preprocessDebuggerObject(dbgObj);
    this.emit("inspectObject", {
      objectActor: this.createValueGrip(dbgObj),
      inspectFromAnnotation,
    });
  }

  // Request handlers for known packet types.

  /**
   * Handler for the "startListeners" request.
   *
   * @param array listeners
   *        An array of events to start sent by the Web Console client.
   * @return object
   *        The response object which holds the startedListeners array.
   */
  // eslint-disable-next-line complexity
  async startListeners(listeners) {
    const startedListeners = [];
    const global = !this.targetActor.isRootActor ? this.global : null;
    const isTargetActorContentProcess =
      this.targetActor.targetType === Targets.TYPES.PROCESS;

    for (const event of listeners) {
      switch (event) {
        case "PageError":
          // Workers don't support this message type yet
          if (isWorker) {
            break;
          }
          if (!this.consoleServiceListener) {
            this.consoleServiceListener = new ConsoleServiceListener(
              global,
              this.onConsoleServiceMessage,
              {
                matchExactWindow: this.targetActor.ignoreSubFrames,
              }
            );
            this.consoleServiceListener.init();
          }
          startedListeners.push(event);
          break;
        case "ConsoleAPI":
          if (!this.consoleAPIListener) {
            // Create the consoleAPIListener
            // (and apply the filtering options defined in the parent actor).
            this.consoleAPIListener = new ConsoleAPIListener(
              global,
              this.onConsoleAPICall,
              {
                matchExactWindow: this.targetActor.ignoreSubFrames,
              }
            );
            this.consoleAPIListener.init();
          }
          startedListeners.push(event);
          break;
        case "NetworkActivity":
          // Workers don't support this message type
          if (isWorker) {
            break;
          }
          // Bug 1807650 removed this in favor of the new Watcher/Resources APIs
          const errorMessage =
            "NetworkActivity is no longer supported. " +
            "Instead use Watcher actor's watchResources and listen to NETWORK_EVENT resource";
          dump(errorMessage + "\n");
          throw new Error(errorMessage);
        case "FileActivity":
          // Workers don't support this message type
          if (isWorker) {
            break;
          }
          if (this.global instanceof Ci.nsIDOMWindow) {
            if (!this.consoleFileActivityListener) {
              this.consoleFileActivityListener =
                new ConsoleFileActivityListener(this.global, this);
            }
            this.consoleFileActivityListener.startMonitor();
            startedListeners.push(event);
          }
          break;
        case "ReflowActivity":
          // Workers don't support this message type
          if (isWorker) {
            break;
          }
          if (!this.consoleReflowListener) {
            this.consoleReflowListener = new ConsoleReflowListener(
              this.global,
              this
            );
          }
          startedListeners.push(event);
          break;
        case "DocumentEvents":
          // Workers don't support this message type
          if (isWorker || isTargetActorContentProcess) {
            break;
          }
          if (!this.documentEventsListener) {
            this.documentEventsListener = new DocumentEventsListener(
              this.targetActor
            );

            this.documentEventsListener.on("dom-loading", data =&gt;
              this.onDocumentEvent("dom-loading", data)
            );
            this.documentEventsListener.on("dom-interactive", data =&gt;
              this.onDocumentEvent("dom-interactive", data)
            );
            this.documentEventsListener.on("dom-complete", data =&gt;
              this.onDocumentEvent("dom-complete", data)
            );

            this.documentEventsListener.listen();
          }
          startedListeners.push(event);
          break;
      }
    }

    // Update the live list of running listeners
    startedListeners.forEach(this._listeners.add, this._listeners);

    return {
      startedListeners,
    };
  }

  /**
   * Handler for the "stopListeners" request.
   *
   * @param array listeners
   *        An array of events to stop sent by the Web Console client.
   * @return object
   *        The response packet to send to the client: holds the
   *        stoppedListeners array.
   */
  stopListeners(listeners) {
    const stoppedListeners = [];

    // If no specific listeners are requested to be detached, we stop all
    // listeners.
    const eventsToDetach = listeners || [
      "PageError",
      "ConsoleAPI",
      "FileActivity",
      "ReflowActivity",
      "DocumentEvents",
    ];

    for (const event of eventsToDetach) {
      switch (event) {
        case "PageError":
          if (this.consoleServiceListener) {
            this.consoleServiceListener.destroy();
            this.consoleServiceListener = null;
          }
          stoppedListeners.push(event);
```

