WebDriver

W3C Recommendation 05 June 2018

This version:
https://www.w3.org/TR/2018/REC-webdriver1-20180605/

Latest published version:
https://www.w3.org/TR/webdriver1/

Latest editor's draft:
https://w3c.github.io/webdriver/

Implementation report:
https://github.com/w3c/webdriver/blob/master/implementation-report.md

Previous version:
https://www.w3.org/TR/2018/PR-webdriver1-20180426/

Editors:
Simon Stewart
David Burns (Mozilla)

Participate:
GitHub w3c/webdriver
File a bug
Commit history
Pull requests

Channel:
#webdriver on irc.w3.org

Please check the **errata** for any errors or issues reported since publication.

See also **translations**.

Abstract

WebDriver is a remote control interface that enables introspection and control of user agents. It provides a platform- and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers.

Provided is a set of interfaces to discover and manipulate DOM elements in web documents and to control the behavior of a user agent. It is primarily intended to allow web authors to write tests that

automate a user agent from a separate controlling process, but may also be used in such a way as to allow in-browser scripts to control a — possibly separate — browser.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

Use GitHub w3c/webdriver for comments/issues.

This document was published by the Browser Testing and Tools Working Group as a Recommendation. Comments regarding this document are welcome. Please send them to public-browser-tools-testing@w3.org (subscribe, archives).

Please see the Working Group's implementation report.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 February 2018 W3C Process Document.

Table of Contents

1. Conformance

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The key words “*MUST*”, “*MUST NOT*”, “*REQUIRED*”, “*SHOULD*”, “*SHOULD NOT*”, “*RECOMMENDED*”, “*MAY*”, and “*OPTIONAL*” in the normative parts of this document are to be interpreted as described in [RFC2119]. The key word “OPTIONALLY” in the normative parts of this document is to be interpreted with the same normative meaning as “*MAY*” and “*OPTIONAL*”.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent.

1.1 Dependencies

This specification relies on several other underlying specifications.

Web App Security
The following terms are defined in the Content Security Policy Level 3 specification: [CSP3]

- ***Directives***
- ***Should block navigation response***

DOM
The following terms are defined in the Document Object Model specification: [DOM]

- ***Attribute***
- ***compareDocumentPosition***
- ***connected***
- ***context object***
- ***Descendant***
- ***Document element***
- ***Document***
- ***DOCUMENT_POSITION_DISCONNECTED*** (1)
- ***Document type***
- ***Document URL***
- ***Element***
- ***Equals***
- ***Event***
- ***Fire an event***
- ***Get an attribute by name***
- ***getAttribute***

- *getElementsByTagName*
- *hasAttribute*
- *HTMLCollection*
- *Inclusive descendant*
- *isTrusted*
- *Node document*
- *Node Length*
- *Node*
- *NodeList*
- *querySelectorAll*
- *querySelector*
- *tagName*
- *Text node*

The following attributes are defined in the Document Object Model specification: [DOM]

- *textContent attribute*

The following attributes are defined in the DOM Parsing and Serialisation specification: [DOM-PARSING]

- *innerHTML IDL attribute*
- *outerHTML IDL attribute*
- *serializeToString*

The following attributes are defined in the UI Events specification: [UI-EVENTS]

- *Activation trigger*
- *click event*
- *Keyboard event*
- *Keyboard event order*
- *keyDown event*
- *keyPress event*
- *keyUp event*
- *mouseDown event*
- *Mouse event*

- ***Mouse event order***
- ***mouseMove event***
- ***mouseOver event***
- ***mouseUp event***

The following attributes are defined in the UI Events Code specification: [UIEVENTS-CODE]

- ***Keyboard event code tables***

The following attributes are defined in the UI Events Code specification: [UIEVENTS-KEY]

- ***Keyboard modifier keys***

DOM Parsing

The following terms are defined in the DOM Parsing and Serialization Specification: [DOMPARSING]

- ***Fragment serializing algorithm***

ECMAScript

The following terms are defined in the ECMAScript Language Specification: [ECMA-262]

- ***Abrupt Completion***
- ***Directive prologue***
- ***Early error***
- ***Function***
- ***FunctionCreate***
- ***FunctionBody***
- ***Global environment***
- ***Own property***
- ***parseFloat***
- ***realm***
- ***Use strict directive***

This specification also presumes that you are able to call some of the ***internal methods*** from the ECMAScript Language Specification:

- ***Call***
- ***[[Class]]***

- *[[GetOwnProperty]]*
- *[[GetProperty]]*
- *Index of*
- *[[Put]]*
- *Substring*

The ECMAScript Language Specification also defines the following types, values, and operations that are used throughout this specification:

- *Array*
- *Boolean* type
- *List*
- *maximum safe integer*
- *Null*
- *Number*
- *Object*
- *[[Parse]]*
- *String*
- *[[Stringify]]*
- *ToInteger*
- *Undefined*

Fetch

The following terms are defined in the WHATWG Fetch specification: [FETCH]

- *Body*
- *Header*
- *Header Name*
- *Header Value*
- *Local scheme*
- *Method*
- *Response*
- *Request*
- *Set Header*
- *HTTP Status*

- *Status message*

Fullscreen

The following terms are defined in the WHATWG Fullscreen specification: [FULLSCREEN]

- *Fullscreen element*
- *Fullscreen an element*
- *Fullscreen is supported*
- *fully exit fullscreen*
- *unfullscreen a document*

HTML

The following terms are defined in the HTML specification: [HTML]

- *2D context creation algorithm*
- *A browsing context is discarded*
- *A serialization of the bitmap as a file*
- *API value*
- *Active document*
- ***Active element*** being the `activeElement` attribute on `Document`
- *Associated window*
- *`body` element*
- *Boolean attribute*
- *Browsing context*
- *Button* state
- *Buttons*
- *Candidate for constraint validation*
- *Canvas context mode*
- *Checkbox* state
- *Checkedness*
- *Child browsing context*
- *Clean up after running a callback*
- *Clean up after running a script*
- *Close a browsing context*
- *Code entry-point*

- *Cookie-averse `Document` object*
- *Current entry*
- *Dirty checkedness flag*
- *Dirty value flag*
- *Disabled*
- *Document address*
- *Document readiness*
- *Document title*
- *Element contexts*
- *Enumerated attribute*
- *Environment settings object*
- *Event loop*
- *File* state
- *File upload state*
- *Focusing steps*
- *Focusable area*
- *`[[GetOwnProperty]]` of a `Window` object*
- *Hidden* state
- *Image Button* state
- *In parallel*
- *`input` event applies*
- *Selected Files*
- *Joint session history*
- *Mature* navigation.
- *Missing value default state*
- *Mutable*
- *Navigate*
- *Navigator object*
- *Nested browsing context*
- *Origin-clean*
- *An overridden reload*

- ***Parent browsing context***
- ***HTML Pause***
- ***Prepare to run a callback***
- ***Prepare to run a script***
- ***Prompt to unload a document***
- ***Radio Button*** state
- ***Raw value***
- ***Refresh state pragma directive***
- ***Reset algorithm***
- ***Resettable*** element
- ***Resettable element***
- ***Run the animation frame callbacks***
- ***Satisfies its constraints***
- ***Script***
- ***Script execution environment***
- ***Selectedness***
- ***Session history***
- ***Settings object***
- ***Simple dialogs***
- ***Submit Button*** state
- ***Submittable elements***
- ***Suffering from bad input***
- ***Top-level browsing context***
- ***Traverse the history***
- ***Traverse the history by a delta***
- ***Tree order***
- ***unfocusing steps***
- ***User prompt***
- ***Value***
- ***Value mode flag***
- ***Value sanitization algorithm***

- *Window* object
- *WindowProxy* exotic object
- *WorkerNavigator object*
- *setSelectionRange*
- *window.confirm*
- *window.alert*
- *window.prompt*

The HTML specification also defines a number of elements which this specification has special-cased behavior for:

- *a element*
- *area element*
- *canvas element*
- *datalist element*
- *frame element*
- *html element*
- *iframe element*
- *input element*
- *map element*
- *optgroup element*
- *option element*
- *output element*
- *select element*
- *textarea element*

The HTML specification also defines *states* of the input element:

- *Color state*
- *Date state*
- *Email state*
- *Local Date and Time state*
- *Month state*
- *Number state*
- *Password state*

- *Range state*
- *Telephone state*
- *Text and Search state*
- *Time state*
- *URL state*
- *Week state*

The HTML specification also defines a range of different attributes:

- *`canvas`'s height attribute*
- *`canvas`' width attribute*
- *Checked*
- *`multiple` attribute*
- *`readOnly` attribute*
- *`type` attribute*
- *`value` attribute*

The HTML Editing APIs specification defines the following terms: [EDITING]

- *Content editable*
- *Editing host*

The following events are also defined in the HTML specification:

- *`beforeunload`*
- *`change`*
- *`DOMContentLoaded`*
- *`input`*
- *`load`*
- *`pageHide`*
- *`pageShow`*

The "data" URL scheme specification defines the following terms: [RFC2397]

- *`data:` URL*

HTTP and related specifications

To be ***HTTP compliant***, it is supposed that the implementation supports the relevant subsets of [RFC7230], [RFC7231], [RFC7232], [RFC7234], and [RFC7235].

The following terms are defined in the Cookie specification: [RFC6265]

- ***Compute `cookie-string`***
- ***Cookie***
- ***Cookie store***
- ***Receives a cookie***

The following terms are defined in the Hypertext Transfer Protocol (HTTP) Status Code Registry:

- ***Status code registry***

The following terms are defined in the Netscape Navigator Proxy Auto-Config File Format:

- ***Proxy autoconfiguration***

The specification uses ***URI Templates***. [URI-TEMPLATE]

Infra

The following terms are defined in the Infra standard: [INFRA]

- ***ASCII lowercase***
- ***JavaScript string's length***
- ***queue***

Interaction

The following terms are defined in the Page Visibility Specification [PAGE-VISIBILITY]

- ***Visibility state `hidden`***
- ***Visibility state*** being the `visibilityState` attribute on Document
- ***Visibility state `visible`***

Selenium

The following functions are defined within the Selenium project, at revision `1721e627e3b5ab90a06e82df1b088a33a8d11c20`.

- ***`bot.dom.getVisibleText`***
- ***`bot.dom.isShown`***

Styling

The following terms are defined in the CSS Values and Units Module Level 3 specification: [CSS3-VALUES]

- ***CSS pixels***

The following properties are defined in the CSS Basic Box Model Level 3 specification: [CSS3-BOX]

- The ***visibility*** property

The following terms are defined in the CSS Device Adaptation Module Level 1 specification: [CSS-DEVICE-ADAPT]

- ***Initial viewport***, sometimes here referred to as the *viewport*.

The following properties are defined in the CSS Display Module Level 3 specification: [CSS3-DISPLAY]

- The ***display*** property

The following terms are defined in the Geometry Interfaces Module Level 1 specification: [GEOMETRY-1]

- ***DOMRect***
- ***Rectangle***
- ***Rectangle height dimension***
- ***Rectangle width dimension***
- ***Rectangle x coordinate***
- ***Rectangle y coordinate***

The following terms are defined in the CSS Cascading and Inheritance Level 4 specification: [CSS-CASCADE-4]

- ***Computed value***

The following terms are defined in the CSS Object Model: [CSSOM]:

- ***Resolved value***

The following functions are defined in the CSSOM View Module: [CSSOM-VIEW]:

- ***getBoundingClientRect***
- ***Element from point*** as elementFromPoint()
- ***Elements from point*** as elementsFromPoint()
- ***getClientRects***
- ***innerHeight***
- ***innerWidth***

- ***moveTo(x, y)***
- ***offsetLeft***
- ***offsetParent***
- ***offsetTop***
- ***outerHeight***
- ***outerWidth***
- ***screenX***
- ***screenY***
- ***scrollX***
- ***scrollY***
- ***scrollIntoView***
- ***`ScrollIntoViewOptions`***
- ***Logical scroll position `"block"`***
- ***Logical scroll position `"inline"`***

SOCKS Proxy and related specification:

To be ***SOCKS Proxy*** and ***SOCKS authentication*** compliant, it is supposed that the implementation supports the relevant subsets of [RFC1928] and [RFC1929].

Unicode

The following terms are defined in the standard: [Unicode]

- ***Code Point***
- ***Extended grapheme cluster***

Unicode Standard Annex #29

The following terms are defined in the standard: [UAX29]

- ***Grapheme cluster boundaries***

Unicode Standard Annex #44

The following terms are defined in the standard: [UAX44]

- ***Unicode character property***

URLs

The following terms are defined in the WHATWG URL standard: [URL]

- ***Absolute URL***
- ***Absolute URL with fragment***

- *Default port*
- *Domain*
- *Host*
- *Includes credentials*
- *IPv4 address*
- *IPv6 address*
- *Is special*
- *Path-absolute URL*
- *Path*
- *Port*
- *URL*
- *URL serializer*

Web IDL

The IDL fragments in this specification must be interpreted as required for conforming IDL fragments, as described in the Web IDL specification. [WEBIDL]

- `DOMException`
- *Sequence*
- *Supported property indices*
- `SyntaxError`

Promises Guide

The following terms are defined in the Promises Guide. [PROMISES-GUIDE]

- *A new promise*
- *Promise-calling*
- *reject*
- *resolve*

XML Namespaces

The following terms are defined in the Namespaces in XML [XML-NAMES]

- *qualified element name*

XPATH

The following terms are defined in the Document Object Model XPath standard [XPATH]

- `evaluate`

- `ORDERED_NODE_SNAPSHOT_TYPE`
- `snapshotItem`
- `XPathException`

2. Design Notes

This section is non-normative

The WebDriver standard attempts to follow a number of design goals:

2.1 Compatibility

This specification is derived from the popular Selenium WebDriver browser automation framework. Selenium is a long-lived project, and due to its age and breadth of use it has a wide range of expected functionality. This specification uses these expectations to inform its design. Where improvements or clarifications have been made, they have been made with care to allow existing users of Selenium WebDriver to avoid unexpected breakages.

2.2 Simplicity

The largest intended group of users of this specification are software developers and testers writing automated tests and other tooling, such as monitoring or load testing, that relies on automating a browser. As such, care has been taken to provide commands that simplify common tasks such as typing into and clicking elements.

2.3 Extensions

WebDriver provides a mechanism for others to define extensions to the protocol for the purposes of automating functionality that cannot be implemented entirely in ECMAScript. This allows other web standards to support the automation of new platform features. It also allows vendors to expose functionality that is specific to their browser.

3. Terminology

In equations, all numbers are integers, addition is represented by “+”, subtraction is represented by “–”, and bitwise OR by “|”. The characters “(” and “)” are used to provide logical grouping in these contexts.

The shorthand ***min***(*value*, *value*[, *value*]) returns the smallest item of two or more values. Conversely, the shorthand ***max***(*value*, *value*[, *value*]) returns the largest item of two or more values.

A ***Universally Unique IDentifier (UUID)*** is a 128 bits long URN that requires no central registration process. [RFC4122]. ***Generating a UUID*** means Creating a UUID From Truly Random or Pseudo-Random Numbers [RFC4122], and converting it to the string representation [RFC4122].

The ***Unix Epoch*** is a value that approximates the number of seconds that have elapsed since the Epoch, as described by The Open Group Base Specifications Issue 7 section 4.15 (IEEE Std 1003.1).

An ***integer*** is a Number that is unchanged under the ToInteger operation.

The ***initial value*** of an ECMAScript property is the value defined by the platform for that property, i.e. the value it would have in the absence of any shadowing by content script.

The ***browser chrome*** is a non-normative term to refer to the representation through which the user interacts with the user agent itself, as distinct from the accessed web content. Examples of ***browser chrome elements*** include, but are not limited to, toolbars (such as the bookmark toolbar), menus (such as the file or context menu), buttons (such as the back and forward buttons), door hangers (such as security and certificate indicators), and decorations (such as operating system widget borders).

4. Interface

The ***webdriver-active flag*** is set to true when the user agent is under remote control. It is initially false.

WebIDL

```
Navigator includes NavigatorAutomationInformation;
```

Note that the `NavigatorAutomationInformation` interface should not be exposed on `WorkerNavigator`.

WebIDL

```
interface mixin NavigatorAutomationInformation {
    readonly attribute boolean webdriver;
};
```

webdriver
: Returns true if webdriver-active flag is set, false otherwise.

> EXAMPLE 1
>
> For web authors (non-normative):
>
> **navigator.webdriver**
> : Defines a standard way for co-operating user agents to inform the document that it is controlled by WebDriver, for example so that alternate code paths can be triggered during automation.

5. Nodes

The WebDriver protocol consists of communication between:

Local end
: The local end represents the client side of the protocol, which is usually in the form of language-specific libraries providing an API on top of the WebDriver protocol. This specification does not place any restrictions on the details of those libraries above the level of the wire protocol.

Remote end
: The remote end hosts the server side of the protocol. Defining the behavior of a remote end in response to the WebDriver protocol forms the largest part of this specification.

For remote ends the standard defines two broad conformance classes, known as ***node types***:

Intermediary node
: Intermediary nodes are those that act as proxies, implementing both the local end and remote end of the protocol. However they are not expected to implement remote end steps directly. All nodes between a specific intermediary node and a local end are said to be ***downstream*** of that node. Conversely, any nodes between a specific intermediary node and an endpoint node are said to be ***upstream***.

Endpoint node
: An endpoint node is the final remote end in a chain of nodes that is not an intermediary node. The endpoint node is implemented by a user agent or a similar program.

All remote end node types must be black-box indistinguishable from a remote end, from the point of view of local end, and so are bound by the requirements on a remote end in terms of the wire protocol.

A remote end's ***readiness state*** is the value corresponding to the first matching statement:

↪ **the maximum active sessions is equal to the length of the list of active sessions**
↪ **the node is an intermediary node and is known to be in a state in which attempting to create a new session would fail**
False

↪ **Otherwise**
True

> If the intermediary node is a multiplexer that manages multiple endpoint nodes, this might indicate its ability to purvey more sessions, for example if it has hit its maximum capacity.

6. Protocol

WebDriver remote ends must provide an HTTP compliant wire protocol where the endpoints map to different commands.