```
          break;
        case "ConsoleAPI":
          if (this.consoleAPIListener) {
            this.consoleAPIListener.destroy();
            this.consoleAPIListener = null;
          }
          stoppedListeners.push(event);
          break;
        case "FileActivity":
          if (this.consoleFileActivityListener) {
            this.consoleFileActivityListener.stopMonitor();
            this.consoleFileActivityListener = null;
          }
          stoppedListeners.push(event);
          break;
        case "ReflowActivity":
          if (this.consoleReflowListener) {
            this.consoleReflowListener.destroy();
            this.consoleReflowListener = null;
          }
          stoppedListeners.push(event);
          break;
        case "DocumentEvents":
          if (this.documentEventsListener) {
            this.documentEventsListener.destroy();
            this.documentEventsListener = null;
          }
          stoppedListeners.push(event);
          break;
      }
    }

    // Update the live list of running listeners
    stoppedListeners.forEach(this._listeners.delete, this._listeners);

    return { stoppedListeners };
  }

  /**
   * Handler for the "getCachedMessages" request. This method sends the cached
   * error messages and the window.console API calls to the client.
   *
   * @param array messageTypes
   *        An array of message types sent by the Web Console client.
   * @return object
   *         The response packet to send to the client: it holds the cached
   *         messages array.
   */
  getCachedMessages(messageTypes) {
    if (!messageTypes) {
      return {
        error: "missingParameter",
        message: "The messageTypes parameter is missing.",
      };
    }

    const messages = [];

    const consoleServiceCachedMessages =
      messageTypes.includes("PageError") || messageTypes.includes("LogMessage")
        ? this.consoleServiceListener?.getCachedMessages(
            !this.targetActor.isRootActor
          )
        : null;

    for (const type of messageTypes) {
      switch (type) {
        case "ConsoleAPI": {
          if (!this.consoleAPIListener) {
            break;
          }

          // this.global might not be a window (can be a worker global or a Sandbox),
          // and in such case performance isn't defined
          const winStartTime =
            this.global?.performance?.timing?.navigationStart;

          const cache = this.consoleAPIListener.getCachedMessages(
            !this.targetActor.isRootActor
          );
          cache.forEach(cachedMessage =&gt; {
            // Filter out messages that came from a ServiceWorker but happened
            // before the page was requested.
            if (
              cachedMessage.innerID === "ServiceWorker" &amp;&amp;
              winStartTime &gt; cachedMessage.timeStamp
            ) {
              return;
            }

            messages.push({
              message: this.prepareConsoleMessageForRemote(cachedMessage),
              type: "consoleAPICall",
            });
          });
          break;
        }

        case "PageError": {
          if (!consoleServiceCachedMessages) {
            break;
          }

          for (const cachedMessage of consoleServiceCachedMessages) {
            if (!(cachedMessage instanceof Ci.nsIScriptError)) {
              continue;
            }

            messages.push({
              pageError: this.preparePageErrorForRemote(cachedMessage),
              type: "pageError",
            });
          }
          break;
        }

        case "LogMessage": {
          if (!consoleServiceCachedMessages) {
            break;
          }

          for (const cachedMessage of consoleServiceCachedMessages) {
            if (cachedMessage instanceof Ci.nsIScriptError) {
              continue;
            }

            messages.push({
              message: this._createStringGrip(cachedMessage.message),
              timeStamp: cachedMessage.microSecondTimeStamp / 1000,
              type: "logMessage",
            });
          }
          break;
        }
      }
    }

    return {
      messages,
    };
  }

  /**
   * Handler for the "evaluateJSAsync" request. This method evaluates a given
   * JavaScript string with an associated `resultID`.
   *
   * The result will be returned later as an unsolicited `evaluationResult`,
   * that can be associated back to this request via the `resultID` field.
   *
   * @param object request
   *        The JSON request object received from the Web Console client.
   * @return object
   *         The response packet to send to with the unique id in the
```

```
   *          `resultID` field.
   */
  async evaluateJSAsync(request) {
    const startTime = ChromeUtils.dateNow();
    // Use  a timestamp instead of a UUID as this code is used by workers, which
    // don't have access to the UUID XPCOM component.
    // Also use a counter in order to prevent mixing up response when calling
    // at the exact same time.
    const resultID = startTime + "-" + this._evalCounter++;

    // Execute the evaluation in the next event loop in order to immediately
    // reply with the resultID.
    //
    // The console input should be evaluated with micro task level != 0,
    // so that microtask checkpoint isn't performed while evaluating it.
    DevToolsUtils.executeSoonWithMicroTask(async () =&gt; {
      try {
        // Execute the script that may pause.
        let response = await this.evaluateJS(request);
        // Wait for any potential returned Promise.
        response = await this._maybeWaitForResponseResult(response);