As this standard only defines the remote end protocol, it puts no demands to how local ends should be implemented. Local ends are only expected to be compatible to the extent that they can speak the remote end's protocol; no requirements are made upon their exposed user-facing API.

6.1 Algorithms

Various parts of this specification are written in terms of step-by-step algorithms. The details of these algorithms do not have any normative significance; implementations are free to adopt any implementation strategy that produces equivalent output to the specification. In particular, algorithms in this document are optimised for readability rather than performance.

Where algorithms that return values are fallible, they are written in terms of returning either ***success*** or ***error***. A success value has an associated *data* field which encapsulates the value returned, whereas an error response has an associated error code.

When calling a fallible algorithm, the construct "Let *result* be the result of ***trying*** to call *algorithm*" is equivalent to

1. Let *temp* be the result of calling *algorithm*.
2. If *temp* is an error return *temp*, otherwise let *result* be *temp*'s *data* field.

The result of ***getting a property*** with argument *name* is defined as being the same as the result of calling Object.[[GetOwnProperty]](*name*).

Setting a property with arguments *name* and *value* is defined as being the same as calling Object.[[Put]](*name*, *value*).

The result of ***JSON serialization*** with *object* of type JSON Object is defined as the result of calling `JSON.[[Stringify]]`(*object*).

The result of ***JSON deserialization*** with *text* is defined as the result of calling `JSON.[[Parse]]`(*text*).

6.2 Commands

The WebDriver protocol is organised into commands. Each HTTP request with a method and template defined in this specification represents a single ***command***, and therefore each command produces a single HTTP response. In response to a command, a remote end will run a series of actions against the browser.

Each command defined in this specification has an associated list of ***remote end steps***. This provides the sequence of actions that a remote end takes when it receives a particular command.

6.3 Processing Model

The remote end is an HTTP server reading requests from the client and writing responses, typically over a TCP socket. For the purposes of this specification we model the data transmission between a particular local end and remote end with a ***connection*** to which the remote end may ***write bytes*** and ***read bytes***. However the exact details of how this connection works and how it is established are out of scope.

After such a connection has been established, a remote end *MUST* run the following steps:

1. Read bytes from the connection until a complete HTTP request can be constructed from the data. Let *request* be a request constructed from the received data, according to the requirements of [RFC7230]. If it is not possible to construct a complete HTTP request, the remote end must either close the connection, return an HTTP response with status code 500, or return an error with error code unknown error.

2. Let *request match* be the result of the algorithm to match a request with *request*'s method and URL as arguments.

3. If *request match* is of type error, send an error with *request match*'s error code and jump to step 1.

   Otherwise, let *command* and *command parameters* be *request match*'s data. Let *url variables* be a url variables dictionary mapping the *command parameters* to their corresponding values.

4. If *session id* is among the variables defined by *command parameters*:

> NOTE
>
> This condition is intended to exclude the New Session and Status commands and any extension commands which do not operate on a particular session.

1. Let *session id* be the corresponding variable from *command parameters*.
2. Let the current session be the session with ID *session id* in the list of active sessions, or null if there is no such matching session.
3. If the current session is null send an error with error code invalid session id, then jump to step 1 in this overall algorithm.
4. If the current session is not null:
    1. Enqueue *request* in the current session's request queue.
    2. Wait until the first element in the current session's request queue is *request*:
    3. Dequeue *request* from the current session's request queue.
    4. If the list of active sessions no longer contains the current session, set the current session to null.
5. If *request*'s method is POST:
    1. Let *parse result* be the result of parsing as JSON with *request*'s body as the argument. If this process throws an exception, return an error with error code invalid argument and jump back to step 1 in this overall algorithm.
    2. If *parse result* is not an Object, send an error with error code invalid argument and jump back to step 1 in this overall algorithm.

       Otherwise, let *parameters* be *parse result*.

   Otherwise, let *parameters* be null.
6. Wait for navigation to complete. If this returns an error return its value and jump to step 1 in this overall algorithm, otherwise continue.
7. Let *response result* be the return value obtained by running the remote end steps for *command* with an argument named *url variables* whose value is *url variables* and an additional argument named *parameters* whose value is *parameters*.
8. If *response result* is an error, send an error with error code equal to *response result*'s error code and jump back to step 1 in this overall algorithm.

Otherwise, if *response result* is a success, let *response data* be *response result*'s data.

9. Send a response with status 200 and *response data*.

10. Jump to step 1.

When required to ***send an error***, with *error code* and an optional *error data* dictionary, a remote end must run the following steps:

1. Let *http status* and *name* be the error response data for *error code*.

2. Let *message* be an implementation-defined string containing a human-readable description of the reason for the error.

3. Let *stacktrace* be an implementation-defined string containing a stack trace report of the active stack frames at the time when the error occurred.

   Let *body* be a new JSON Object initialised with the following properties:

   `"error"`
   : *name*

   `"message"`
   : *message*

   `"stacktrace"`
   : *stacktrace*

4. If the error data dictionary contains any entries, set the "`data`" field on *body* to a new JSON Object populated with the dictionary.

5. Send a response with *status* and *body* as arguments.

When required to ***send a response***, with arguments *status* and *data*, a remote end must run the following steps:

1. Let *response* be a new response.

2. Set *response*'s HTTP status to *status*, and status message to the string corresponding to the description of *status* in the status code registry.

3. Set the *response*'s header with name and value with the following values:

   `Content-Type`
   : "`application/json; charset=utf-8`"

   `Cache-Control`
   : "`no-cache`"

4. Let *response*'s body be a JSON Object with a key "`value`" set to the JSON serialization of *data*.

5. Let *response bytes* be the byte sequence resulting from serializing *response* according to the rules in [RFC7230].

6. Write *response bytes* to the connection.

A ***url variable*** dictionary is defined as the mapping of a command's URI template variable names to their corresponding values.

6.4 Routing Requests

Request routing is the process of going from a HTTP request to the series of steps needed to implement the command represented by that request.

A remote end has an associated ***URL prefix***, which is used as a prefix on all WebDriver-defined URLs on that remote end. This must either be undefined or a path-absolute URL.

> EXAMPLE 2
>
> For example a remote end wishing to run alongside other services on `example.com` might set its URL prefix to `/wd` so that a new session command would be invoked by sending a POST request to `/wd/session`, rather than `/session`.

In order to ***match a request*** given a *method* and *URL*, the following steps must be taken:

1. Let *endpoints* be a list containing each row in the table of endpoints.

2. Remove each entry from *endpoints* for which the concatenation of the URL prefix and the entry's URI template does not match *URL*'s path.

3. If there are no entries in *endpoints*, return error with error code unknown command.

4. Remove each entry in *endpoints* for which the *method* column is not an exact case-sensitive match for *method*.

5. If there are no entries in *endpoints*, return error with error code unknown method.

6. There is now exactly one entry in *endpoints*; let *entry* be this entry.

7. Let *parameters* be the result of extracting the variables from *URL* using *entry*'s URI template.

8. Let *command* be *entry*'s command.

9. Return success with data *command* and *parameters*.

6.5 List of Endpoints

The following ***table of endpoints*** lists the method and URI template for each endpoint node command. Extension commands are implicitly appended to this table.

Method	URI Template	Command
POST	/session	New Session
DELETE	/session/{*session id*}	Delete Session
GET	/status	Status
GET	/session/{*session id*}/timeouts	Get Timeouts
POST	/session/{*session id*}/timeouts	Set Timeouts
POST	/session/{*session id*}/url	Navigate To
GET	/session/{*session id*}/url	Get Current URL
POST	/session/{*session id*}/back	Back
POST	/session/{*session id*}/forward	Forward
POST	/session/{*session id*}/refresh	Refresh
GET	/session/{*session id*}/title	Get Title
GET	/session/{*session id*}/window	Get Window Handle
DELETE	/session/{*session id*}/window	Close Window
POST	/session/{*session id*}/window	Switch To Window
GET	/session/{*session id*}/window/handles	Get Window Handles
POST	/session/{*session id*}/frame	Switch To Frame
POST	/session/{*session id*}/frame/parent	Switch To Parent Frame
GET	/session/{*session id*}/window/rect	Get Window Rect
POST	/session/{*session id*}/window/rect	Set Window Rect
POST	/session/{*session id*}/window/maximize	Maximize Window
POST	/session/{*session id*}/window/minimize	Minimize Window
POST	/session/{*session id*}/window/fullscreen	Fullscreen Window
GET	/session/{*session id*}/element/active	Get Active Element
POST	/session/{*session id*}/element	Find Element
POST	/session/{*session id*}/elements	Find Elements

POST	/session/{*session id*}/element/{element id}/element	Find Element From Element
POST	/session/{*session id*}/element/{element id}/elements	Find Elements From Element
GET	/session/{*session id*}/element/{*element id*}/selected	Is Element Selected
GET	/session/{*session id*}/element/{*element id*}/attribute/{*name*}	Get Element Attribute
GET	/session/{*session id*}/element/{*element id*}/property/{*name*}	Get Element Property
GET	/session/{*session id*}/element/{*element id*}/css/{*property name*}	Get Element CSS Value
GET	/session/{*session id*}/element/{*element id*}/text	Get Element Text
GET	/session/{*session id*}/element/{*element id*}/name	Get Element Tag Name
GET	/session/{*session id*}/element/{*element id*}/rect	Get Element Rect
GET	/session/{*session id*}/element/{*element id*}/enabled	Is Element Enabled
POST	/session/{*session id*}/element/{*element id*}/click	Element Click
POST	/session/{*session id*}/element/{*element id*}/clear	Element Clear
POST	/session/{*session id*}/element/{*element id*}/value	Element Send Keys
GET	/session/{*session id*}/source	Get Page Source
POST	/session/{*session id*}/execute/sync	Execute Script
POST	/session/{*session id*}/execute/async	Execute Async Script
GET	/session/{*session id*}/cookie	Get All Cookies
GET	/session/{*session id*}/cookie/{*name*}	Get Named Cookie
POST	/session/{*session id*}/cookie	Add Cookie
DELETE	/session/{*session id*}/cookie/{*name*}	Delete Cookie
DELETE	/session/{*session id*)/cookie	Delete All Cookies
POST	/session/{*session id*}/actions	Perform Actions
DELETE	/session/{*session id*}/actions	Release Actions
POST	/session/{*session id*}/alert/dismiss	Dismiss Alert
POST	/session/{*session id*}/alert/accept	Accept Alert
GET	/session/{*session id*}/alert/text	Get Alert Text
POST	/session/{*session id*}/alert/text	Send Alert Text
GET	/session/{*session id*}/screenshot	Take Screenshot

GET	/session/{*session id*}/element/{*element id*}/screenshot	Take Element Screenshot

6.6 Handling Errors

Errors are represented in the WebDriver protocol by an HTTP response with an HTTP status in the 4xx or 5xx range, and a JSON body containing details of the error. The body is a JSON Object and has a field named "`value`" whose value is an object bearing three, and sometimes four, fields:

- "`error`", containing a string indicating the error code.
- "`message`", containing an implementation-defined string with a human readable description of the kind of error that occurred.
- "`stacktrace`", containing an implementation-defined string with a stack trace report of the active stack frames at the time when the error occurred.
- Optionally "`data`", which is a JSON Object with additional error data helpful in diagnosing the error.

EXAMPLE 3

A `GET` request to `/session/1234/url`, where `1234` is not the session id of a current session would return an HTTP response with the status 404 and a body of the form:

```
{
        "value": {
                "error": "invalid session id",
                "message": "No active session with ID 1234",
                "stacktrace": ""
        }
}
```

Certain commands may also annotate errors with additional error data. Notably, this is the case for commands which invoke the user prompt handler, where the user prompt message may be included in a "`text`" field:

```
{
        "value": {
                "error": "unexpected alert open",
                "message": "",
                "stacktrace": "",
                "data": {
                        "text": "Message from window.alert"
                }
        }
}
```

The following table lists each ***error code***, its associated HTTP status, JSON `error` code, and a non-normative description of the error. The ***error response data*** for a particular error code is the values of the *HTTP Status* and *JSON Error Code* columns for the row corresponding to that error code.

Error Code	HTTP Status	JSON Error Code	Description
element click intercepted	400	`element click intercepted`	The Element Click command could not be completed because the element receiving the events is obscuring the element that was requested clicked.
element not interactable	400	`element not interactable`	A command could not be completed because the element is not pointer- or keyboard interactable.
insecure certificate	400	`insecure certificate`	Navigation caused the user agent to hit a certificate warning, which is usually the result of an expired or

			invalid TLS certificate.
invalid argument	400	`invalid argument`	The arguments passed to a command are either invalid or malformed.
invalid cookie domain	400	`invalid cookie domain`	An illegal attempt was made to set a cookie under a different domain than the current page.
invalid element state	400	`invalid element state`	A command could not be completed because the element is in an invalid state, e.g. attempting to clear an element that isn't both editable and resettable.
invalid selector	400	`invalid selector`	Argument was an invalid selector.
invalid session id	404	`invalid session id`	Occurs if the given session id is not in the list of active sessions, meaning the session either does not exist or that it's not active.
javascript error	500	`javascript error`	An error occurred while executing JavaScript supplied by the user.
move target out of bounds	500	`move target out of bounds`	The target for mouse interaction is not in the browser's viewport and cannot be brought into that viewport.
no such alert	404	`no such alert`	An attempt was made to operate on a modal dialog when one was not open.
no such cookie	404	`no such cookie`	No cookie matching the given path name was found amongst the associated cookies of the current browsing context's active document.
no such element	404	`no such element`	An element could not be located on the page using the given search parameters.
no such frame	404	`no such frame`	A command to switch to a frame could not be satisfied because the frame could not be found.
no such window	404	`no such window`	A command to switch to a window could not be satisfied because the window could not be found.
script timeout	408	`script timeout`	A script did not complete before its timeout expired.
session not created	500	`session not created`	A new session could not be created.
stale element reference	404	`stale element reference`	A command failed because the referenced element is no longer attached to the DOM.

timeout	408	`timeout`	An operation did not complete before its timeout expired.
unable to set cookie	500	`unable to set cookie`	A command to set a cookie's value could not be satisfied.
unable to capture screen	500	`unable to capture screen`	A screen capture was made impossible.
unexpected alert open	500	`unexpected alert open`	A modal dialog was open, blocking this operation.
unknown command	404	`unknown command`	A command could not be executed because the remote end is not aware of it.
unknown error	500	`unknown error`	An unknown error occurred in the remote end while processing the command.
unknown method	405	`unknown method`	The requested command matched a known URL but did not match an method for that URL.
unsupported operation	500	`unsupported operation`	Indicates that a command that should have executed properly cannot be supported for some reason.

An ***error data*** dictionary is a mapping of string keys to JSON serializable values that can optionally be included with error objects.

6.7 Protocol Extensions

Using the terminology defined in this section, others may define additional commands that seamlessly integrate with the standard protocol. This allows vendors to expose functionality that is specific to their user agent, and it also allows other web standards to define commands for automating new platform features.

Commands defined in this way are called ***extension commands*** and behave no differently than other commands; each has a dedicated HTTP endpoint and a set of remote end steps.

Each extension command has an associated ***extension command URI Template*** that is a URI Template string, and which should bear some resemblance to what the command performs. This value, along with the HTTP method and extension command, is added to the table of endpoints and thus follows the same rules for request routing as that of other built-in commands.

In order to avoid potential resource conflicts with other implementations, vendor-specific extension command URI Templates must begin with one or more path segments which uniquely identifies the vendor and UA. It is suggested that vendors use their vendor prefixes without additional characters

as outlined in [CSS21], notably in section 4.1.2.2 on vendor keywords, as the name for this path element, and include a vendor-chosen UA identifier.

> NOTE
> If the extension command URI Template includes a variable named *session id*, the value of this variable will be used to define the current session during command processing.

> EXAMPLE 4
>
> This might lead to a URL of the form `/session/5d376174-36f0-11e5-9b9a-6bdf200a3f7f/ms/edge/context`, where `session/{session id}` associates the request with the specified session, `ms/edge` identifies the command as specific to the Edge browser distributed by Microsoft, and `context` describes the functionality that, in the context of Edge, allows a local end to switch between browser-specific contexts. Requesting this URL will call the extension command's remote end steps.

Remote ends may also introduce ***extension capabilities*** that are extra capabilities used to provide configuration or fulfill other vendor-specific needs. Extension capabilities' key must contain a ":" (colon) character, denoting an implementation specific namespace. The value can be arbitrary JSON types.

As with extension commands, it is suggested that the key used to denote the extension capability namespace is based on the vendor keywords listed in [CSS21] and precedes the first ":" character in the string.

EXAMPLE 5

Extension capabilities are typically used to provide UA or intermediary node specific configuration that is not handled by the table of standard capabilities.

An example new session request body might look like this:

```
{
  "capabilities": {
    "alwaysMatch": {
      // browser specific configuration
      "<prefix>:browserOptions": {
        "binary": "/usr/bin/browser-binary",
        "args": ["--start-page=http://example.com"],
      }
    }
  }
}
```

7. Capabilities

WebDriver ***capabilities*** are used to communicate the features supported by a given implementation. The local end may use capabilities to define which features it requires the remote end to satisfy when creating a new session. Likewise, the remote end uses capabilities to describe the full feature set for a session.

The following ***table of standard capabilities*** enumerates the capabilities each implementation *MUST* support. An implementation *MAY* define additional extension capabilities.

EXAMPLE 6

As an example, Mozilla could elect to hide new features behind capabilities with a "moz:" prefix:

```
{
        "browserName": "firefox",
        "browserVersion": "1234",
        "moz:experimental-webdriver": true
}
```

Capability	Key	Value Type	Description
Browser name	`"browserName"`	string	Identifies the user agent.
Browser version	`"browserVersion"`	string	Identifies the version of the user agent.
Platform name	`"platformName"`	string	Identifies the operating system of the endpoint node.
Accept insecure TLS certificates	`"acceptInsecureCerts"`	boolean	Indicates whether untrusted and self-signed TLS certificates are implicitly trusted on navigation for the duration of the session.
Page load strategy	`"pageLoadStrategy"`	string	Defines the current session's page load strategy.
Proxy configuration	`"proxy"`	JSON Object	Defines the current session's proxy configuration.
Window dimensioning/positioning	`"setWindowRect"`	boolean	Indicates whether the remote end supports all of the commands in Resizing and Positioning Windows.
Session timeouts configuration	`"timeouts"`	JSON Object	Describes the timeouts imposed on certain session operations.
Unhandled prompt behavior	`"unhandledPromptBehavior"`	string	Describes the current session's user prompt handler.