        // Set the timestamp only now, so any messages logged in the expression (e.g. console.log)
        // can be appended before the result message (unlike the evaluation result, other
        // console resources are throttled before being handled by the webconsole client,
        // which might cause some ordering issue).
        // Use ChromeUtils.dateNow() as it gives us a higher precision than Date.now().
        response.timestamp = ChromeUtils.dateNow();
        // Finally, emit an unsolicited evaluationResult packet with the evaluation result.
        this.emit("evaluationResult", {
          type: "evaluationResult",
          resultID,
          startTime,
          ...response,
        });
      } catch (e) {
        const message = `Encountered error while waiting for Helper Result: ${e}\n${e.stack}`;
        DevToolsUtils.reportException("evaluateJSAsync", Error(message));
      }
    });
    return { resultID };
  }

  /**
   * In order to support async evaluations (e.g. top-level await, …),
   * we have to be able to handle promises. This method handles waiting for the promise,
   * and then returns the result.
   *
   * @private
   * @param object response
   *          The response packet to send to with the unique id in the
   *         `resultID` field, and potentially a promise in the `helperResult` or in the
   *         `awaitResult` field.
   *
   * @return object
   *         The updated response object.
   */
  async _maybeWaitForResponseResult(response) {
    if (!response?.awaitResult) {
      return response;
    }

    let result;
    try {
      result = await response.awaitResult;

      // `createValueGrip` expect a debuggee value, while here we have the raw object.
      // We need to call `makeDebuggeeValue` on it to make it work.
      const dbgResult = this.makeDebuggeeValue(result);
      response.result = this.createValueGrip(dbgResult);
    } catch (e) {
      // The promise was rejected. We let the engine handle this as it will report a
      // `uncaught exception` error.
      response.topLevelAwaitRejected = true;
    }

    // Remove the promise from the response object.
    delete response.awaitResult;

    return response;
  }

  /**
   * Handler for the "evaluateJS" request. This method evaluates the given
   * JavaScript string and sends back the result.
   *
   * @param object request
   *        The JSON request object received from the Web Console client.
   * @return object
   *         The evaluation response packet.
   */
  evaluateJS(request) {
    const input = request.text;

    const evalOptions = {
      frameActor: request.frameActor,
      url: request.url,
      innerWindowID: request.innerWindowID,
      selectedNodeActor: request.selectedNodeActor,
      selectedObjectActor: request.selectedObjectActor,
      eager: request.eager,
      bindings: request.bindings,
      lineNumber: request.lineNumber,
      // This flag is set to true in most cases as we consider most evaluations as internal and:
      // * prevent any breakpoint from being triggerred when evaluating the JS input
      // * prevent spawning Debugger.Source for the evaluated JS and showing it in Debugger UI
      // This is only set to false when evaluating the console input.
      disableBreaks: !!request.disableBreaks,
      // Optional flag, to be set to true when Console Commands should override local symbols with
      // the same name. Like if the page defines `$`, the evaluated string will use the `$` implemented
      // by the console command instead of the page's function.
      preferConsoleCommandsOverLocalSymbols:
        !!request.preferConsoleCommandsOverLocalSymbols,
    };

    const { mapped } = request;

    // Set a flag on the thread actor which indicates an evaluation is being
    // done for the client. This is used to disable all types of breakpoints for all sources
    // via `disabledBreaks`. When this flag is used, `reportExceptionsWhenBreaksAreDisabled`
    // allows to still pause on exceptions.
    this.targetActor.threadActor.insideClientEvaluation = evalOptions;

    let evalInfo;
    try {
      evalInfo = evalWithDebugger(input, evalOptions, this);
    } finally {
      this.targetActor.threadActor.insideClientEvaluation = null;
    }

    return new Promise((resolve, reject) =&gt; {
      // Queue up a task to run in the next tick so any microtask created by the evaluated
      // expression has the time to be run.
      // e.g. in :
      // ```
      // const promiseThenCb = result =&gt; "result: " + result;
      // new Promise(res =&gt; res("hello")).then(promiseThenCb)
      // ```
      // we want`promiseThenCb` to have run before handling the result.
      DevToolsUtils.executeSoon(() =&gt; {
        try {
          const result = this.prepareEvaluationResult(
            evalInfo,
            input,
            request.eager,
            mapped,
            request.evalInTracer
          );
          resolve(result);
        } catch (err) {
          reject(err);
        }
      });
```