7.1 Proxy

The ***proxy configuration*** capability is a JSON Object nested within the primary capabilities. Implementations may define additional proxy configuration options, but they must not alter the semantics of those listed below.

Key	Value Type	Description	Valid values
proxyType	string	Indicates the type of proxy configuration.	"pac", "direct", "autodetect", "system", or "manual".
proxyAutoconfigUrl	string	Defines the URL for a proxy auto-config file if proxyType is equal to "pac".	Any URL.
ftpProxy	string	Defines the proxy host for FTP traffic when the proxyType is "manual".	A host and optional port for scheme "ftp".
httpProxy	string	Defines the proxy host for HTTP traffic when the proxyType is "manual".	A host and optional port for scheme "http".
noProxy	array	Lists the address for which the proxy should be bypassed when the proxyType is "manual".	A List containing any number of Strings.
sslProxy	string	Defines the proxy host for encrypted TLS traffic when the proxyType is "manual".	A host and optional port for scheme "https".
socksProxy	string	Defines the proxy host for a SOCKS proxy when the proxyType is "manual".	A host and optional port with an undefined scheme.
socksVersion	number	Defines the SOCKS proxy version when the proxyType is "manual".	Any integer between 0 and 255 inclusive.

A ***host and optional port*** for a *scheme* is defined as being a valid host, optionally followed by a colon and a valid port. The host may include credentials. If the port is omitted and *scheme* has a default port, this is the implied port. Otherwise, the port is left undefined.

A proxyType of "direct" indicates that the browser should not use a proxy at all.

A proxyType of "system" indicates that the browser should use the various proxies configured for the underlying Operating System.

A `proxyType` of "`autodetect`" indicates that the proxy to use should be detected in an implementation-specific way.

The remote end steps to ***deserialize as a proxy*** argument *parameter* are:

1. If *parameter* is not a JSON Object return an error with error code invalid argument.
2. Let *proxy* be a new, empty proxy configuration object.
3. For each enumerable own property in *parameter* run the following substeps:
    1. Let *key* be the name of the property.
    2. Let *value* be the result of getting a property named *name* from *parameter*.
    3. If there is no matching `key` for *key* in the proxy configuration table return an error with error code invalid argument.
    4. If *value* is not one of the `valid values` for that `key`, return an error with error code invalid argument.
    5. Set a property *key* to *value* on *proxy*.
4. If *proxy* does not have an own property for "`proxyType`" return an error with error code invalid argument.
5. If the result of getting a property named *proxyType* from *proxy* equals "`pac`", and *proxy* does not have an own property for "`proxyAutoconfigUrl`" return an error with error code invalid argument.
6. If *proxy* has an own property for "`socksProxy`" and does not have an own property for "`socksVersion`" return an error with error code invalid argument.
7. Return success with data *proxy*.

A ***proxy configuration object*** is a JSON Object where each of its own properties matching keys in the proxy configuration meets the validity criteria for that key.

7.2 Processing Capabilities

To ***process capabilities*** with argument *parameters*, the endpoint node must take the following steps:

1. Let *capabilities request* be the result of getting the property "`capabilities`" from *parameters*.

1. If *capabilities request* is not a JSON object, return error with error code invalid argument.

2. Let *required capabilities* be the result of getting the property "`alwaysMatch`" from *capabilities request*.

    1. If *required capabilities* is undefined, set the value to an empty JSON Object.

    2. Let *required capabilities* be the result of trying to validate capabilities with argument *required capabilities*.

3. Let *all first match capabilities* be the result of getting the property "`firstMatch`" from *capabilities request*.

    1. If *all first match capabilities* is undefined, set the value to a JSON List with a single entry of an empty JSON Object.

    2. If *all first match capabilities* is not a JSON List with one or more entries, return error with error code invalid argument.

4. Let *validated first match capabilities* be an empty JSON List.

5. For each *first match capabilities* corresponding to an indexed property in *all first match capabilities*:

    1. Let *validated capabilities* be the result of trying to validate capabilities with argument *first match capabilities*.

    2. Append *validated capabilities* to *validated first match capabilities*.

6. Let *merged capabilities* be an empty List.

7. For each *first match capabilities* corresponding to an indexed property in *validated first match capabilities*:

    1. Let *merged* be the result of trying to merge capabilities with *required capabilities* and *first match capabilities* as arguments.

    2. Append *merged* to *merged capabilities*.

8. For each *capabilities* corresponding to an indexed property in *merged capabilities*:

    1. Let *matched capabilities* be the result of trying to match capabilities with *capabilities* as an argument.

    2. If *matched capabilities* is not null, return *matched capabilities*.

9. Return success with data null.

When required to ***validate capabilities*** with argument *capability*:

1. If *capability* is not a JSON Object return an error with error code invalid argument.
2. Let *result* be an empty JSON Object.
3. For each enumerable own property in *capability*, run the following substeps:
    1. Let *name* be the name of the property.
    2. Let *value* be the result of getting a property named *name* from *capability*.
    3. Run the substeps of the first matching condition:

    ↪ ***value* is null**
    Let *deserialized* be set to null.

    ↪ ***name* equals "`acceptInsecureCerts`"**
    If *value* is not a boolean return an error with error code invalid argument. Otherwise, let *deserialized* be set to *value*

    ↪ ***name* equals "`browserName`"**
    ↪ ***name* equals "`browserVersion`"**
    ↪ ***name* equals "`platformName`"**
    If *value* is not a string return an error with error code invalid argument. Otherwise, let *deserialized* be set to *value*.

    ↪ ***name* equals "`pageLoadStrategy`"**
    Let *deserialized* be the result of trying to deserialize as a page load strategy with argument *value*.

    ↪ ***name* equals "`proxy`"**
    Let *deserialized* be the result of trying to deserialize as a proxy with argument *value*.

    ↪ ***name* equals "`timeouts`"**
    Let *deserialized* be the result of trying to deserialize as a timeout with argument *value*.

    ↪ ***name* equals "`unhandledPromptBehavior`"**
    Let *deserialized* be the result of trying to deserialize as an unhandled prompt behavior with argument *value*.

    ↪ ***name* is the key of an extension capability**
    Let *deserialized* be the result of trying to deserialize *value* in an implementation-specific way.

    ↪ **The remote end is an endpoint node**
    Return an error with error code invalid argument.

4. If *deserialized* is not null, set a property on *result* with name *name* and value *deserialized*.

4. Return success with data *result*.

When ***merging capabilities*** with JSON Object arguments *primary* and *secondary*, an endpoint node must take the following steps:

1. Let *result* be a new JSON Object.
2. For each enumerable own property in *primary*, run the following substeps:
    1. Let *name* be the the name of the property.
    2. Let *value* be the the result of getting a property named *name* from *primary*.
    3. Set a property on *result* with name *name* and value *value*.
3. If *secondary* is undefined, return *result*.
4. For each enumerable own property in *secondary*, run the following substeps:
    1. Let *name* be the the name of the property.
    2. Let *value* be the the result of getting a property named *name* from *secondary*.
    3. Let *primary value* be the result of getting the property *name* from *primary*.
    4. If *primary value* is not undefined, return an error with error code invalid argument.
    5. Set a property on *result* with name *name* and value *value*.
5. Return *result*.

> NOTE
>
> The algorithm outlined in matching capabilities blithely ignores real-world problems that make implementation less than perfectly straightforward, particularly since capabilities can interact in unforeseen ways.
>
> As an example, an implementation could have a capability that gives the path to the browser binary to use. This could cause both `browserName` and `browserVersion` to be impossible to match against until the browser process is started.

When ***matching capabilities*** with JSON Object argument *capabilities*, an endpoint node must take the following steps:

1. Let *matched capabilities* be a JSON Object with the following entries:

   "browserName"
   : Lowercase name of the user agent as a string.

   "browserVersion"
   : The user agent version, as a string.

   "platformName"
   : Lowercase name of the current platform as a string.

   "acceptInsecureCerts"
   : Boolean initially set to `false`, indicating the session will not implicitly trust untrusted or self-signed TLS certificates on navigation.

   "setWindowRect"
   : Boolean indicating whether the remote end supports all of the commands in Resizing and Positioning Windows.

2. Optionally add extension capabilities as entries to *matched capabilities*. The values of these may be elided, and there is no requirement that all extension capabilities be added.

   > NOTE
   >
   > This allows a remote end to add information that might be useful to a local end without unnecessarily bloating the response sent back to the user with (e.g.) an entire browser profile.
   >
   > For example, an implementation could choose to indicate that a screenshot will be taken when returning an error by setting the capability `se:screenshot-on-error` to `true`.

3. For each *name* and *value* corresponding to *capability*'s own properties:

   1. Run the substeps of the first matching *name*:

      ↪ **"browserName"**
      : If *value* is not a string equal to the "`browserName`" entry in *matched capabilities*, return success with data null.

        > NOTE
        >
        > There is a chance the remote end will need to start a browser process to correctly determine the `browserName`. Lightweight checks are preferred before this is done.

↪ "`browserVersion`"

Compare *value* to the "`browserVersion`" entry in *matched capabilities* using an implementation-defined comparison algorithm. The comparison is to accept a *value* that places constraints on the version using the "`<`", "`<=`", "`>`", and "`>=`" operators.

If the two values do not match, return success with data null.

> NOTE
>
> Version comparison is left as an implementation detail since each user agent will likely have conflicting methods of encoding the user agent version, and standardizing these schemes is beyond the scope of this standard.

> NOTE
>
> There is a chance the remote end will need to start a browser process to correctly determine the `browserVersion`. Lightweight checks are preferred before this is done.

↪ "`platformName`"

If *value* is not a string equal to the "`platformName`" entry in *matched capabilities*, return success with data null.

NOTE

The following platform names are in common usage with well-understood semantics and, when matching capabilities, greatest interoperability can be achieved by honoring them as valid synonyms for well-known Operating Systems:

Key	System
`"linux"`	Any server or desktop system based upon the Linux kernel.
`"mac"`	Any version of Apple's macOS.
`"windows"`	Any version of Microsoft Windows, including desktop and mobile versions.

This list is not exhaustive.

When returning capabilities from New Session, it is valid to return a more specific `platformName`, allowing users to correctly identify the Operating System the WebDriver implementation is running on.

↪ **`"acceptInsecureCerts"`**

If *value* is `true` and the endpoint node does not support insecure TLS certificates, return success with data null.

NOTE

If the endpoint node does not support insecure TLS certificates and this is the reason no match is ultimately made, it is useful to provide this information to the local end.

↪ **`"proxy"`**

If the endpoint node does not allow the proxy it uses to be configured, or if the proxy configuration defined in *value* is not one that passes the endpoint node's implementation-specific validity checks, return success with data null.

> NOTE
>
> A local end would only send this capability if it expected it to be honored and the configured proxy used. The intent is that if this is not possible a new session will not be established.

↪ **Otherwise**

If *name* is the key of an extension capability,

let *value* be the result of trying implementation-specific steps to match on *name* with *value*. If the match is not successful, return success with data null.

2. Set a property on *matched capabilities* with name *name* and value *value*.

4. Return success with data *matched capabilities*.

8. Sessions

A session is equivalent to a single instantiation of a particular user agent, including all its child browsers. WebDriver gives each session a unique session ID that can be used to differentiate one session from another, allowing multiple user agents to be controlled from a single HTTP server, and allowing sessions to be routed via a multiplexer (known as an intermediary node).

A WebDriver ***session*** represents the connection between a local end and a specific remote end.

A session is started when a New Session is invoked. It is an error to send any commands before starting a session, or to continue to send commands after the session has been closed. Maintaining session continuity between commands to the remote end requires passing a session ID.

A session is torn down at some later point; either explicitly by invoking Delete Session, or implicitly when Close Window is called at the last remaining top-level browsing context.

An intermediary node will maintain an ***associated session*** for each active session. This is the session on the upstream neighbor that is created when the intermediary node executes the New Session command. Closing a session on an intermediary node will also close the session of the associated session.

All commands, except New Session and Status, have an associated ***current session***, which is the session in which that command will run.

A remote end has an associated list of ***active sessions***, which is a list of all sessions that are currently started. A remote end that is not an intermediary node has at most one active session at a given time.

A remote end has an associated ***maximum active sessions*** (an integer) that defines the number of active sessions that are supported. This may be “unlimited” for intermediary nodes, but must be exactly one for a remote end that is an endpoint node.

A session has an associated ***session ID*** (a string representation of a UUID) used to uniquely identify this session. Unless stated otherwise it is null.

A session has an associated ***current browsing context***, which is the browsing context against which commands will run.

A session has an associated ***current top-level browsing context***, which is the current browsing context if it itself is a top-level browsing context, and otherwise is the top-level browsing context of the current browsing context.

A session has an associated ***session script timeout*** that specifies a time to wait for scripts to run. If equal to null then session script timeout will be indefinite. Unless stated otherwise it is 30,000 milliseconds.

A session has an associated ***session page load timeout*** that specifies a time to wait for the page loading to complete. Unless stated otherwise it is 300,000 milliseconds.

A session has an associated ***session implicit wait timeout*** that specifies a time to wait in milliseconds for the element location strategy when retrieving elements and when waiting for an element to become interactable when performing element interaction . Unless stated otherwise it is zero milliseconds.

A session has an associated ***page loading strategy***, which is one of the keywords from the table of page load strategies. Unless stated otherwise, it is normal.

A session has an associated ***secure TLS*** state that indicates whether untrusted or self-signed TLS certificates should be trusted for the duration of the WebDriver session. If it is unset, this indicates that certificate- or TLS errors that occur upon navigation should be suppressed. The state can be unset by providing an "`acceptInsecureCerts`" capability with the value true. Unless stated otherwise, it is set.

A session has an associated user prompt handler. Unless stated otherwise it is null.

A session has an associated list of active input sources.

A session has an associated input state table.

A session has an associated input cancel list.

A session has an associated ***request queue*** which is a queue of requests that are currently awaiting processing.

When asked to ***close the session***, a remote end must take the following steps:

1. Perform the following substeps based on the remote end's type:

   ↪ **Remote end is an endpoint node**

   1. Set the webdriver-active flag to false.

   2. An endpoint node must close any top-level browsing contexts associated with the session, without prompting to unload.

   ↪ **Remote end is an intermediary node**

   1. Close the associated session. If this causes an error to occur, complete the remainder of this algorithm before returning the error.

2. Remove the current session from active sessions.

3. Perform any implementation-specific cleanup steps.

4. If an error has occurred in any of the steps above, return the error, otherwise return success with data null.

Closing a session might cause the associated browser process to be killed. It is assumed that any implementation-specific cleanup steps are performed *after* the response has been sent back to the client so that the connection is not prematurely closed.

8.1 *New Session*

HTTP Method	URI Template
POST	/session

The New Session command creates a new WebDriver session with the endpoint node. If the creation fails, a session not created error is returned.

If the remote end is an intermediary node, it *MAY* use the result of the capabilities processing algorithm to route the new session request to the appropriate endpoint node. An intermediary node *MAY* also define extension capabilities to assist in this process, however, these specific capabilities *MUST NOT* be forwarded to the endpoint node.

If the intermediary node requires additional information unrelated to user agent features, it is *RECOMMENDED* that this information be passed as top-level parameters, and not as part of the requested capabilities. An intermediary node *MUST* forward custom, top-level parameters (i.e. non-capabilities) to subsequent remote end nodes.

EXAMPLE 7

An intermediary node might require authentication on creating a new session. This authentication is an argument to the New Session command itself and not the user agent's capabilities. Therefore, the authentication should be passed as a top-level parameter and not embedded in `capabilities`:

```
{
        "user": "alice",
        "password": "hunter2",
        "capabilities": {…}
}
```

However, because an intermediary node cannot forward extension capabilities specific to that implementation to an endpoint node, the following is also permitted by this specification:

```
{
    "capabilities": {
        "alwaysMatch": {
            "cloud:user": "alice",
            "cloud:password": "hunter2",
            "platformName": "linux"
        },
        "firstMatch": [
            {"browserName": "chrome"},
            {"browserName": "edge"}
        ]
    }
}
```

Once all capabilities are merged from this example, an endpoint node would receive New Session capabilities identical to:

```
[
    {"browserName": "chrome", "platformName": "linux"},
    {"browserName": "edge", "platformName": "linux"}
]
```

The remote end steps are:

1. If the maximum active sessions is equal to the length of the list of active sessions, return error with error code session not created.

2. If the remote end is an intermediary node, take implementation-defined steps that either result in returning an error with error code session not created, or in returning a success with data that is isomorphic to that returned by remote ends according to the rest of this algorithm. If an error is not returned, the intermediary node must retain a reference to the session created on the upstream node as the associated session such that commands may be forwarded to this associated session on subsequent commands.

   > NOTE
   >
   > How this is done is entirely up to the implementation, but typically the `sessionId`, and URL and URL prefix of the upstream remote end will need to be tracked.

3. If the maximum active sessions is equal to the length of the list of active sessions, return error with error code session not created.

4. Let *capabilities* be the result of trying to process capabilities with *parameters* as an argument.

5. If *capabilities*'s is null, return error with error code session not created.

6. Let *session id* be the result of generating a UUID.

7. Let *session* be a new session with the session ID of *session id*.

8. Set the current session to *session*.

9. Append *session* to active sessions.

10. Let *body* be a JSON Object initialised with:

    "`sessionId`"
    : *session id*

    "`capabilities`"
    : *capabilities*

11. Initialize the following from *capabilities*:

    1. Let *strategy* be the result of getting property "`pageLoadStrategy`" from *capabilities*.

    2. If *strategy* is a string, set the current session's page loading strategy to *strategy*. Otherwise, set the page loading strategy to *normal* and set a property of *capabilities* with name "`pageLoadStrategy`" and value "`normal`".

    3. Let *proxy* be the result of getting property "`proxy`" from *capabilities* and run the substeps of the first matching statement:

       ↪ ***proxy* is a proxy configuration object**

Take implementation-defined steps to set the user agent proxy using the extracted *proxy* configuration. If the defined proxy cannot be configured return error with error code session not created.