```
  });
}

// eslint-disable-next-line complexity
prepareEvaluationResult(evalInfo, input, eager, mapped, evalInTracer) {
  const evalResult = evalInfo.result;
  const helperResult = evalInfo.helperResult;

  let result,
    errorDocURL,
    errorMessage,
    errorNotes = null,
    errorGrip = null,
    frame = null,
    awaitResult,
    errorMessageName,
    exceptionStack;
  if (evalResult) {
    if ("return" in evalResult) {
      result = evalResult.return;
      if (
        mapped?.await &amp;&amp;
        result &amp;&amp;
        result.class === "Promise" &amp;&amp;
        typeof result.unsafeDereference === "function"
      ) {
        awaitResult = result.unsafeDereference();
      }
    } else if ("yield" in evalResult) {
      result = evalResult.yield;
    } else if ("throw" in evalResult) {
      const error = evalResult.throw;
      errorGrip = this.createValueGrip(error);

      exceptionStack = this.prepareStackForRemote(evalResult.stack);

      if (exceptionStack) {
        // Set the frame based on the topmost stack frame for the exception.
        const {
          filename: source,
          sourceId,
          lineNumber: line,
          columnNumber: column,
        } = exceptionStack[0];
        frame = { source, sourceId, line, column };

        exceptionStack =
          WebConsoleUtils.removeFramesAboveDebuggerEval(exceptionStack);
      }

      errorMessage = String(error);
```

```
        if (typeof error === "object" &amp;&amp; error !== null) {
          try {
            errorMessage = DevToolsUtils.callPropertyOnObject(
              error,
              "toString"
            );
          } catch (e) {
            // If the debuggee is not allowed to access the "toString" property
            // of the error object, calling this property from the debuggee's
            // compartment will fail. The debugger should show the error object
            // as it is seen by the debuggee, so this behavior is correct.
            //
            // Unfortunately, we have at least one test that assumes calling the
            // "toString" property of an error object will succeed if the
            // debugger is allowed to access it, regardless of whether the
            // debuggee is allowed to access it or not.
            //
            // To accomodate these tests, if calling the "toString" property
            // from the debuggee compartment fails, we rewrap the error object
            // in the debugger's compartment, and then call the "toString"
            // property from there.
            if (typeof error.unsafeDereference === "function") {
              const rawError = error.unsafeDereference();
              errorMessage = rawError ? rawError.toString() : "";
            }
          }
        }

        // It is possible that we won't have permission to unwrap an
        // object and retrieve its errorMessageName.
        try {
          errorDocURL = ErrorDocs.GetURL(error);
          errorMessageName = error.errorMessageName;
        } catch (ex) {
          // ignored
        }

        try {
          const line = error.errorLineNumber;
          const column = error.errorColumnNumber;

          if (typeof line === "number" &amp;&amp; typeof column === "number") {
            // Set frame only if we have line/column numbers.
            frame = {
              source: "debugger eval code",
              line,
              column,
            };
          }
        } catch (ex) {
          // ignored
        }

        try {
          const notes = error.errorNotes;
          if (notes?.length) {
            errorNotes = [];
            for (const note of notes) {
              errorNotes.push({
                messageBody: this._createStringGrip(note.message),
                frame: {
                  source: note.fileName,
                  line: note.lineNumber,
                  column: note.columnNumber,
                },
              });
            }
          }
        } catch (ex) {
          // ignored
        }
      }
    }
    // If a value is encountered that the devtools server doesn't support yet,
    // the console should remain functional.
    let resultGrip;
    if (!awaitResult) {
      try {
        const objectActor =
          this.targetActor.threadActor.getThreadLifetimeObject(result);
        if (evalInTracer) {
          const tracerActor = this.targetActor.getTargetScopedActor("tracer");
          resultGrip = tracerActor.createValueGrip(result);
        } else if (objectActor) {
          resultGrip = this.targetActor.threadActor.createValueGrip(result);
        } else {
          resultGrip = this.createValueGrip(result);
        }
      } catch (e) {
        errorMessage = e;
      }
    }

    // Don't update _lastConsoleInputEvaluation in eager evaluation, as it would interfere
    // with the $_ command.
    if (!eager) {
      if (!awaitResult) {
        this._lastConsoleInputEvaluation = result;
      } else {
        // If we evaluated a top-level await expression, we want to assign its result to the
        // _lastConsoleInputEvaluation only when the promise resolves, and only if it
        // resolves. If the promise rejects, we don't re-assign _lastConsoleInputEvaluation,
        // it will keep its previous value.

        const p = awaitResult.then(res =&gt; {
          this._lastConsoleInputEvaluation = this.makeDebuggeeValue(res);
        });

        // If the top level await was already rejected (e.g. `await Promise.reject("bleh")`),
        // catch the resulting promise of awaitResult.then.
        // If we don't do that, the new Promise will also be rejected, and since it's
        // unhandled, it will generate an error.
        // We don't want to do that for pending promise (e.g. `await new Promise((res, rej) =&gt; setTimeout(rej,250))`),
        // as the the Promise rejection will be considered as handled, and the "Uncaught (in promise)"
        // message wouldn't be emitted.
        const { state } = ObjectUtils.getPromiseState(evalResult.return);
        if (state === "rejected") {
          p.catch(() =&gt; {});
        }
      }
    }

    return {
      input,
      result: resultGrip,
      awaitResult,
      exception: errorGrip,
      exceptionMessage: this._createStringGrip(errorMessage),
      exceptionDocURL: errorDocURL,
      exceptionStack,
      hasException: errorGrip !== null,
      errorMessageName,
      frame,
      helperResult,
      notes: errorNotes,
    };
  }

  /**
   * The Autocomplete request handler.
   *
   * @param string text
   *        The request message - what input to autocomplete.
   * @param number cursor
   *        The cursor position at the moment of starting autocomplete.
   * @param string frameActor
   *        The frameactor id of the current paused frame.
   * @param string selectedNodeActor
   *        The actor id of the currently selected node.
   * @param array authorizedEvaluations
   *        Array of the properties access which can be executed by the engine.
   * @return object
   *         The response message - matched properties.
   */
  autocomplete(
    text,
    cursor,
    frameActorId,
    selectedNodeActor,
    authorizedEvaluations,
    expressionVars = []
  ) {
    let dbgObject = null;
    let environment = null;
    let matches = [];
    let matchProp;
    let isElementAccess;

    const reqText = text.substr(0, cursor);

    if (isCommand(reqText)) {
      matchProp = reqText;
      matches = WebConsoleCommandsManager.getAllColonCommandNames()
        .filter(c =&gt; `:${c}`.startsWith(reqText))
        .map(c =&gt; `:${c}`);
    } else {
      // This is the case of the paused debugger
      if (frameActorId) {
        const frameActor = this.conn.getActor(frameActorId);
        try {
          // Need to try/catch since accessing frame.environment
          // can throw "Debugger.Frame is not live"
          const frame = frameActor.frame;
          environment = frame.environment;
        } catch (e) {
          DevToolsUtils.reportException(
            "autocomplete",
            Error("The frame actor was not found: " + frameActorId)
          );
        }
      } else {
        dbgObject = this.dbg.addDebuggee(this.evalGlobal);
      }

      const result = jsPropertyProvider({
        dbgObject,
        environment,
        frameActorId,
        inputValue: text,
        cursor,
        webconsoleActor: this,
        selectedNodeActor,
        authorizedEvaluations,
        expressionVars,
      });

      if (result === null) {
        return {
          matches: null,
        };
      }

      if (result &amp;&amp; result.isUnsafeGetter === true) {
        return {
          isUnsafeGetter: true,
          getterPath: result.getterPath,
        };
      }

      matches = result.matches || new Set();
      matchProp = result.matchProp || "";
      isElementAccess = result.isElementAccess;

      // We consider '$' as alphanumeric because it is used in the names of some
      // helper functions; we also consider whitespace as alphanum since it should not
      // be seen as break in the evaled string.
      const lastNonAlphaIsDot = /[.][a-zA-Z0-9$\s]*$/.test(reqText);

      // We only return commands and keywords when we are not dealing with a property or
      // element access.
      if (matchProp &amp;&amp; !lastNonAlphaIsDot &amp;&amp; !isElementAccess) {
        const colonOnlyCommands =
          WebConsoleCommandsManager.getColonOnlyCommandNames();
        for (const name of WebConsoleCommandsManager.getAllCommandNames()) {
          // Filter out commands like `screenshot` as it is inaccessible without the `:` prefix
          if (
            !colonOnlyCommands.includes(name) &amp;&amp;
            name.startsWith(result.matchProp)
          ) {
            matches.add(name);
          }
        }

        for (const keyword of RESERVED_JS_KEYWORDS) {
          if (keyword.startsWith(result.matchProp)) {
            matches.add(keyword);
          }
        }
      }

      // Sort the results in order to display lowercased item first (e.g. we want to
      // display `document` then `Document` as we loosely match the user input if the
      // first letter was lowercase).
      const firstMeaningfulCharIndex = isElementAccess ? 1 : 0;
      matches = Array.from(matches).sort((a, b) =&gt; {
        const aFirstMeaningfulChar = a[firstMeaningfulCharIndex];
        const bFirstMeaningfulChar = b[firstMeaningfulCharIndex];
        const lA =
          aFirstMeaningfulChar.toLocaleLowerCase() === aFirstMeaningfulChar;
        const lB =
          bFirstMeaningfulChar.toLocaleLowerCase() === bFirstMeaningfulChar;
        if (lA === lB) {
          if (a === matchProp) {
            return -1;
          }
          if (b === matchProp) {
            return 1;
          }
          return a.localeCompare(b);
        }
        return lA ? -1 : 1;
      });
    }

    return {
      matches,
      matchProp,
      isElementAccess: isElementAccess === true,
    };
  }