↪ **Otherwise**

Set a property of *capabilities* with name "`proxy`" and a value that is a new JSON Object.

4. Let *timeouts* be the result of getting property "`timeouts`" from *capabilities*.
5. If *timeouts* is a session timeouts configuration object:
    1. If *timeouts* has a numeric property with key "`implicit`", set the current session's session implicit wait timeout to the value of property "`implicit`". Otherwise, set the session implicit wait timeout to 0 (zero) milliseconds.
    2. If *timeouts* has a numeric property with key "`pageLoad`", set the current session's session page load timeout to the value of property "`pageLoad`". Otherwise, set the session page load timeout to 300,000 milliseconds.
    3. If *timeouts* has a numeric property with key "`script`", set the current session's session script timeout to the value of property "`script`". Otherwise, set the session script timeout to 30,000 milliseconds.
6. Let *configured timeouts* be a new JSON Object.
7. Set a property on *configured timeouts* with name "`implicit`" and value equal to the current session's session implicit wait timeout.
8. Set a property on *configured timeouts* with name "`pageLoad`" and value equal to the current session's session page load timeout.
9. Set a property on *configured timeouts* with name "`script`" and value equal to the current session's session script timeout.
10. Set a property on *capabilities* with name "`timeouts`" and value *configured timeouts*.
11. Apply changes to the user agent for any implementation-defined capabilities selected during the capabilities processing step.
12. Set the webdriver-active flag to true.
13. Set the current top-level browsing context for *session* in an implementation-specific way. This should be the top-level browsing context of the UA's current browsing context.

> NOTE
>
> WebDriver implementations typically start a completely new browser instance, but there is no requirement in this specification (or for WebDriver only to be used to automate only web browsers). Implementations might choose to use an existing browser instance, eg. by selecting the window that currently has focus.

14. Set the request queue to a new queue.

15. Return success with data *body*.

8.2 *Delete Session*

HTTP Method	URI Template
DELETE	/session/{*session id*}

The remote end steps are:

1. If the current session is an active session, try to close the session.

2. Return success with data null.

8.3 *Status*

HTTP Method	URI Template
GET	/status

> NOTE
>
> Status returns information about whether a remote end is in a state in which it can create new sessions, but may additionally include arbitrary meta information that is specific to the implementation.
>
> The remote end's readiness state is represented by the `ready` property of the body, which is false if an attempt to create a session at the current time would fail. However, the value true does not guarantee that a New Session command will succeed.

Implementations may optionally include additional meta information as part of the body, but the top-level properties `ready` and `message` are reserved and must not be overwritten.

The remote end steps are:

1. Let *body* be a new JSON Object with the following properties:

   "ready"
   The remote end's readiness state.

   "message"
   An implementation-defined string explaining the remote end's readiness state.

2. Return success with data *body*.

8.4 *Get Timeouts*

HTTP Method	URI Template
GET	/session/{*session id*}/timeouts

The remote end steps are:

1. Let *body* be a new JSON Object with the following properties:

   "script"
   session script timeout

   "pageLoad"
   session page load timeout

   "implicit"
   session implicit wait timeout

2. Return success with data *body*.

8.5 *Set Timeouts*

HTTP Method	URI Template
POST	/session/{*session id*}/timeouts

The following ***table of session timeouts*** enumerates the different timeout types that may be set using the Set Timeouts command. The keywords given in the first column maps to the timeout type given in the second.

Keyword	Type	Description
"script"	session script timeout	Determines when to interrupt a script that is being evaluated.

"pageLoad"	session page load timeout	Provides the timeout limit used to interrupt navigation of the browsing context.
"implicit"	session implicit wait timeout	Gives the timeout of when to abort locating an element.

A ***session timeouts configuration*** is a JSON Object. It consists of the properties named after the keys in the table of session timeouts.

The steps to ***deserialize as a timeout*** with argument *parameters* are:

1. If *parameters* is not a JSON Object, return error with error code invalid argument.
2. For each enumerable own property in *parameters*, run the following substeps:
    1. Let *name* be the name of the property.
    2. Let *value* be the result of getting a property named *name* from *parameters*.
    3. Find the *timeout type* from the table of session timeouts by looking it up by its keyword *key*.

       If no keyword matches *key*, return error with error code invalid argument.
    4. If *value* is not an integer, or it is less than 0 or greater than the maximum safe integer, return error with error code invalid argument.
3. Return success with data *parameters*.

The remote end steps of the Set Timeouts command are:

1. Let *parameters* be the result of trying to deserialize as a timeout *parameters*.
2. For each enumerable own property in *parameters*:
    1. Let *key* be the name of the property.
    2. Let *value* be the result of getting the property *name* from *parameters*.
    3. Find the *timeout type* from the table of session timeouts by looking it up by its keyword *key*.
    4. Set *timeout type*'s *value*.
3. Return success with data null.

9. Navigation

The commands in this section allow navigation of the current top-level browsing context to new URLs and introspection of the document currently loaded in this browsing context.

For commands that cause a new document to load, the point at which the command returns is determined by the session's page loading strategy. The normal state causes it to return after the `load` event fires on the new page, eager causes it to return after the `DOMContentLoaded` event fires, and none causes it to return immediately.

Navigation actions are also affected by the value of the session page load timeout, which determines the maximum time that commands will block before returning with a timeout error.

The following is the ***table of page load strategies*** that links the `pageLoadStrategy` capability keyword to a page loading strategy state, and shows which document readiness state that corresponds to it:

Keyword	Page load strategy state	Document readiness state
`"none"`	***none***	
`"eager"`	***eager***	`"interactive"`
`"normal"`	***normal***	`"complete"`

When asked to ***deserialize as a page load strategy*** with argument *value*:

1. If *value* is not a string return an error with error code invalid argument.
2. If there is no entry in the table of page load strategies with `keyword` *value* return an error with error code invalid argument.
3. Return success with data *value*.

When asked to ***wait for navigation to complete***, run the following steps:

1. If the current session has a page loading strategy of none, return success with data null.
2. If the current browsing context is no longer open, return success with data `null`.
3. Start a *timer*. If this algorithm has not completed before *timer* reaches the session's session page load timeout in milliseconds, return an error with error code timeout.
4. If there is an ongoing attempt to navigate the current browsing context that has not yet matured, wait for navigation to mature.
5. Let *readiness target* be the document readiness state associated with the current session's page loading strategy, which can be found in the table of page load strategies.

6. Wait for the the current browsing context's document readiness state to reach *readiness target*, or for the session page load timeout to pass, whichever occurs sooner.

7. If the previous step completed by the session page load timeout being reached and the browser does not have an active user prompt, return error with error code timeout.

8. Return success with data null.

When asked to run the ***post-navigation checks***, run the substeps of the first matching statement:

↪ **response is a network error**

Return error with error code unknown error.

> NOTE
>
> A "network error" in this case is not an HTTP response with a status code indicating an unsucessful result, but could be a problem occurring lower in the OSI model, or a failed DNS lookup.

↪ **response is blocked by content security policy**

If the current session's secure TLS state is disabled, take implementation specific steps to ensure the navigation is not aborted and that the untrusted- or invalid TLS certificate error that would normally occur under these circumstances, are suppressed.

Otherwise return error with error code insecure certificate.

↪ **response's HTTP status is 401**

↪ **Otherwise**

Irrespective of how a possible authentication challenge is handled, return success with data null.

9.1 *Navigate To*

HTTP Method	URI Template
POST	/session/{*session id*}/url

> NOTE
>
> The command causes the user agent to navigate the current top-level browsing context to a new location.

If the session is not in a secure TLS state, no certificate errors that would normally cause the user agent to abort and show a security warning are to hinder navigation to the requested address.

> EXAMPLE 8
>
> To navigate the current top-level browsing context of the session with ID *1* to `https://example.com`, the local end would POST to */session/1/url* with the body:
>
> ```
> {"url": "https://example.com"}
> ```

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Let *url* be the result of getting the property `url` from the *parameters* argument.
3. If *url* is not an absolute URL or is not an absolute URL with fragment or not a local scheme, return error with error code invalid argument.
4. Handle any user prompts and return its value if it is an error.
5. Let *current URL* be the current top-level browsing context's active document's document URL.
6. If *current URL* and *url* do not have the same absolute URL:
    1. If *timer* has not been started, start a *timer*. If this algorithm has not completed before *timer* reaches the session's session page load timeout in milliseconds, return an error with error code timeout.
7. Navigate the current top-level browsing context to *url*.
8. If *url* is special except for `file` and *current URL* and *URL* do not have the same absolute URL :
    1. Try to wait for navigation to complete.
    2. Try to run the post-navigation checks.
9. Set the current browsing context to the current top-level browsing context.
10. If the current top-level browsing context contains a refresh state pragma directive of *time* 1 second or less, wait until the refresh timeout has elapsed, a new navigate has begun, and return to the first step of this algorithm.

11. Return success with data null.

9.2 *Get Current URL*

HTTP Method	URI Template
GET	/session/{*session id*}/url

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *url* be the serialization of the current top-level browsing context's active document's document URL.
4. Return success with data *url*.

9.3 *Back*

HTTP Method	URI Template
POST	/session/{*session id*}/back

> NOTE
>
> This command causes the browser to traverse one step backward in the joint session history of the current top-level browsing context. This is equivalent to pressing the back button in the browser chrome or invoking `window.history.back`.

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Traverse the history by a delta –1 for the current browsing context.
4. If the previous step completed results in a `pageHide` event firing, wait until `pageShow` event fires or for the session page load timeout milliseconds to pass, whichever occurs sooner.

5. If the previous step completed by the session page load timeout being reached, and user prompts have been handled, return error with error code timeout.

6. Return success with data null.

9.4 *Forward*

HTTP Method	URI Template
POST	/session/{*session id*}/forward

> NOTE
>
> This command causes the browser to traverse one step forwards in the joint session history of the current top-level browsing context.

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.

2. Handle any user prompts and return its value if it is an error.

3. Traverse the history by a delta 1 for the current browsing context.

4. If the previous step completed results in a `pageHide` event firing, wait until `pageShow` event fires or for the session page load timeout milliseconds to pass, whichever occurs sooner.

5. If the previous step completed by the session page load timeout being reached, and user prompts have been handled, return error with error code timeout.

6. Return success with data null.

9.5 *Refresh*

HTTP Method	URI Template
POST	/session/{*session id*}/refresh

> NOTE
>
> This command causes the browser to reload the page in the current top-level browsing context.

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Initiate an overridden reload of the current top-level browsing context's active document.
4. If *url* is special except for `file`:
    1. Try to wait for navigation to complete.
    2. Try to run the post-navigation checks.
5. Set the current browsing context to the current top-level browsing context.
6. Return success with data null.

9.6 *Get Title*

HTTP Method	URI Template
GET	/session/{*session id*}/title

> NOTE
>
> This command returns the document title of the current top-level browsing context, equivalent to calling `document.title`.

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.

3. Let *title* be the result of calling the algorithm for getting the `title` attribute of the current top-level browsing context's active document.

4. Return success with data *title*.

10. Command Contexts

Many WebDriver commands happen in the context of either the current browsing context or current top-level browsing context. The current top-level browsing context is represented in the protocol by its associated window handle. When a top-level browsing context is selected using the Switch To Window command, a specific browsing context can be selected using the Switch to Frame command.

> NOTE
>
> The use of the term “window” to refer to a top-level browsing context is legacy and doesn’t correspond with either the operating system notion of a “window” or the DOM `Window` object.

A browsing context is said to be ***no longer open*** if it has been discarded.

Each browsing context has an associated ***window handle*** which uniquely identifies it. This must be a String and must not be "`current`".

The ***web window identifier*** is the string constant "`window-fcc6-11e5-b4f8-330a88ab9d7f`".

The ***web frame identifier*** is the string constant "`frame-075b-4da1-b6ba-e579c2d3230a`".

The ***JSON serialization of the `WindowProxy` object*** is the JSON Object obtained by applying the following algorithm to the given `WindowProxy` object *window*:

1. Let *identifier* be the web window identifier if the associated browsing context of *window* is a top-level browsing context.

   Otherwise let it be the web frame identifier.

2. Return a JSON Object initialised with the following properties:

   identifier
   : Associated window handle of the *window*’s browsing context.

> NOTE
>
> In accordance with the focus section of the [HTML] specification, commands are unaffected by whether the operating system window has focus or not.

10.1 *Get Window Handle*

HTTP Method	URI Template
GET	/session/{*session id*}/window

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Return success with data being the window handle associated with the current top-level browsing context.

10.2 *Close Window*

HTTP Method	URI Template
DELETE	/session/{*session id*}/window

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Close the current top-level browsing context.
4. If there are no more open top-level browsing contexts, then try to close the session.
5. Return the result of running the remote end steps for the Get Window Handles command.

10.3 *Switch To Window*

HTTP Method	URI Template

POST	/session/{*session id*}/window

> NOTE
>
> Switching window will select the current top-level browsing context used as the target for all subsequent commands. In a tabbed browser, this will typically make the tab containing the browsing context the selected tab.

The remote end steps are:

1. Let *handle* be the result of getting the property "`handle`" from the *parameters* argument.
2. If *handle* is undefined, return error with error code invalid argument.
3. If there is an active user prompt, that prevents the focussing of another top-level browsing context, return error with error code unexpected alert open.
4. If *handle* is equal to the associated window handle for some top-level browsing context in the current session, set the session's current browsing context to that browsing context.

   Otherwise, return error with error code no such window.
5. Update any implementation-specific state that would result from the user selecting the current browsing context for interaction, without altering OS-level focus.
6. Return success with date null.

10.4 *Get Window Handles*

HTTP Method	URI Template
GET	/session/{*session id*}/window/handles

The order in which the window handles are returned is arbitrary.

The remote end steps are:

1. Let *handles* be a JSON List.
2. For each top-level browsing context in the remote end, push the associated window handle onto *handles*.
3. Return success with data *handles*.

EXAMPLE 9

In order to determine whether or not a particular interaction with the browser opens a new window, one can obtain the set of window handles before the interaction is performed and compare it with the set after the action is performed.

10.5 *Switch To Frame*

HTTP Method	URI Template
POST	/session/{*session id*}/frame

NOTE

The Switch To Frame command is used to select the current top-level browsing context or a child browsing context of the current browsing context to use as the current browsing context for subsequent commands.

The remote end steps are:

1. Let *id* be the result of getting the property "`id`" from the *parameters* argument.
2. If *id* is not null, a `Number` object, or an Object that represents a web element, return error with error code no such frame.
3. If the current browsing context is no longer open, return error with error code no such window.
4. Handle any user prompts and return its value if it is an error.
5. Run the substeps of the first matching condition:

   ↪ ***id* is null**

   1. Set the current browsing context to the current top-level browsing context.

   ↪ ***id* is a `Number` object**

   1. If *id* is less than 0 or greater than $2^{16} - 1$, return error with error code no such frame.
   2. Let *window* be the associated window of the current browsing context's active document.

3. If *id* is not a supported property index of *window*, return error with error code no such frame.

4. Let *child window* be the `WindowProxy` object obtained by calling *window*.`[[GetOwnProperty]]` (*id*).

5. Set the current browsing context to *child window*'s browsing context.

↪ ***id* represents a web element**

1. Let *element* be the result of trying to get a known element by web element reference *id*.

2. If *element* is stale, return error with error code stale element reference.

3. If *element* is not a `frame` or `iframe` element, return error with error code no such frame.

4. Set the current browsing context to *element*'s nested browsing context.

6. Update any implementation-specific state that would result from the user selecting the current browsing context for interaction, without altering OS-level focus.

7. Return success with data null.

> NOTE
>
> WebDriver is not bound by the same origin policy, so it is always possible to switch into child browsing contexts, even if they are different origin to the current browsing context.

10.6 *Switch To Parent Frame*

HTTP Method	URI Template
POST	/session/{*session id*}/frame/parent

> NOTE
>
> The Switch to Parent Frame command sets the current browsing context for future commands to the parent of the current browsing context.

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. If the current browsing context is not equal to the current top-level browsing context, set the current browsing context to the parent browsing context of the current browsing context.
4. Update any implementation-specific state that would result from the user selecting the current browsing context for interaction, without altering OS-level focus.
5. Return success with data null.

10.7 Resizing and Positioning Windows

WebDriver provides commands for interacting with the operating system window containing the current top-level browsing context. Because different operating systems' window managers provide different abilities, not all of the commands in this section can be supported by all remote ends. Support for these commands is determined by the window dimensioning/positioning capability. Where a command is not supported, an unsupported operation error is returned.

The top-level browsing context has an associated ***window state*** which describes what visibility state its OS widget window is in. It can be in one of the following states:

State	Keyword	Default	Description
Maximized window state	`"maximized"`		The window is maximized.
Minimized window state	`"minimized"`		The window is iconified.
Normal window state	`"normal"`	✓	The window is shown normally.
Fullscreen window state	`"fullscreen"`		The window is in full screen mode.

If for whatever reason the top-level browsing context's OS window cannot enter either of the window states, or if this concept is not applicable on the current system, the default state must be normal.

A top-level browsing context's ***window rect*** is defined as a dictionary of the screenX, screenY, outerWidth and outerHeight attributes of the `WindowProxy`. Its ***JSON representation*** is the following:

`"x"`
: `WindowProxy`'s screenX attribute.

`"y"`
: `WindowProxy`'s screenY attribute.

"`width`"
: `WindowProxy`'s outerWidth attribute.

"`height`"
: `WindowProxy`'s outerHeight attribute.

To ***maximize the window***, given an operating system level window with an associated top-level browsing context, run the implementation-specific steps to transition the operating system level window into the maximized window state. If the window manager supports window resizing but does not have a concept of window maximization, the window dimensions must be increased to the maximum available size permitted by the window manager for the current screen. Return when the window has completed the transition, or within an implementation-defined timeout.

To ***iconify the window***, given an operating system level window with an associated top-level browsing context, run implementation-specific steps to iconify, minimize, or hide the window from the visible screen. Do not return from this operation until the visibility state of the top-level browsing context's active document has reached the hidden state, or until the operation times out.

To ***restore the window***, given an operating system level window with an associated top-level browsing context, run implementation-specific steps to restore or unhide the window to the visible screen. Do not return from this operation until the visibility state of the top-level browsing context's active document has reached the visible state, or until the operation times out.