  /**
   * The "clearMessagesCacheAsync" request handler.
   */
  clearMessagesCacheAsync() {
    if (isWorker) {
      // Defined on WorkerScope
      clearConsoleEvents();
      return;
    }

    const windowId = !this.targetActor.isRootActor
      ? WebConsoleUtils.getInnerWindowId(this.global)
      : null;

    const ConsoleAPIStorage = Cc[
      "@mozilla.org/consoleAPI-storage;1"
    ].getService(Ci.nsIConsoleAPIStorage);
    ConsoleAPIStorage.clearEvents(windowId);

    CONSOLE_WORKER_IDS.forEach(id =&gt; {
      ConsoleAPIStorage.clearEvents(id);
    });

    if (this.targetActor.isRootActor || !this.global) {
      // If were dealing with the root actor (e.g. the browser console), we want
      // to remove all cached messages, not only the ones specific to a window.
      Services.console.reset();
    } else if (this.targetActor.ignoreSubFrames) {
      Services.console.resetWindow(windowId);
    } else {
      WebConsoleUtils.getInnerWindowIDsForFrames(this.global).forEach(id =&gt;
        Services.console.resetWindow(id)
      );
    }
  }

  // End of request handlers.

  // Event handlers for various listeners.

  /**
   * Handler for messages received from the ConsoleServiceListener. This method
   * sends the nsIConsoleMessage to the remote Web Console client.
   *
   * @param nsIConsoleMessage message
   *        The message we need to send to the client.
   */
  onConsoleServiceMessage(message) {
    if (message instanceof Ci.nsIScriptError) {
      this.emit("pageError", {
        pageError: this.preparePageErrorForRemote(message),
      });
    } else {
      this.emit("logMessage", {
        message: this._createStringGrip(message.message),
        timeStamp: message.microSecondTimeStamp / 1000,
      });
    }
  }

  getActorIdForInternalSourceId(id) {
    const actor =
      this.targetActor.sourcesManager.getSourceActorByInternalSourceId(id);
    return actor ? actor.actorID : null;
  }

  /**
   * Prepare a SavedFrame stack to be sent to the client.
   *
   * @param SavedFrame errorStack
   *        Stack for an error we need to send to the client.
   * @return object
   *         The object you can send to the remote client.
   */
  prepareStackForRemote(errorStack) {
    // Convert stack objects to the JSON attributes expected by client code
    // Bug 1348885: If the global from which this error came from has been
    // nuked, stack is going to be a dead wrapper.
    if (!errorStack || (Cu &amp;&amp; Cu.isDeadWrapper(errorStack))) {
      return null;
    }
    const stack = [];
    let s = errorStack;
    while (s) {
      stack.push({
        filename: s.source,
        sourceId: this.getActorIdForInternalSourceId(s.sourceId),
        lineNumber: s.line,
        columnNumber: s.column,
        functionName: s.functionDisplayName,
        asyncCause: s.asyncCause ? s.asyncCause : undefined,
      });
      s = s.parent || s.asyncParent;
    }
    return stack;
  }

  /**
   * Prepare an nsIScriptError to be sent to the client.
   *
   * @param nsIScriptError pageError
   *        The page error we need to send to the client.
   * @return object
   *         The object you can send to the remote client.
   */
  preparePageErrorForRemote(pageError) {
    const stack = this.prepareStackForRemote(pageError.stack);
    let notesArray = null;
    const notes = pageError.notes;
    if (notes?.length) {
      notesArray = [];
      for (let i = 0, len = notes.length; i &lt; len; i++) {
        const note = notes.queryElementAt(i, Ci.nsIScriptErrorNote);
        notesArray.push({
          messageBody: this._createStringGrip(note.errorMessage),
          frame: {
            source: note.sourceName,
            sourceId: this.getActorIdForInternalSourceId(note.sourceId),
            line: note.lineNumber,
            column: note.columnNumber,
          },
        });
      }
    }