10.7.1 *Get Window Rect*

HTTP Method	URI Template
GET	/session/{session id}/window/rect

> NOTE
>
> The Get Window Rect command returns the size and position on the screen of the operating system window corresponding to the current top-level browsing context.

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Return success with the JSON serialization of the current top-level browsing context's window rect.

10.7.2 *Set Window Rect*

HTTP Method	URI Template
POST	/session/{*session id*}/window/rect

> NOTE
>
> The Set Window Rect command alters the size and the position of the operating system window corresponding to the current top-level browsing context.

The remote end steps are:

1. Let *width* be the result of getting a property named `width` from the *parameters* argument, else let it be null.
2. Let *height* be the result of getting a property named `height` from the *parameters* argument, else let it be null.
3. Let *x* be the result of getting a property named `x` from the *parameters* argument, else let it be null.
4. Let *y* be the result of getting a property named `y` from the *parameters* argument, else let it be null.
5. If *width* or *height* is neither null nor a Number from 0 to $2^{31} - 1$, return error with error code invalid argument.
6. If *x* or *y* is neither null nor a Number from $-(2^{31})$ to $2^{31} - 1$, return error with error code invalid argument.
7. If the remote end does not support the Set Window Rect command for the current top-level browsing context for any reason, return error with error code unsupported operation.
8. If the current top-level browsing context is no longer open, return error with error code no such window.
9. Handle any user prompts and return its value if it is an error.
10. Fully exit fullscreen.
11. Restore the window.
12. If *width* and *height* are not null:

1. Set the width, in CSS pixels, of the operating system window containing the current top-level browsing context, including any browser chrome and externally drawn window decorations to a value that is as close as possible to *width*.

2. Set the height, in CSS pixels, of the operating system window containing the current top-level browsing context, including any browser chrome and externally drawn window decorations to a value that is as close as possible to *height*.

> NOTE
>
> The specification does not guarantee that the resulting window size will exactly match that which was requested. In particular the implementation is expected to clamp values that are larger than the physical screen dimensions, or smaller than the minimum window size.
>
> Particular implementations may have other limitations such as not being able to resize in single-pixel increments.
>
> This is intended to mutate the value of the current top-level browsing context's `WindowProxy`'s outerWidth and outerHeight properties. Specifically, the value of outerWidth should be as close as possible to *width* and the value of outerHeight should be as close as possible to *height*.

13. If *x* and *y* are not null:

    1. Run the implementation-specific steps to set the position of the operating system level window containing the current top-level browsing context to the position given by the *x* and *y* coordinates.

    > NOTE
    >
    > Note that this step is similar to calling the moveTo(x, y) method on the `WindowProxy` object associated with the current top-level browsing context, but without the security restrictions that you
    >
    > a. cannot move a window or tab that was not created by `window.open`.
    >
    > b. cannot move a window or tab when it is in a window with more than one tab.

14. Return success with the JSON serialization of the current top-level browsing context's window rect.

10.7.3 *Maximize Window*

HTTP Method	URI Template
POST	/session/{*session id*}/window/maximize

> NOTE
>
> The Maximize Window command invokes the window manager-specific “maximize” operation, if any, on the window containing the current top-level browsing context. This typically increases the window to the maximum available size without going full-screen.

The remote end steps are:

1. If the remote end does not support the Maximize Window command for the current top-level browsing context for any reason, return error with error code unsupported operation.
2. If the current top-level browsing context is no longer open, return error with error code no such window.
3. Handle any user prompts and return its value if it is an error.
4. Fully exit fullscreen.
5. Restore the window.
6. Maximize the window of the current top-level browsing context.
7. Return success with the JSON serialization of the current top-level browsing context’s window rect.

10.7.4 *Minimize Window*

HTTP Method	URI Template
POST	/session/{*session id*}/window/minimize

> NOTE
>
> The Minimize Window command invokes the window manager-specific “minimize” operation, if any, on the window containing the current top-level browsing context. This typically hides the window in the system tray.

The remote end steps are:

1. If the remote end does not support the Minimize Window command for the current top-level browsing context for any reason, return error with error code unsupported operation.
2. If the current top-level browsing context is no longer open, return error with error code no such window.
3. Handle any user prompts and return its value if it is an error.
4. Fully exit fullscreen.
5. Iconify the window.
6. Return success with the JSON serialization of the current top-level browsing context's window rect.

10.7.5 *Fullscreen Window*

HTTP Method	URI Template
POST	/session/{*session id*}/window/fullscreen

The remote end steps are:

1. If the remote end does not support fullscreen return error with error code unsupported operation.
2. If the current top-level browsing context is no longer open, return error with error code no such window.
3. Handle any user prompts and return its value if it is an error.
4. Restore the window.
5. Call fullscreen an element with the current top-level browsing context's active document's document element.
6. Return success with the JSON serialization of the current top-level browsing context's window rect.

11. Elements

A ***web element*** is an abstraction used to identify an element when it is transported via the protocol, between remote- and local ends.

The ***web element identifier*** is the string constant "`element-6066-11e4-a52e-4f735466cecf`".

Each element has an associated ***web element reference*** that uniquely identifies the element across all browsing contexts. The web element reference for every element representing the same element must be the same. It must be a string, and should be the result of generating a UUID.

An ECMAScript Object ***represents a web element*** if it has a web element identifier own property.

Each browsing context has an associated ***list of known elements***. When the browsing context is discarded, the list of known elements is discarded along with it.

To ***get a known element*** with argument *reference*, run the following steps:

1. Let *element* be the item in the current browsing context's list of known elements for which the web element reference matches *reference*, if such an element exists. Otherwise return error with error code no such element.
2. If *element* is stale, return error with error code stale element reference.
3. Return success with *element*.

To ***get a known connected element*** with argument *reference*, run the following steps:

1. Let *element* be the result of trying to get a known element with argument *reference*.
2. If *element* is not connected return error with error code stale element reference.
3. Return success with *element*.

To ***create a web element reference*** for an *element*:

1. For each *known element* of the current browsing context's list of known elements:
    1. If *known element* equals *element*, return success with *known element*'s web element reference.
2. Add *element* to the list of known elements of the current browsing context.
3. Return success with the *element*'s web element reference.

The ***JSON serialization of an element*** is a JSON Object where the web element identifier key is mapped to the element's web element reference.

When required to ***deserialize a web element*** by a JSON Object *object* that represents a web element:

1. If *object* has no own property web element identifier, return error with error code invalid argument.
2. Let *reference* be the result of getting the web element identifier property from *object*.

3. Let *element* be the result of trying to get a known element with argument *reference*.

4. Return success with data *element*.

An element ***is stale*** if its node document is not the active document or if its context object is not connected.

To ***scroll into view*** an *element* perform the following steps only if the element is not already in view:

1. Let *options* be the following `ScrollIntoViewOptions`:

   "behavior"
   "instant"

   Logical scroll position "block"
   "end"

   Logical scroll position "inline"
   "nearest"

2. On the *element*, Call(scrollIntoView, *options*).

Editable elements are those that can be used for typing and clearing, and they fall into two subcategories:

Mutable form control elements
Denotes `input` elements that are mutable (e.g. that are not read only or disabled) and whose `type` attribute is in one of the following states:

⇒ Text and Search, URL, Telephone, Email, Password, Date, Month, Week, Time, Local Date and Time, Number, Range, Color, File Upload

And the `textarea` element.

Mutable elements
Denotes elements that are editing hosts or content editable.

An element is said to have ***pointer events disabled*** if the resolved value of its "`pointer-events`" style property is "`none`".

An element is to be considered ***read only*** if it is an `input` element whose `readOnly` attribute is set.

11.1 Element Interactability

In order to determine if an element can be interacted with using pointer actions, WebDriver performs hit-testing to find if the interaction will be able to reach the requested element.

An ***interactable element*** is an element which is either pointer-interactable or keyboard-interactable.

A ***pointer-interactable element*** is defined to be the first element, defined by the paint order found at the center point of its rectangle that is inside the viewport, excluding the size of any rendered scrollbars.

A ***keyboard-interactable element*** is any element that has a focusable area, is a `body` element, or is the document element.

An element's ***in-view center point*** is the origin position of the rectangle that is the intersection between the element's first DOM client rectangle and the initial viewport. Given an element that is known to be in view, it can be calculated this way:

1. Let *rectangle* be the first element of the `DOMRect` sequence returned by calling getClientRects on *element*.
2. Let *left* be (max(0, min(x coordinate, x coordinate + width dimension))).
3. Let *right* be (min(innerWidth, max(x coordinate, x coordinate + width dimension))).
4. Let *top* be (max(0, min(y coordinate, y coordinate + width dimension))).
5. Let *bottom* be (min(innerHeight, max(y coordinate, y coordinate + height dimension))).
6. Let *x* be (0.5 × (*left* + *right*)).
7. Let *y* be (0.5 × (*top* + *bottom*)).
8. Return *x* and *y* as a pair.

An element is ***in view*** if it is a member of its own pointer-interactable paint tree, given the pretense that its pointer events are not disabled.

An element is ***obscured*** if the pointer-interactable paint tree at its center point is empty, or the first element in this tree is not an inclusive descendant of itself.

EXAMPLE 10

This ascertains if the element's in-view center point would be possible to interact with.

For example, the paint tree at this button's center point, the red square, is not itself the button or a descendant of the button. In other words, it is not an *inclusive descendant*. This makes the button *obscured*:

On the other hand, the center point of the following select list is the third `option` element, because unlike a drop-down list, `<select multiple>`'s options are individually visible and painted. Because the option is a *descendant* of the `select` element, it is *not* obscured:

An *element*'s ***pointer-interactable paint tree*** is produced this way:

1. If *element* is not in the same tree as the current browsing context's active document, return an empty sequence.
2. Let *rectangles* be the `DOMRect` sequence returned by calling getClientRects.
3. If *rectangles* has the length of 0, return an empty sequence.
4. Let *center point* be the in-view center point of the first indexed element in *rectangles*.
5. Return the elements from point given the coordinates *center point*.

12. Element Retrieval

The Find Element, Find Elements, Find Element From Element, and Find Elements From Element commands allow lookup of individual elements and collections of elements. Element retrieval searches are performed using pre-order traversal of the document's nodes that match the provided selector's expression. Elements are serialized and returned as web elements.

When required to ***find*** with arguments *start node*, *using* and *value*, a remote end must run the following steps:

1. Let *end time* be the current time plus the session implicit wait timeout.
2. Let *location strategy* be equal to *using*.
3. Let *selector* be equal to *value*.
4. Let *elements returned* be the result of trying to call the relevant element location strategy with arguments *start node*, and *selector*.
5. If a `DOMException`, `SyntaxError`, `XPathException`, or other error occurs during the execution of the element location strategy, return error invalid selector.
6. If *elements returned* is empty and the current time is less than *end time* return to step 4. Otherwise, continue to the next step.
7. Let *result* be an empty JSON List.
8. For each *element* in *elements returned*, append the serialization of *element* to *result*.
9. Return success with data *result*.

12.1 Locator Strategies

An ***element location strategy*** is an enumerated attribute deciding what technique should be used to search for elements in the current browsing context. The following ***table of location strategies*** lists the keywords and states defined for this attribute:

State	Keyword
CSS selector	`"css selector"`
Link text selector	`"link text"`
Partial link text selector	`"partial link text"`
Tag name	`"tag name"`
XPath selector	`"xpath"`

12.1.1 CSS Selectors

To find a web element with the ***CSS Selector*** strategy the following steps need to be completed:

1. Let *elements* be the result of calling querySelectorAll with *selector* with the context object equal to the *start node*. If this causes an exception to be thrown, return error with error code invalid selector.

2. Return success with data *elements*.

12.1.2 Link Text

To find a web element with the ***Link Text*** strategy the following steps need to be completed:

1. Let *elements* be the result of calling querySelectorAll, with argument a elements, with the context object equal to the *start node*. If this throws an exception, return error with error code unknown error.
2. Let *result* be an empty NodeList.
3. For each *element* in *elements*:
    1. Let *rendered text* be the value that would be returned via a call to Get Element Text for *element*.
    2. Let *trimmed text* be the result of removing all whitespace from the start and end of the string *rendered text*.
    3. If *trimmed text* equals *selector*, append *element* to *result*.
4. Return success with data *result*.

12.1.3 Partial Link Text

The ***Partial link text*** strategy is very similar to the Link Text strategy, but rather than matching the entire string, only a substring needs to match. That is, return all a elements with rendered text that contains the selector expression.

To find a web element with the Partial Link Text strategy the following steps need to be completed:

1. Let *elements* be the result of calling querySelectorAll, with argument a elements, with the context object equal to the *start node*. If this throws an exception, return error with error code unknown error.
2. Let *result* be an empty NodeList.
3. For each *element* in *elements*:
    1. Let *rendered text* be the value that would be returned via a call to Get Element Text for *element*.
    2. If *rendered text* contains *selector*, append *element* to *result*.

4. Return success with data *result*.

12.1.4 Tag Name

To find a web element with the ***Tag Name*** strategy return success with data set to the result of calling `getElementsByTagName` with the argument *selector*, with the context object equal to the *start node*.

12.1.5 XPath

To find a web element with the ***XPath Selector*** strategy the following steps need to be completed:

1. Let *evaluateResult* be the result of calling `evaluate`, with arguments *selector*, *start node*, null, `ORDERED_NODE_SNAPSHOT_TYPE`, and null.

   > NOTE
   >
   > A snapshot is used to promote operation atomicity.

2. Let *index* be 0.
3. Let *length* be the result of getting the property "`snapshotLength`" from *evaluateResult*. If this throws an `XPathException` return error with error code invalid selector, otherwise if this throws any other exception return error with error code unknown error.
4. Let *result* be an empty NodeList.
5. Repeat, while *index* is less than *length*:
   1. Let *node* be the result of calling `snapshotItem` with argument *index*, with the context object equal to *evaluateResult*.
   2. If *node* is not an element return an error with error code invalid selector.
   3. Append *node* to *result*.
   4. Increment *index* by 1.
6. Return success with data *result*.

12.2 *Find Element*

HTTP Method	URI Template

POST	/session/{*session id*}/element

> NOTE
>
> The Find Element command is used to find an element in the current browsing context that can be used as the web element context for future element-centric commands.
>
> For example, consider this pseudo code which retrieves an element with the `#toremove` ID and uses this as the argument for a script it injects to remove it from the HTML document:
>
> ```
> let body = session.find.css("#toremove");
> session.execute("arguments[0].remove()", [body]);
> ```

The remote end steps are:

1. Let *location strategy* be the result of getting a property called "`using`".
2. If *location strategy* is not present as a keyword in the table of location strategies, return error with error code invalid argument.
3. Let *selector* be the result of getting a property called "`value`".
4. If *selector* is undefined, return error with error code invalid argument.
5. If the current browsing context is no longer open, return error with error code no such window.
6. Let *start node* be the current browsing context's document element.
7. If *start node* is null, return error with error code no such element.
8. Let *result* be the result of trying to Find with *start node*, *location strategy*, and *selector*.
9. If *result* is empty, return error with error code no such element. Otherwise, return the first element of *result*.

12.3 *Find Elements*

HTTP Method	URI Template
POST	/session/{*session id*}/elements

The remote end steps are:

1. Let *location strategy* be the result of getting a property called "`using`".
2. If *location strategy* is not present as a keyword in the table of location strategies, return error with error code invalid argument.
3. Let *selector* be the result of getting a property called "`value`".
4. If *selector* is undefined, return error with error code invalid argument.
5. If the current browsing context is no longer open, return error with error code no such window.
6. Let *start node* be the current browsing context's document element.
7. If *start node* is null, return error with error code no such element.
8. Return the result of trying to Find with *start node*, *location strategy*, and *selector*.

12.4 *Find Element From Element*

HTTP Method	URI Template
POST	/session/{*session id*}/element/{*element id*}/element

The remote end steps are:

1. Let *location strategy* be the result of getting a property called "`using`".
2. If *location strategy* is not present as a keyword in the table of location strategies, return error with error code invalid argument.
3. Let *selector* be the result of getting a property called "`value`".
4. If *selector* is undefined, return error with error code invalid argument.
5. If the current browsing context is no longer open, return error with error code no such window.
6. Let *start node* be the result of trying to get a known connected element with url variable *element id*.
7. Let *result* be the value of trying to Find with *start node*, *location strategy*, and *selector*.
8. If *result* is empty, return error with error code no such element. Otherwise, return the first element of *result*.

12.5 *Find Elements From Element*

HTTP Method	URI Template
POST	/session/{*session id*}/element/{*element id*}/elements

The remote end steps are:

1. Let *location strategy* be the result of getting a property called "`using`".
2. If *location strategy* is not present as a keyword in the table of location strategies, return error with error code invalid argument.
3. Let *selector* be the result of getting a property called "`value`".
4. If *selector* is undefined, return error with error code invalid argument.
5. If the current browsing context is no longer open, return error with error code no such window.
6. Let *start node* be the result of trying to get a known connected element with url variable *element id*.
7. Return the result of trying to Find with *start node*, *location strategy*, and *selector*.

12.6 *Get Active Element*

HTTP Method	URI Template
GET	/session/{*session id*}/element/active

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *active element* be the active element of the current browsing context's document element.
4. If *active element* is a non-null element, return success with its JSON serialization.

   Otherwise, return error with error code no such element.

13. Element State

To ***calculate the absolute position*** of *element*:

1. Let *rect* be the value returned by calling getBoundingClientRect.
2. Let *window* be the associated window of current top-level browsing context.
3. Let *x* be (scrollX of *window* + *rect*'s x coordinate).
4. Let *y* be (scrollY of *window* + *rect*'s y coordinate).
5. Return a pair of (*x*, *y*).

To determine if node is ***not in the same tree*** as another node, *other*, run the following substeps:

1. If the node's node document is not *other*'s node document, return true.
2. Return true if the result of calling the node's compareDocumentPosition with *other* as argument is DOCUMENT_POSITION_DISCONNECTED (1), otherwise return false.