    // If there is no location information in the error but we have a stack,
    // fill in the location with the first frame on the stack.
    let { sourceName, sourceId, lineNumber, columnNumber } = pageError;
    if (!sourceName &amp;&amp; !sourceId &amp;&amp; !lineNumber &amp;&amp; !columnNumber &amp;&amp; stack) {
      sourceName = stack[0].filename;
      sourceId = stack[0].sourceId;
      lineNumber = stack[0].lineNumber;
      columnNumber = stack[0].columnNumber;
    }

    const isCSSMessage = pageError.category === MESSAGE_CATEGORY.CSS_PARSER;

    const result = {
      errorMessage: this._createStringGrip(pageError.errorMessage),
      errorMessageName: isCSSMessage ? undefined : pageError.errorMessageName,
      exceptionDocURL: ErrorDocs.GetURL(pageError),
      sourceName,
      sourceId: this.getActorIdForInternalSourceId(sourceId),
      lineNumber,
      columnNumber,
      category: pageError.category,
      innerWindowID: pageError.innerWindowID,
      timeStamp: pageError.microSecondTimeStamp / 1000,
      warning: !!(pageError.flags &amp; pageError.warningFlag),
      error: !(pageError.flags &amp; (pageError.warningFlag | pageError.infoFlag)),
      info: !!(pageError.flags &amp; pageError.infoFlag),
      private: pageError.isFromPrivateWindow,
      stacktrace: stack,
      notes: notesArray,
      chromeContext: pageError.isFromChromeContext,
      isPromiseRejection: isCSSMessage
        ? undefined
        : pageError.isPromiseRejection,
      isForwardedFromContentProcess: pageError.isForwardedFromContentProcess,
      cssSelectors: isCSSMessage ? pageError.cssSelectors : undefined,
    };

    // If the pageError does have an exception object, we want to return the grip for it,
    // but only if we do manage to get the grip, as we're checking the property on the
    // client to render things differently.
    if (pageError.hasException) {
      try {
        const obj = this.makeDebuggeeValue(pageError.exception, true);
        if (obj?.class !== "DeadObject") {
          result.exception = this.createValueGrip(obj);
          result.hasException = true;
        }
      } catch (e) {}
    }

    return result;
  }

  /**
   * Handler for window.console API calls received from the ConsoleAPIListener.
   * This method sends the object to the remote Web Console client.
   *
   * @see ConsoleAPIListener
   * @param object message
   *        The console API call we need to send to the remote client.
   * @param object extraProperties
   *        an object whose properties will be folded in the packet that is emitted.
   */
  onConsoleAPICall(message, extraProperties = {}) {
    this.emit("consoleAPICall", {
      message: this.prepareConsoleMessageForRemote(message),
      ...extraProperties,
    });
  }

  /**
   * Handler for the DocumentEventsListener.
   *
   * @see DocumentEventsListener
   * @param {String} name
   *        The document event name that either of followings.
   *        - dom-loading
   *        - dom-interactive
   *        - dom-complete
   * @param {Number} time
   *        The time that the event is fired.
   * @param {Boolean} hasNativeConsoleAPI
   *        Tells if the window.console object is native or overwritten by script in the page.
   *        Only passed when `name` is "dom-complete" (see devtools/server/actors/webconsole/listeners/document-events.js).
   */
  onDocumentEvent(name, { time, hasNativeConsoleAPI }) {
    this.emit("documentEvent", {
      name,
      time,
      hasNativeConsoleAPI,
    });
  }

  /**
   * Handler for file activity. This method sends the file request information
   * to the remote Web Console client.
   *
   * @see ConsoleFileActivityListener
   * @param string fileURI
   *        The requested file URI.
   */
  onFileActivity(fileURI) {
    this.emit("fileActivity", {
      uri: fileURI,
    });
  }

  // End of event handlers for various listeners.

  /**
   * Prepare a message from the console API to be sent to the remote Web Console
   * instance.
   *
   * @param object message
   *        The original message received from the console storage listener.
   * @param boolean aUseObjectGlobal
   *        If |true| the object global is determined and added as a debuggee,
   *        otherwise |this.global| is used when makeDebuggeeValue() is invoked.
   * @return object
   *         The object that can be sent to the remote client.
   */
  prepareConsoleMessageForRemote(message, useObjectGlobal = true) {
    const result = {
      arguments: message.arguments
        ? message.arguments.map(obj =&gt; {
            const dbgObj = this.makeDebuggeeValue(obj, useObjectGlobal);
            return this.createValueGrip(dbgObj);
          })
        : [],
      chromeContext: message.chromeContext,
      columnNumber: message.columnNumber,
      filename: message.filename,
      level: message.level,
      lineNumber: message.lineNumber,
      // messages emitted from Console.sys.mjs don't have a microSecondTimeStamp property
      timeStamp: message.microSecondTimeStamp
        ? message.microSecondTimeStamp / 1000
        : message.timeStamp,
      sourceId: this.getActorIdForInternalSourceId(message.sourceId),
      category: message.category || "webdev",
      innerWindowID: message.innerID,
    };

    // It only make sense to include the following properties in the message when they have
    // a meaningful value. Otherwise we simply don't include them so we save cycles in JSActor communication.
    if (message.counter) {
      result.counter = message.counter;
    }
    if (message.private) {
      result.private = message.private;
    }
    if (message.prefix) {
      result.prefix = message.prefix;
    }

```

```
    if (message.stacktrace) {
      result.stacktrace = message.stacktrace.map(frame =&gt; {
        return {
          ...frame,
          sourceId: this.getActorIdForInternalSourceId(frame.sourceId),
        };
      });
    }

    if (message.styles &amp;&amp; message.styles.length) {
      result.styles = message.styles.map(string =&gt; {
        return this.createValueGrip(string);
      });
    }

    if (message.timer) {
      result.timer = message.timer;
    }

    if (message.level === "table") {
      const tableItems = this._getConsoleTableMessageItems(result);
      if (tableItems) {
        result.arguments[0].ownProperties = tableItems;
        result.arguments[0].preview = null;
      }

      // Only return the 2 first params.
      result.arguments = result.arguments.slice(0, 2);
    }

    return result;
  }

  /**
   * Return the properties needed to display the appropriate table for a given
   * console.table call.
   * This function does a little more than creating an ObjectActor for the first
   * parameter of the message. When layout out the console table in the output, we want
   * to be able to look into sub-properties so the table can have a different layout (
   * for arrays of arrays, objects with objects properties, arrays of objects, …).
   * So here we need to retrieve the properties of the first parameter, and also all the
   * sub-properties we might need.
   *
   * @param {Object} result: The console.table message.
   * @returns {Object} An object containing the properties of the first argument of the
   *                   console.table call.
   */
  _getConsoleTableMessageItems(result) {
    if (
      !result ||
      !Array.isArray(result.arguments) ||
      !result.arguments.length
    ) {
      return null;
    }

    const [tableItemGrip] = result.arguments;
    const dataType = tableItemGrip.class;
    const needEntries = ["Map", "WeakMap", "Set", "WeakSet"].includes(dataType);
    const ignoreNonIndexedProperties = isArray(tableItemGrip);

    const tableItemActor = this.targetActor.objectsPool.getActorByID(
      tableItemGrip.actor
    );
    if (!tableItemActor) {
      return null;
    }

    // Retrieve the properties (or entries for Set/Map) of the console table first arg.
    const iterator = needEntries
      ? tableItemActor.enumEntries()
      : tableItemActor.enumProperties({
          ignoreNonIndexedProperties,
        });
    const { ownProperties } = iterator.all();

    // The iterator returns a descriptor for each property, wherein the value could be
    // in one of those sub-property.
    const descriptorKeys = ["safeGetterValues", "getterValue", "value"];

    Object.values(ownProperties).forEach(desc =&gt; {
      if (typeof desc !== "undefined") {
        descriptorKeys.forEach(key =&gt; {
          if (desc &amp;&amp; desc.hasOwnProperty(key)) {
            const grip = desc[key];

            // We need to load sub-properties as well to render the table in a nice way.
            const actor =
              grip &amp;&amp; this.targetActor.objectsPool.getActorByID(grip.actor);
            if (actor) {
              const res = actor
                .enumProperties({
                  ignoreNonIndexedProperties: isArray(grip),
                })
                .all();
              if (res?.ownProperties) {
                desc[key].ownProperties = res.ownProperties;
              }
            }
          }
        });
      }
    });

    return ownProperties;
  }

  /**
   * The "will-navigate" progress listener. This is used to clear the current
   * eval scope.
   */
  _onWillNavigate({ isTopLevel }) {
    if (isTopLevel) {
      this._evalGlobal = null;
      EventEmitter.off(this.targetActor, "will-navigate", this._onWillNavigate);
      this._progressListenerActive = false;
    }
  }

  /**
   * This listener is called when we switch to another frame,
   * mostly to unregister previous listeners and start listening on the new document.
   */
  _onChangedToplevelDocument() {
    // Convert the Set to an Array
    const listeners = [...this._listeners];

    // Unregister existing listener on the previous document
    // (pass a copy of the array as it will shift from it)
    this.stopListeners(listeners.slice());

    // This method is called after this.global is changed,
    // so we register new listener on this new global
    this.startListeners(listeners);

    // Also reset the cached top level chrome window being targeted
    this._lastChromeWindow = null;
  }
}

exports.WebConsoleActor = WebConsoleActor;
</pre></body></html>
```