An *element*'s ***container*** is:

↪ **`option` element in a valid element context**
↪ **`optgroup` element in a valid element context**

The *element*'s element context, which is determined by:

1. Let *datalist parent* be the first `datalist` element reached by traversing the tree in reverse order from *element*, or undefined if the root of the tree is reached.
2. Let *select parent* be the first `select` element reached by traversing the tree in reverse order from *element*, or undefined if the root of the tree is reached.
3. If *datalist parent* is undefined, the element context is *select parent*. Otherwise, the element context is *datalist parent*.

↪ **`option` element in an invalid element context**

The element does not have a container.

↪ **Otherwise**

The container is the element itself.

13.1 *Is Element Selected*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/selected

> NOTE
>
> The Is Element Selected command determines if the referenced element is selected or not. This operation only makes sense on `input` elements of the Checkbox- and Radio Button states, or on `option` elements.

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known connected element with url variable *element id*.
4. Let *selected* be the value corresponding to the first matching statement:

   ↪ ***element* is an `input` element with a `type` attribute in the Checkbox- or Radio Button state**
   The result of *element*'s checkedness.

   ↪ ***element* is an `option` element**
   The result of *element*'s selectedness.

   ↪ **Otherwise**
   False.

5. Return success with data *selected*.

13.2 *Get Element Attribute*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/attribute/{*name*}

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known element with url variable *element id*.

4. Let *result* be the result of the first matching condition:

   ↪ **If *name* is a boolean attribute**
   "`true`" (string) if the *element* has the attribute, otherwise null.

   ↪ **Otherwise**
   The result of getting an attribute by name *name*.

5. Return success with data *result*.

> NOTE
>
> Please note that the behavior of this command deviates from the behavior of getAttribute in [DOM], which in the case of a set boolean attribute would return an empty string. The reason this command returns true as a string is because this evaluates to true in most dynamically typed programming languages, but still preserves the expected type information.

13.3 *Get Element Property*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/property/{*name*}

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known element with url variable *element id*.
4. Let *property* be the result of calling the *element*.[[GetProperty]](*name*).
5. Let *result* be the value of *property* if not undefined, or null.
6. Return success with data *result*.

13.4 *Get Element CSS Value*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/css/{*property name*}

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known connected element with url variable *element id*.
4. Let *computed value* be the result of the first matching condition:

   ↪ **current browsing context's document type is not "xml"**
   computed value of parameter *property name* from *element*'s style declarations. *property name* is obtained from url variables.

   ↪ **Otherwise**
   "" (empty string)

5. Return success with data *computed value*.

13.5 *Get Element Text*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/text

> NOTE
>
> The Get Element Text command intends to return an element's text "as rendered". An element's rendered text is also used for locating `a` elements by their link text and partial link text.
>
> One of the major inputs to this specification was the open source Selenium project. This was in wide-spread use before this specification written, and so had set user expectations of how the Get Element Text command should work. As such, the approach presented here is known to be flawed, but provides the best compatibility with existing users.

When processing text, ***whitespace*** is defined as characters from the Unicode Character Database ([UAX44]) with the Unicode character property `"WSpace=Y"`.

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known connected element with url variable *element id*.
4. Let *rendered text* be the result of performing implementation-specific steps whose result is exactly the same as the result of a Call(`bot.dom.getVisibleText`, null, *element*).
5. Return success with data *rendered text*.

13.6 *Get Element Tag Name*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/name

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known element with url variable *element id*.
4. Let *qualified name* be the result of getting *element*'s tagName content attribute.
5. Return success with data *qualified name*.

13.7 *Get Element Rect*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/rect

NOTE

The Get Element Rect command returns the dimensions and coordinates of the given web element. The returned value is a dictionary with the following members:

x
: X axis position of the top-left corner of the web element relative to the current browsing context's document element in CSS pixels.

y
: Y axis position of the top-left corner of the web element relative to the current browsing context's document element in CSS pixels.

height
: Height of the web element's bounding rectangle in CSS pixels.

width
: Width of the web element's bounding rectangle in CSS pixels.

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known connected element with url variable *element id*.
4. Calculate the absolute position of *element* and let it be *coordinates*.
5. Let *rect* be *element*'s bounding rectangle.
6. Let *body* be a new JSON Object initialised with:

   "x"
   : The first value of *coordinates*.

   "y"
   : The second value of *coordinates*.

   "width"
   : Value of *rect*'s width dimension.

   "height"
   : Value of *rect*'s height dimension.

7. Return success with data *body*.

13.8 *Is Element Enabled*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/enabled

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts and return its value if it is an error.
3. Let *element* be the result of trying to get a known connected element with url variable *element id*.
4. Let *enabled* be a boolean initially set to true if the current browsing context's document type is not "`xml`".

   Otherwise, let *enabled* to false and jump to the last step of this algorithm.
5. Set *enabled* to false if a form control is disabled.
6. Return success with data *enabled*.

14. Element Interaction

The element interaction commands provide a high-level instruction set for manipulating form controls. Unlike Actions, they will implicitly scroll elements into view and check that it is an interactable element.

Some resettable elements define their own ***clear algorithm***. Unlike their associated reset algorithms, changes made to form controls as part of these algorithms *do* count as changes caused by the user (and thus, e.g. do cause `input` events to fire). When the clear algorithm is invoked for an element that does not define its own clear algorithm, its reset algorithm must be invoked instead.

The clear algorithm for `input` elements is to set the dirty value flag and dirty checkedness flag back to false, set the value of the element to an empty string, set the checkedness of the element to true if the element has a `checked` content attribute and false if it does not, empty the list of selected files, and then invoke the value sanitization algorithm, if the `type` attribute's current state defines one.

The clear algorithm for `textarea` elements is to set the dirty value flag back to false, and set the raw value of element to an empty string.

The clear algorithm for `output` elements is set the element's value mode flag to default and then to set the element's `textContent` IDL attribute to an empty string (thus clearing the element's child nodes).

14.1 *Element Click*

HTTP Method	URI Template
POST	/session/{*session id*}/element/{*element id*}/click

> NOTE
>
> The Element Click command scrolls into view the element if it is not already pointer-interactable, and clicks its in-view center point.
>
> If the element's center point is obscured by another element, an element click intercepted error is returned. If the element is outside the viewport, an element not interactable error is returned.

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Let *element* be the result of trying to get a known connected element with argument *element id*.
3. If the *element* is an `input` element in the file upload state return error with error code invalid argument.
4. Scroll into view the *element*'s container.
5. If *element*'s container is still not in view, return error with error code element not interactable.
6. If *element*'s container is obscured by another element, return error with error code element click intercepted.
7. Matching on *element*:

   ↪ **`option` element**

   1. Let *parent node* be the *element*'s container.
   2. Fire a mouseOver event at *parent node*.

3. Fire a mouseMove event at *parent node*.
4. Fire a mouseDown event at *parent node*.
5. Run the focusing steps on *parent node*.
6. If *element* is not disabled:
    1. Fire an `input` event at *parent node*.
    2. Let *previous selectedness* be equal to *element* selectedness.
    3. If *element*'s container has the `multiple` attribute, toggle the *element*'s selectedness state by setting it to the opposite value of its current selectedness.

       Otherwise, set the *element*'s selectedness state to true.
    4. If *previous selectedness* is false, fire a `change` event at *parent node*.
7. Fire a mouseUp event at *parent node*.
8. Fire a click event at *parent node*.

↪ **Otherwise**

1. Let *mouse* be a new pointer input source.
2. Let *click point* be the *element*'s in-view center point.
3. Let *pointer move action* be an action object with the *mouse*'s `id`, "`pointer`", and "`pointerMove`" as arguments.
4. Set a property `x` to `0` on *pointer move action*.
5. Set a property `y` to `0` on *pointer move action*.
6. Set a property `origin` to *element* on *pointer move action*.
7. Let *pointer down action* be an action object with the *mouse*'s `id`, "`pointer`", and "`pointerDown`" as arguments.
8. Set a property `button` to `0` on *pointer down action*.
9. Let *pointer up action* be an action object with the *mouse*'s `id`, "`mouse`", and "`pointerUp`" as arguments.
10. Set a property `button` to `0` on *pointer up action*.

11. Dispatch a pointerMove action with arguments *mouse*'s `input id`, *pointer move action*, *mouse*'s input source state, and `0`.
12. Dispatch a pointerDown action with arguments *mouse*'s `input id`, *pointer down action*, *mouse*'s input source state, and `0`.
13. Dispatch a pointerUp action with arguments *mouse*'s `input id`, *pointer up action*, *mouse*'s input source state, and `0`.
14. Remove *mouse* from the current session's input state table.
15. Remove *mouse* from the list of active input sources.

8. Wait until the user agent event loop has spun enough times to process the DOM events generated by the previous step.
9. Perform implementation-defined steps to allow any navigations triggered by the click to start.

> NOTE
>
> It is not always clear how long this will cause the algorithm to wait, and it is acknowledged that some implementations may have unavoidable race conditions. The intention is to allow a new attempt to navigate to begin so that the next step in the algorithm is meaningful. It is possible the click does not cause an attempt to navigate, in which case the implementation-defined steps can return immediately, and the next step will also return immediately.

10. Try to wait for navigation to complete.
11. Try to run the post-navigation checks.
12. Return success with data null.

14.2 *Element Clear*

HTTP Method	URI Template
POST	/session/{*session id*}/element/{*element id*}/clear

To ***clear a content editable element***:

1. If *element*'s `innerHTML` IDL attribute is an empty string do nothing and return.
2. Run the focusing steps for *element*.

3. Set *element*'s `innerHTML` IDL attribute to an empty string.

4. Run the unfocusing steps for the *element*.

To ***clear a resettable element***:

1. Let *empty* be the result of the first matching condition:

   ↪ ***element* is an `input` element whose `type` attribute is in the File Upload state**
   True if the list of selected files has a length of 0, and false otherwise.

   ↪ **Otherwise**
   True if its value IDL attribute is an empty string, and false otherwise.

2. If *element* is a candidate for constraint validation it satisfies its constraints, and *empty* is true, abort these substeps.

3. Invoke the focusing steps for *element*.

4. Invoke the clear algorithm for *element*.

5. Invoke the unfocusing steps for the *element*.

The remote end steps for Element Clear are:

1. If the current browsing context is no longer open, return error with error code no such window.

2. Let *element* be the result of trying to get a known connected element with argument *element id*.

3. If *element* is not editable, return an error with error code invalid element state.

4. Scroll into view the *element*.

5. Wait in an implementation-specific way up to the session implicit wait timeout for *element* to become interactable.

6. If *element* is not interactable, return error with error code element not interactable.

7. Run the substeps of the first matching statement:

   ↪ ***element* is a mutable form control element**
   Invoke the steps to clear a resettable element.

   ↪ ***element* is a mutable element**
   Invoke the steps to clear a content editable element.

   ↪ **Otherwise**

Return error with error code invalid element state.

8. Return success with data null.

14.3 *Element Send Keys*

HTTP Method	URI Template
POST	/session/{*session id*}/element/{*element id*}/value

> NOTE
>
> The Element Send Keys command scrolls into view the form control element and then sends the provided keys to the element. In case the element is not keyboard-interactable, an element not interactable error is returned.

The key input state used for input may be cleared mid-way through “typing” by sending the ***null key***, which is U+E000 (NULL).

When required to ***clear the modifier key state*** with an argument of *undo actions* and *keyboard* a remote end must run the following steps:

1. If *keyboard* is not a key input source return error with error code invalid argument.
2. For each *entry key* in the lexically sorted keys of *undo actions*:
    1. Let *action* be the value of *undo actions* matching key *entry key*.
    2. If *action* is not an action object of `type` "`key`" and `subtype` "`keyUp`", return error with error code invalid argument.
    3. Dispatch a keyUp action with arguments *action* and *keyboard*’s key input state.

An extended grapheme cluster is ***typeable*** if it consists of a single unicode code point and the code is not undefined.

The ***shifted state*** for *keyboard* is the value of *keyboard*’s key input state’s `shift` property.

In order to ***dispatch the events for a typeable string*** with arguments *text* and *keyboard*, a remote end must for each *char* corresponding to an indexed property in *text*:

1. If *char* is a shifted character and the shifted state of *keyboard* is `false`, create a new action object with *keyboard*'s `id`, `"key"`, and `"keyDown"`, and set its `value` property to `U+E008` ("left shift"). Dispatch a keyDown action with this action object and *keyboard*'s key input state.

2. If *char* is not a shifted character and the shifted state of *keyboard* is `true`, create a new action object with *keyboard*'s `id`, `"key"`, and `"keyUp"`, and set its `value` property to `U+E008` ("left shift"). Dispatch a keyUp action with this action object and *keyboard*'s key input state.

3. Let *keydown action* be a new action object constructed with arguments *keyboard*'s `id`, `"key"`, and `"keyDown"`.

4. Set the `value` property of *keydown action* to *char*

5. Dispatch a keyDown action with arguments *keydown action* and *keyboard*'s key input state.

6. Let *keyup action* be a copy of *keydown action* with the `subtype` property changed to `"keyUp"`.

7. Dispatch a keyUp action with arguments *keyup action* and *keyboard*'s key input state.

When required to ***dispatch a `composition event`*** with arguments *type* and *cluster* the remote end must perform implementation-specific action dispatch steps equivalent to sending composition events in accordance with the requirements of [UI-EVENTS], and producing the following event with the specified properties.

- `composition event` with properties:

Attribute	Value
`type`	*type*
`data`	*cluster*

When required to ***dispatch actions for a string*** with arguments *text* and *keyboard*, a remote end must run the following steps:

1. Let *clusters* be an array created by breaking *text* into extended grapheme clusters.

2. Let *undo actions* be an empty dictionary.

3. Let *current typeable text* be an empty string.

4. For each *cluster* corresponding to an indexed property in *clusters* run the substeps of the first matching statement:

   ↪ ***cluster* is the null key**

   1. Dispatch the events for a typeable string with arguments *current typeable text* and *keyboard*. Reset *current typeable text* to an empty string.

   2. Clear the modifier key state with arguments being *undo actions* and *keyboard*.

   3. If the previous step results in an error, return that error.

   4. Reset *undo actions* to be an empty dictionary.

↪ ***cluster* is a modifier key**

1. Dispatch the events for a typeable string with arguments *current typeable text* and *keyboard*. Reset *current typeable text* to an empty string.
2. Let *keydown action* be an action object constructed with arguments *keyboard*'s id, `"key"`, and `"keyDown"`.
3. Set the `value` property of *keydown action* to *cluster*.
4. Dispatch a keyDown action with arguments *keydown action* and *keyboard*'s key input state.
5. Add an entry to *undo actions* with key *cluster* and value being a copy of *keydown action* with the `subtype` modified to `"keyUp"`.

↪ ***cluster* is typeable**

Append *cluster* to *current typeable text*.

↪ **Otherwise**

1. Dispatch the events for a typeable string with arguments *current typeable text* and *keyboard*. Reset *current typeable text* to an empty string.
2. Dispatch a `composition event` with arguments `"compositionstart"` and undefined.
3. Dispatch a `composition event` with arguments `"compositionupdate"` and *cluster*.
4. Dispatch a `composition event` with arguments `"compositionend"` and *cluster*.

5. Dispatch the events for a typeable string with arguments *current typeable text* and *keyboard*.
6. Clear the modifier key state with arguments *undo actions* and *keyboard*. If an error is returned, return that error

The remote end steps for Element Send Keys are:

1. Let *text* be the result of getting a property called "`text`" from the *parameters* argument.
2. If *text* is not a String, return an error with error code invalid argument.
3. If the current browsing context is no longer open, return error with error code no such window.
4. Handle any user prompts, and return its value if it is an error.

5. Let *element* be the result of trying to get a known connected element with url variable *element id*.

6. Scroll into view the *element*.

7. Wait in an implementation-specific way up to the session implicit wait timeout for *element* to become keyboard-interactable.

8. If *element* is not keyboard-interactable, return error with error code element not interactable.

9. If *element* is not the active element run the focusing steps for the *element*.

10. Run the substeps of the first matching condition:

    ↪ ***element* is an `input` element whose `type` attribute is File.**

    1. Let *files* be the result of splitting *text* on the newline (`\n`) character.

    2. If *files* is of 0 length, return an error with error code invalid argument.

    3. Let *multiple* equal the result of calling `hasAttribute("multiple")` on *element*.

    4. if *multiple* is `false` and the length of *files* is not equal to 1, return an error with error code invalid argument.

    5. Verify that each file given by the user exists. If any do not, return error with error code invalid argument.

    6. Complete implementation specific steps equivalent to setting the selected files on the `input`. If *multiple* is `true` *files* are be appended to *element*'s selected files.

    7. Fire these events in order on *element:*

        1. `input`

        2. `change`

    8. Return success with data null.

    ↪ **The user agent renders *element* as something other than a text input control (for example an `input` element in the `color` state being presented as a colorwheel):**

    1. If *element* does not have an own property named `value` return an error with error code element not interactable

    2. If *element* is not mutable return an error with error code element not interactable.

    3. Set a property `value` to *text* on *element*.

4. If *element* is suffering from bad input return an error with error code invalid argument.

5. Return success with data null.

↪ ***element*** **is content editable**

Set the text insertion caret after any child content.

↪ **Otherwise**

1. Let *current text length* be the JavaScript string's length of *element*'s API value.

2. Set the text insertion caret using set selection range using *current text length* for both the `start` and `end` parameters.

11. Let *keyboard* be a new key input source.

12. Dispatch actions for a string with arguments *text* and *keyboard*.

13. Remove *keyboard* from the current session's input state table

14. Remove *keyboard* from the list of active input sources.

15. Return success with data null.

15. Document Handling

15.1 *Get Page Source*

HTTP Method	URI Template
GET	/session/{*session id*}/source

> NOTE
>
> The Get Page Source command returns a string serialization of the DOM of the current browsing context active document.

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.

2. Handle any user prompts and return its value if it is an error.

3. Let *source* be the result of invoking the fragment serializing algorithm on a fictional node whose only child is the document element providing `true` for the `require well-formed` flag. If this causes an exception to be thrown, let *source* be null.

4. Let *source* be the result of serializing to string the current browsing context active document, if *source* is null.

5. Return success with data *source*.

15.2 Executing Script

When required to ***JSON deserialize*** with argument *value* and optional argument *seen*, a remote end must run the following steps:

1. If *seen* is not provided, let *seen* be an empty List.

2. Jump to the first appropriate step below:

3. Matching on *value*:

   ↪ **undefined**
   ↪ **null**
   ↪ **type Boolean**
   ↪ **type Number**
   ↪ **type String**
   Return success with data *value*.

   ↪ **Object that represents a web element**
   Return the deserialized web element of *value*.

   ↪ **type Array**
   ↪ **type Object**
   Return the result of running the clone an object algorithm with arguments *value* and *seen*, and the JSON deserialize algorithm as the clone algorithm.

To perform a ***JSON clone*** return the result of calling the internal JSON clone algorithm with arguments *value* and an empty List.

When required to run the ***internal JSON clone algorithm*** with arguments *value* and *seen*, a remote end must return the value of the first matching statement, matching on *value*:

↪ **undefined**
↪ **null**
Success with data null.

↪ **type Boolean**
↪ **type Number**
↪ **type String**

Success with data *value*.

↪ **instance of element**
If the *element* is stale, return error with error code stale element reference.

Otherwise, return success with the serialization to JSON of the web element *value*.

↪ **a `WindowProxy` object**
If the associated browsing context of the `WindowProxy` object in *value* has been discarded, return error with error code stale element reference.

Otherwise return success with the JSON serialization of the `WindowProxy` object of *value*.

↪ **instance of NodeList**
↪ **instance of HTMLCollection**
↪ **instance of Array**
↪ **instance of Object**

1. If *value* is in *seen*, return error with error code javascript error.
2. Append *value* to *seen*.
3. Let *result* be the value of running the clone an object algorithm with arguments *value* and *seen*, and the internal JSON clone algorithm as the *clone algorithm*.
4. Remove the last element of *seen*.
5. Return *result*.

↪ **has an own property named "`toJSON`" that is a Function**
Return success with the value returned by Call(`toJSON`).

↪ **Otherwise**
Error with error code javascript error.

To ***clone an object***, taking the arguments *value*, *seen*, and *clone algorithm*:

1. Let *result* be the value of the first matching statement, matching on *value*:

   ↪ **instance of NodeList**
   ↪ **instance of HTMLCollection**
   ↪ **instance of Array**
   A new Array which `length` property is equal to the result of getting the property `length` of *value*.

   ↪ **Otherwise**
   A new Object.

2. For each enumerable own property in *value*, run the following substeps:

1. Let *name* be the name of the property.
2. Let *source property value* be the result of getting a property named *name* from *value*. If doing so causes script to be run and that script throws an error, return error with error code javascript error.
3. Let *cloned property result* be the result of calling the *clone algorithm* with arguments *source property value* and *seen*.
4. If *cloned property result* is a success, set a property of *result* with name *name* and value equal to *cloned property result*'s data.
5. Otherwise, return *cloned property result*.

When required to ***extract the script arguments from a request*** with argument *parameters* the implementation must:

1. Let *script* be the result of getting a property named `script` from the *parameters*.
2. If *script* is not a String, return error with error code invalid argument.
3. Let *args* be the result of getting a property named `args` from the *parameters*.
4. If *args* is not an Array return error with error code invalid argument.
5. Let *arguments* be the result of calling the JSON deserialize algorithm with arguments *args*.
6. Return success with data *script* and *arguments*.

The rules to ***execute a function body*** are as follows. The algorithm will return success with the JSON representation of the function's return value, or an error if the evaluation of the function results in a JavaScript exception being thrown.

If at any point during the algorithm a user prompt appears, abort all subsequent substeps of this algorithm, and return success with data *null*.

1. Let *window* be the associated window of the current browsing context's active document.
2. Let *environment settings* be the environment settings object for *window*.
3. Let *global scope* be *environment settings* realm's global environment.
4. If *body* is not parsable as a FunctionBody or if parsing detects an early error, return error with error code javascript error.
5. If *body* begins with a directive prologue that contains a use strict directive then let *strict* be true, otherwise let *strict* be false.

6. Prepare to run a script with *environment settings*.

7. Prepare to run a callback with *environment settings*.

8. Let *function* be the result of calling FunctionCreate, with arguments:

   kind
   Normal.

   list
   An empty List.

   body
   The result of parsing *body* above.

   global scope
   The result of parsing *global scope* above.

   strict
   The result of parsing *strict* above.

9. Let *completion* be Call(*function*, *window*, *parameters*).

10. Clean up after running a callback with *environment settings*.

11. Clean up after running a script with *environment settings*.

12. If *completion* is an abrupt completion, return an error with error code javascript error.

13. Let *json data* be a JSON clone of *completion*`.[[Value]]`.

14. Return success with data *json data*.

> NOTE
>
> The above algorithm is not associated with any particular element, and is therefore not subject to the document CSP directives.

15.2.1 *Execute Script*

HTTP Method	URI Template
POST	/session/{*session id*}/execute/sync

The remote end steps are:

1. Let *body* and *arguments* be the result of trying to extract the script arguments from a request with argument *parameters*.

2. If the current browsing context is no longer open, return error with error code no such window.

3. Handle any user prompts, and return its value if it is an error.

4. Let *promise* be a new Promise.

5. Run the following substeps in parallel:

    1. Let *result* be the result of promise-calling execute a function body, with arguments *body* and *arguments*.

    2. If *result* is an error, reject *promise* with *result*.

6. If *promise* is still pending and session script timeout milliseconds is reached, reject *promise* with error with error code script timeout.

7. Upon fulfillment of *promise* with value *v*, return success with data *v*.

8. Upon rejection of *promise* with reason *r*, if *r* is an error, return *r*. Otherwise, return error with error code javascript error and data *r*.

15.2.2 *Execute Async Script*

HTTP Method	URI Template
POST	/session/{*session id*}/execute/async

> NOTE
>
> The Execute Async Script command causes JavaScript to execute as an anonymous function. Unlike the Execute Script command, the result of the function is ignored. instead an additional argument is provided as the final argument to the function. This is a function that, when called, returns its first argument as the response.

The remote end steps are:

1. Let *body* and *arguments* by the result of trying to extract the script arguments from a request with argument *parameters*.

2. If the current browsing context is no longer open, return error with error code no such window.

3. Handle any user prompts, and return its value if it is an error.

4. Let *promise* be a new Promise.
5. Run the following substeps in parallel:
    1. Append resolve to *script arguments*.
    2. Let *result* be the result of promise-calling execute a function body, with arguments *body* and *arguments*.
    3. If *result* is an error, reject *promise* with *result*.
6. If *promise* is still pending and session script timeout milliseconds is reached, reject *promise* with error with error code script timeout.
7. Upon fulfillment of *promise* with value *v*, return success with data *v*.
8. Upon rejection of *promise* with reason *r*, if *r* is an error, return *r*. Otherwise, return error with error code javascript error and data *r*.

The ***execute async script callback*** algorithm is initialized with a single argument *webdriver callback state*. It defines a function with a single optional argument *result*. When this function is called, the following steps are run:

1. If *webdriver callback state* is not in the `unset` state, return undefined.
2. If *result* is not present, let *result* be null.
3. Let *json result* be a JSON clone of *result*.
4. Set the *webdriver callback state* to `set` with data *json result*.
5. Return undefined.

16. Cookies

This section describes the interaction with cookies as described in [RFC6265].

A cookie is described in [RFC6265] by a name-value pair holding the cookie's data, followed by zero or more attribute-value pairs describing its characteristics.

The following ***table for cookie conversion*** defines the cookie concepts relevant to WebDriver, how these are referred to in [RFC6265], what keys they map to in a serialized cookie, as well as the attribute-value keys needed when constructing a list of arguments for creating a cookie.

For informational purposes, the table includes a legend of whether the field is optional in the serialized cookie provided to Add Cookie, and a brief non-normative description of the field and

the expected input type of its associated value.

Concept	RFC 6265 Field	JSON Key	Attribute Key	Optional	Description
Cookie name	`name`	`"name"`			The name of the cookie.
Cookie value	`value`	`"value"`			The cookie value.
Cookie path	`path`	`"path"`	`"Path"`	✓	The cookie path. Defaults to `"/"` if omitted when adding a cookie.
Cookie domain	`domain`	`"domain"`	`"Domain"`	✓	The domain the cookie is visible to. Defaults to the current browsing context's active document's URL domain if omitted when adding a cookie.
Cookie secure only	`secure-only-flag`	`"secure"`	`"Secure"`	✓	Whether the cookie is a secure cookie. Defaults to false if omitted when adding a cookie.
Cookie HTTP only	`http-only-flag`	`"httpOnly"`	`"HttpOnly"`	✓	Whether the cookie is an HTTP only cookie. Defaults to false if omitted when adding a cookie.
Cookie expiry time	`expiry-time`	`"expiry"`	`"Max-Age"`	✓	When the cookie expires, specified in seconds since Unix Epoch. Must not be set if omitted when adding a cookie.

A ***serialized cookie*** is a JSON Object where a cookie's [RFC6265] fields listed in the table for cookie conversion are mapped using the *JSON Key* and the associated field's value from the cookie store. The optional fields may be omitted.

To get ***all associated cookies*** to a document, the user agent must return the enumerated set of cookies that meet the requirements set out in the first step of the algorithm in [RFC6265] to compute `cookie-string` for an 'HTTP API', from the cookie store of the given document's address. The returned cookies must include HttpOnly cookies.

When the remote end is instructed to ***create a cookie***, this is synonymous to carrying out the steps described in [RFC6265] section 5.3, under receiving a cookie, except the user agent may not ignore the received cookie in its entirety (disregard step 1).

To ***delete cookies*** given an optional filter argument *name* that is a string:

1. For each cookie among all associated cookies of the current browsing context's active document, run the substeps of the first matching condition:

   ↪ ***name* is undefined**
   ↪ ***name* is equal to cookie name**
   Set the cookie expiry time to a Unix timestamp in the past.

   ↪ **Otherwise**
   Do nothing.

16.1 *Get All Cookies*

HTTP Method	URI Template
GET	/session/{*session id*}/cookie

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts, and return its value if it is an error.
3. Let *cookies* be a new JSON List.
4. For each *cookie* in all associated cookies of the current browsing context's active document:
   1. Let *serialized cookie* be the result of serializing *cookie*.
   2. Append *serialized cookie* to *cookies*
5. Return success with data *cookies*.

16.2 *Get Named Cookie*

HTTP Method	URI Template
GET	/session/{*session id*}/cookie/{*name*}

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.

2. Handle any user prompts, and return its value if it is an error.

3. If the url variable *name* matches a cookie's cookie name amongst all associated cookies of the current browsing context's active document, return success with the serialized cookie as data.

   Otherwise, return error with error code no such cookie.

16.3 *Add Cookie*

HTTP Method	URI Template
POST	/session/{*session id*}/cookie

The remote end steps are:

1. Let *data* be the result of getting a property named `cookie` from the *parameters* argument.

2. If *data* is not a JSON Object with all the required (non-optional) JSON keys listed in the table for cookie conversion, return error with error code invalid argument.

3. If the current browsing context is no longer open, return error with error code no such window.

4. Handle any user prompts, and return its value if it is an error.

5. If the current browsing context's document element is a cookie-averse `Document` object, return error with error code invalid cookie domain.

6. If cookie name or cookie value is null, cookie domain is not equal to the current browsing context's active document's domain, cookie secure only or cookie HTTP only are not boolean types, or cookie expiry time is not an integer type, or it less than 0 or greater than the maximum safe integer, return error with error code invalid argument.

7. Create a cookie in the cookie store associated with the active document's address using cookie name *name*, cookie value *value*, and an attribute-value list of the following cookie concepts listed in the table for cookie conversion from *data*:

   Cookie path
   The value if the entry exists, otherwise "`/`".

   Cookie domain
   The value if the entry exists, otherwise the current browsing context's active document's URL domain.

   Cookie secure only
   The value if the entry exists, otherwise false.

Cookie HTTP only
The value if the entry exists, otherwise false.

Cookie expiry time
The value if the entry exists, otherwise leave unset to indicate that this is a session cookie.

If there is an error during this step, return error with error code unable to set cookie.

8. Return success with data null.

16.4 *Delete Cookie*

HTTP Method	URI Template
DELETE	/session/{*session id*}/cookie/{*name*}

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts, and return its value if it is an error.
3. Delete cookies using the url variable *name* parameter as the filter argument.
4. Return success with data null.

16.5 *Delete All Cookies*

HTTP Method	URI Template
DELETE	/session/{*session id*}/cookie

The remote end steps are:

1. If the current browsing context is no longer open, return error with error code no such window.
2. Handle any user prompts, and return its value if it is an error.
3. Delete cookies, giving no filtering argument.
4. Return success with data null.

17. *Actions*

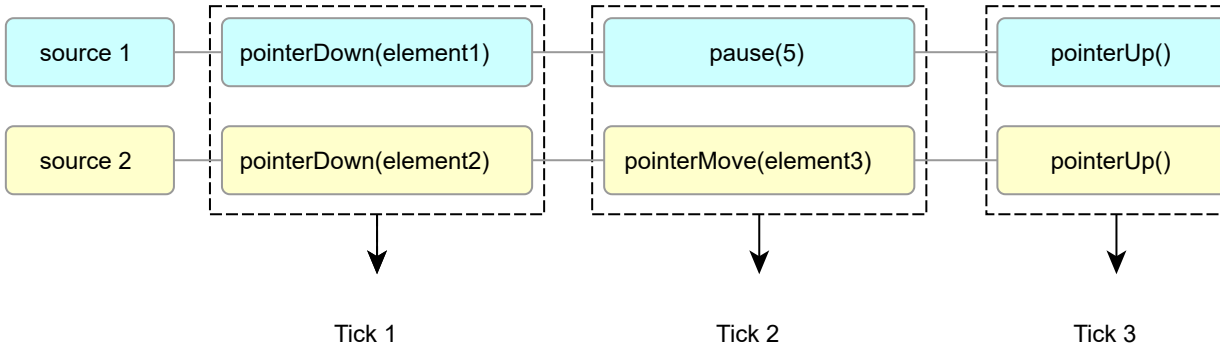
The Actions API provides a low-level interface for providing virtualised device input to the web browser. Conceptually, the Actions commands divide time into a series of ticks. The local end sends a series of actions which correspond to the change in state, if any, of each input device during each tick. For example, pressing a key is represented by an action sequence consisting of a single key input device and two ticks, the first containing a keyDown action, and the second a keyUp action, whereas a pinch-zoom input is represented by an action sequence consisting of three ticks and two pointer input devices of type touch, each performing a sequence of actions pointerDown, followed by pointerMove, and then pointerUp.

EXAMPLE 11

Imagine we have two fingers acting on a touchscreen. One finger will press down on element1 at the same moment that another finger presses down on element2. Once these actions are done, the first finger will wait 5 seconds while the other finger moves to element3. Then both fingers release from the touchscreen.

When the remote end receives this, it will look at each input source's action lists. It will dispatch the first action of each source together, then the second actions together, and lastly, the final actions together.

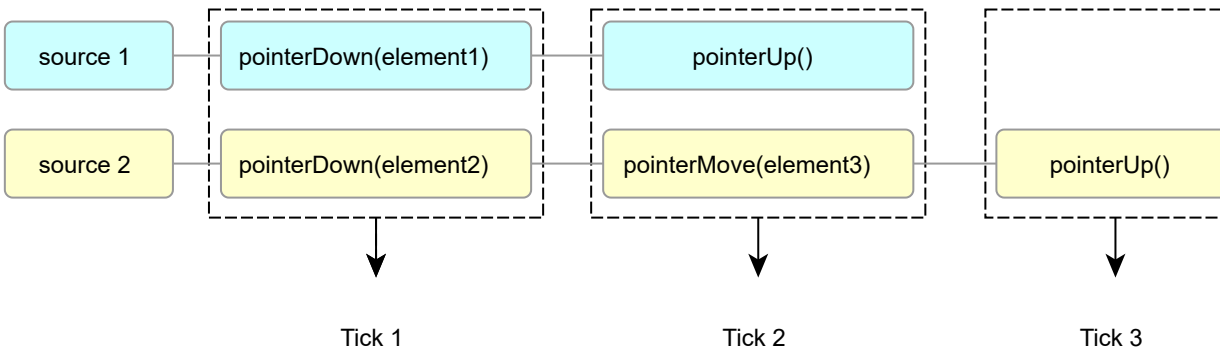
The diagram below displays when each action gets executed. "Source 1" is the first finger, and "source 2" is the second.



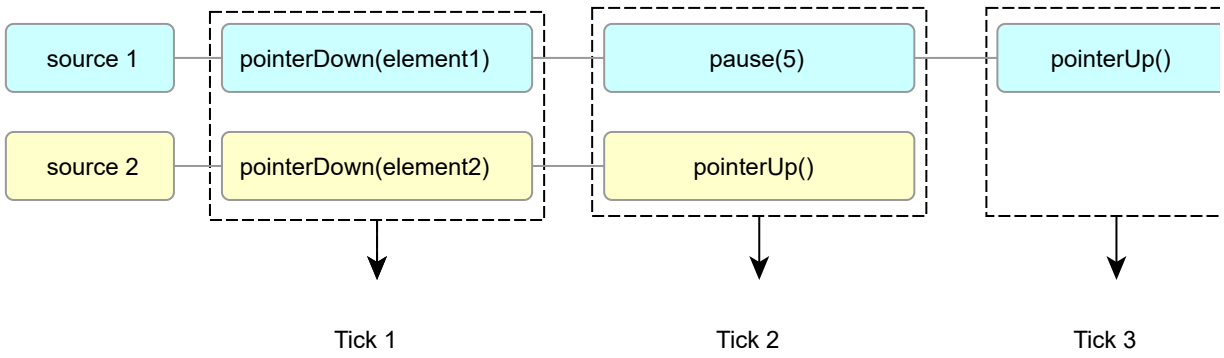
There is no limit to the number of input sources, and there is no restriction regarding the length of each input's action list. This means, there is no requirement that all action lists have to be the same length. It is possible for one input source's action list may have more actions than another.

In this case, the action list for the first finger contains 2 actions (pointerDown, pointerUp), and the action list for the second finger contains 3 (pointerDown, pointerMove, pointerUp).

And the execution of each action will be done as follows:



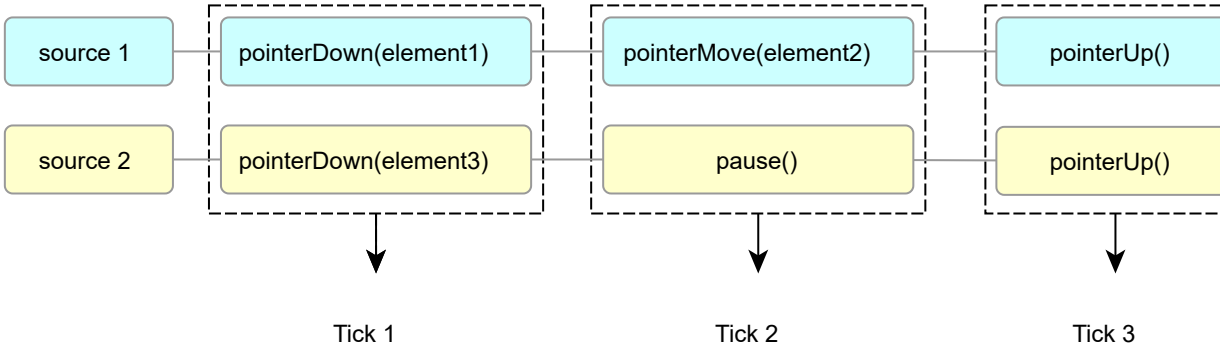
Specific timing for the actions can also be expressed. The pause action can be used to either (a) indicate a specific amount of time an input source must wait, or (b) can be used to signify that the current input source must wait until all other actions in the tick are completed. For the former case, the current tick being executed must wait for the longest pause to complete. For example, in this diagram:



The remote end will dispatch the pointerDown action in the first tick. In the second tick, since source 1 declares a pause of 5 seconds, the remote end will dispatch the pointerUp event for source 2, and will wait 5 seconds before moving on to executing the third tick.

In the event that one tick contains multiple pause durations, the remote end will wait the maximum duration before moving on to executing the next tick.

As noted before, pause can be used to signify inaction during a tick. If `pause` is declared without a time period, then the input source will not have any actions executed in the containing tick. As an example:



During tick 2, source 1 will have its pointerMove action dispatched, while source 2 will do nothing.

17.1 Terminology

An ***input source*** is a virtual device providing input events. Each input source has an associated ***input id***, which is a string that identifies the particular device, and a ***source type*** which determines the kind of input the device can provide. As with real devices, virtual devices are stateful; this state is recorded in an input source state object associated with each input source.

A ***null input source*** is an input source that is not associated with a specific physical device. A null input source supports the following actions:

Action	Non-normative Description
pause	Used with an integer argument to specify the duration of a tick, or as a placeholder to indicate that an input source does nothing during a particular tick.

A ***key input source*** is an input source that is associated with a keyboard-type device. A key input source supports the same pause action as a null input source plus the following actions:

Action	Non-normative Description
keyDown	Used to indicate that a particular key should be held down.
keyUp	Used to indicate that a depressed key should be released.

A ***pointer input source*** is an input source that is associated with a pointer-type input device. Such an input source has an associated ***pointer type*** specifying exactly which kind of pointing device it is associated with. A pointer input source supports the same pause action as a null input source plus the following actions:

Action	Non-normative Description
pointerDown	Used to indicate that a pointer should be depressed in some way e.g. by holding a button down (for a mouse) or by coming into contact with the active surface (for a touch or pen device).
pointerUp	Used to indicate that a pointer should be released in some way e.g. by releasing a mouse button or moving a pen or touch device away from the active surface.
pointerMove	Used to indicate a location on the screen that a pointer should move to, either in its active (pressed) or inactive state.
pointerCancel	Used to cancel a pointer action.

Each session maintains a list of ***active input sources***. This list is initially empty. When an input source is added to the list of active input sources, a corresponding entry is made in the input state table where the key is the input source's `input id` and the value is the input source's input source

state. When an input source is removed from the list of active input sources, the corresponding entry in the input state table is also removed.

A ***tick*** is the basic unit of time over which actions can be performed. During a tick, each input source has an assigned action — possibly a noop pause action — which may result in changes to the user agent internal state and eventually cause DOM events to be fired at the page. The next tick begins after the user agent has had a chance to process all DOM events generated in the current tick.

Waiting asynchronously means waiting for something to occur whilst allowing the browser to continue processing the event loop.

At the lowest level, the behavior of actions is intended to mimic the remote end's behavior with an actual input device as closely as possible, and the implementation strategy may involve e.g. injecting synthesized events into a browser event loop. Therefore the steps to dispatch an action will inevitably end up in implementation-specific territory. However there are certain content observable effects that must be consistent across implementations. To accommodate this, the specification requires that remote ends ***perform implementation-specific action dispatch steps***, along with a list of events and their properties. This list is not comprehensive; in particular the default action of the input source may cause additional events to be generated depending on the implementation and the state of the browser (e.g. input events relating to key actions when the focus is on an editable element, scroll events, etc.).

> NOTE
>
> An activation trigger generated by the WebDriver API user needs to be indistinguishable from those generated by a real user interacting with the browser. In particular, the dispatched events will have the isTrusted attribute set to true. The most robust way to dispatch these events is by creating them in the browser implementation itself. Sending OS-specific input messages to the browser's window has the disadvantage that the browser being automated may not be properly isolated from a user accidentally modifying input source state. Use of an OS-level accessibility API has the disadvantage that the browser's window must be focused, and as a result, multiple WebDriver instances cannot run in parallel.
>
> An advantage of an OS-level accessibility API is that it guarantees that inputs correctly mirror user input, and allows interaction with the host OS if necessary. This might, however, have performance penalties from a machine utilisation perspective.

17.2 Input Source State

> NOTE
>
> The objects and properties defined in this section are spec-internal constructs and do not correspond to ECMAScript objects. For convenience the same terminology is used for their manipulation.

"***Input source state***" is used as a generic term to describe the state associated with each input source.

The ***corresponding input source state type*** for a label *action type* is given by the following table:

Action type	Input state
"`none`"	null input state
"`key`"	key input state
"`pointer`"	pointer input state

A null input source's input source state is a ***null input state***. This is always an empty object.

A key input source's input source state is a ***key input state*** object. This is an object with a property, `pressed`, which is a set of strings representing currently pressed keys and properties `alt`, `shift`, `ctrl`, and `meta`, which are Booleans.

When required to ***create a new key input state object***, an implementation must return a key input state object with the `pressed` property set to the empty set and `alt`, `shift`, `ctrl`, and `meta` all set to `false`.

A pointer input source's input source state is a ***pointer input state*** object. This consists of a `subtype` property, which has the possible values `"mouse"`, `"pen"`, and `"touch"`, a `pressed` property which is a set of unsigned integers, an `x` property which is an unsigned integer, and a `y` property which is an unsigned integer.

When required to ***create a new pointer input state*** object with arguments *subtype* an implementation must return a pointer input state object with `subtype` set to *subtype*, `pressed` set to an empty set and both `x` and `y` set to `0`.

Each session has an associated ***input state table***. This is a map between input id and the input source state for that input source, with one entry for each item in the list of active input sources.

Each session also has an associated ***input cancel list***, which is a list of actions. This list is used to manage dispatching events when resetting the state of the input sources. For simplicity the algorithms described here only append actions to the list and rely on the fact that the reset operations are idempotent.

The ***calculated global key state*** is the aggregated key state from all key input state objects. It can be calculated this way:

1. Let *pressed* be a new Set.
2. Let *alt*, *ctrl*, *meta*, and *shift* be the Boolean `false` value.
3. For enumerable own property in the input state table:
    1. Let *source* be the value of the property.
    2. If *source* is not a key input state, continue to the first step of this loop.
    3. Let *key state pressed* be the result of getting a property named `pressed` from *source*.
    4. Add all strings from *key state pressed* to *pressed*.
    5. Let *alt* be a logical OR of *alt* and the result of getting a property named `alt` from *source*.
    6. Let *ctrl* be a logical OR of *ctrl* and the result of getting a property named `ctrl` from *source*.
    7. Let *meta* be a logical OR of *meta* and the result of getting a property named `meta` from *source*.
    8. Let *shift* be a logical OR of *shift* and the result of getting a property named `shift` from *source*.
4. Let *state* be a new JSON Object.
5. Set a property on *state* with name *pressed* and value *pressed*.
6. Set a property on *state* with name *alt* and value *alt*.
7. Set a property on *state* with name *ctrl* and value *ctrl*.
8. Set a property on *state* with name *meta* and value *meta*.
9. Set a property on *state* with name *shift* and value *shift*.
10. Return *state*.

17.3 Processing Actions Requests

The algorithm for extracting an action sequence from a request takes the JSON Object representing an action sequence, validates the input, and returns a data structure that is the transpose of the input JSON, such that the actions to be performed in a single tick are grouped together.

When required to ***extract an action sequence*** with argument *parameters*, a remote end must run the following steps:

1. Let *actions* be the result of getting a property from *parameters* named `actions`.
2. If *actions* is undefined or is not an Array, return error with error code invalid argument.
3. Let *actions by tick* be an empty List.
4. For each value *action sequence* corresponding to an indexed property in *actions*:
    1. Let *input source actions* be the result of trying to process an input source action sequence with argument *action sequence*.
    2. For each *action* in *input source actions*:
        1. Let *i* be the zero-based index of *action* in *input source actions*.
        2. If the length of *actions by tick* is less than *i* + *1*, append a new List to *actions by tick*.
        3. Append *action* to the List at index *i* in *actions by tick*.
5. Return success with data *actions by tick*.

When required to ***process an input source action sequence***, with argument *action sequence*, a remote end must run the following steps:

1. Let *type* be the result of getting a property named `type` from *action sequence*.
2. If *type* is not `"key"`, `"pointer"`, or `"none"`, return an error with error code invalid argument.
3. Let *id* be the result of getting the property `id` from *action sequence*.
4. If *id* is undefined or is not a String, return error with error code invalid argument.
5. If *type* is equal to `"pointer"`, let *parameters data* be the result of getting the property `"parameters"` from *action sequence*. Then let *parameters* be the result of trying to process pointer parameters with argument *parameters data*.
6. Let *source* be the input source in the list of active input sources where that input source's input id matches *id*, or undefined if there is no matching input source.
7. If *source* is undefined:
    1. Let *source* be a new input source created from the first match against *type*:

        ↪ `"none"`

Create a new null input source.

↪ `"key"`
Create a new key input source.

↪ `"pointer"`
Create a new pointer input source. Let the new input source's pointer type be set to the value of *parameters*'s `pointerType` property.

2. Add *source* to the current session's list of active input sources.
3. Add *source*'s input source state to the current session's input state table, keyed on *source*'s input id.

8. If *source*'s source type does not match *type* return an error with error code invalid argument.
9. If *parameters* is not undefined, then if its `pointerType` property does not match *source*'s pointer type, return an error with error code invalid argument.
10. Let *action items* be the result of getting a property named `actions` from *action sequence*.
11. If *action items* is not an Array, return error with error code invalid argument.
12. Let *actions* be a new list.
13. For each *action item* in *action items*:
    1. If *action item* is not an Object return error with error code invalid argument.
    2. If *type* is `"none"` let *action* be the result of trying to process a null action with parameters *id*, and *action item*.
    3. Otherwise, if *type* is `"key"` let *action* be the result of trying to process a key action with parameters *id*, and *action item*.
    4. Otherwise, if *type* is `"pointer"` let *action* be the result of trying to process a pointer action with parameters *id*, *parameters*, and *action item*.
    5. Append *action* to *actions*.
14. Return success with data *actions*.

The ***default pointer parameters*** consist of an object with property `pointerType` set to `mouse`.

When required to ***process pointer parameters*** with argument *parameters data*, a remote end must perform the following steps:

1. Let *parameters* be the default pointer parameters.

2. If *parameters data* is undefined, return success with data *parameters*.
3. If *parameters data* is not an Object return error with error code invalid argument.
4. Let *pointer type* be the result of getting a property named `pointerType` from *parameters data*.
5. If *pointer type* is not undefined:
    1. If *pointer type* does not have one of the values `"mouse"`, `"pen"`, or `"touch"` return error with error code invalid argument.
    2. Set the `pointerType` property of *parameters* to *pointer type*.
6. Return success with data *parameters*.

An ***action object*** constructed with arguments *id*, *type*, and *subtype* is an object with property `id` set to *id*, `type` set to *type* and `subtype` set to *subtype*. Specific action objects have further properties added by other algorithms in this specification.

When required to ***process a null action*** with arguments *id* and *action item*, a remote end must perform the following steps:

1. Let *subtype* be the result of getting a property named `type` from *action item*.
2. If *subtype* is not `"pause"` return error with error code invalid argument.
3. Let *action* be an action object constructed with arguments *id*, `"none"`, and *subtype*.
4. Let *result* be the result of trying to process a pause action with arguments *action item*, and *action*.
5. Return *result*.

When required to ***process a key action*** with arguments *id* and *action item*, a remote end must perform the following steps:

1. Let *subtype* be the result of getting a property named `type` from *action item*.
2. If *subtype* is not one of the values `"keyUp"`, `"keyDown"`, or `"pause"`, return an error with error code invalid argument.
3. Let *action* be an action object constructed with arguments *id*, `"key"`, and *subtype*.
4. If *subtype* is `"pause"`; let *result* be the result of trying to process a pause action with arguments *action item*, and *action*, and return *result*.
5. Let *key* be the result of getting a property named `value` from *action item*.

6. If *key* is not a String containing a single unicode code point or grapheme cluster? return error with error code invalid argument.

7. Set the `value` property on *action* to *key*.

8. Return success with data *action*.

When required to ***process a pointer action*** with arguments *id*, *parameters*, and *action item*, a remote end must perform the following steps:

1. Let *subtype* be the result of getting a property named `type` from *action item*.

2. If *subtype* is not one of the values `pause`, `"pointerUp"`, `"pointerDown"`, `"pointerMove"`, or `"pointerCancel"` return an error with error code invalid argument.

3. Let *action* be an action object constructed with arguments *id*, `"pointer"`, and *subtype*.

4. If *subtype* is `"pause"`; let *result* be the result of trying to process a pause action with arguments *action item*, and *action*, and return *result*.

5. Set the `pointerType` property of *action* equal to the `pointerType` property of *parameters*.

6. If *subtype* is `"pointerUp"` or `"pointerDown"`, process a pointer up or pointer down action with arguments *action item*, and *action*. If doing so results in an error, return that error.

7. If *subtype* is `"pointerMove"` process a pointer move action with arguments *action item*, and *action*. If doing so results in an error, return that error.

8. If *subtype* is `"pointerCancel"` process a pointer cancel action . If doing so results in an error, return that error.

9. Return success with data *action*.

When required to ***process a pause action*** with arguments *action item*, and *action*, a remote end must run the following steps:

1. Let *duration* be the result of getting the property "`duration`" from *action item*.

2. If *duration* is not undefined and *duration* is not an Integer greater than or equal to 0, return error with error code invalid argument.

3. Set the `duration` property of *action* to *duration*.

4. Return success with data *action*.

When required to ***process a pointer up or pointer down action*** with arguments *action item*, and *action*, a remote end must run the following steps:

1. Let *button* be the result of getting the property `button` from *action item*.
2. If *button* is not an Integer greater than or equal to 0 return error with error code invalid argument.
3. Set the `button` property of *action* to *button*.
4. Return success with data null.

When required to ***process a pointer move action*** with arguments *action item*, and *action*, a remote end must run the following steps:

1. Let *duration* be the result of getting the property `duration` from *action item*.
2. If *duration* is not undefined and *duration* is not an Integer greater than or equal to 0, return error with error code invalid argument.
3. Set the `duration` property of *action* to *duration*.
4. Let *origin* be the result of getting the property `origin` from *action item*.
5. If *origin* is undefined let *origin* equal "`viewport`".
6. If *origin* is not equal to `"viewport"` or `"pointer"` and *origin* is not an Object that represents a web element, return error with error code invalid argument.
7. Set the `origin` property of *action* to *origin*.
8. Let *x* be the result of getting the property `x` from *action item*.
9. If *x* is not undefined or is not an Integer, return error with error code invalid argument.
10. Set the `x` property of *action* to *x*.
11. Let *y* be the result of getting the property `y` from *action item*.
12. If *y* is not undefined or is not an Integer, return error with error code invalid argument.
13. Set the `y` property of *action* to *y*.
14. Return success with data null.

17.4 Dispatching Actions

The algorithm to dispatch actions takes a list of actions grouped by tick, and then causes each action to be run at the appropriate point in the sequence.

When asked to ***dispatch actions*** with argument *actions by tick*, a remote end must run the following steps:

1. For each item *tick actions* in *actions by tick*:
    1. Let *tick duration* be the result of computing the tick duration with argument *tick actions*.
    2. Dispatch tick actions with arguments *tick actions* and *tick duration*. If this results in an error return that error.
    3. Wait until the following conditions are all met:
        - There are no pending asynchronous waits arising from the last invocation of the dispatch tick actions steps.
        - The user agent event loop has spun enough times to process the DOM events generated by the last invocation of the dispatch tick actions steps.
        - At least *tick duration* milliseconds have passed.
2. Return success with data null.

When required to ***compute the tick duration*** with argument *tick actions*, a remote end must take the following steps:

1. Let *max duration* be 0.
2. For each *action object* in *tick actions*:
    1. let *duration* be undefined.
    2. If *action object* has `subtype` property set to `"pause"` or *action object* has `type` property set to `"pointer"` and `subtype` property set to `"pointerMove"`, let *duration* be equal to the `duration` property of *action object*.
    3. If *duration* is not undefined, and *duration* is greater than *max duration*, let *max duration* be equal to duration.
3. Return *max duration*.

When required to ***dispatch tick actions*** with arguments *tick actions* and *tick duration*, a remote end must run the following steps:

1. For each *action object* in *tick actions*:
    1. Let *source id* be equal to the value of *action object*'s `id` property.
    2. Let *source type* be equal to the value of *action object*'s `type` property.

3. If the current session's input state table doesn't have a property corresponding to *source id*, then let the property corresponding to *source id* be a new object of the corresponding input source state type for *source type*.

4. Let *device state* be the input source state corresponding to *source id* in the current session's input state table.

5. Let *algorithm* be the value of the column *dispatch action algorithm* from the following table of ***dispatch action algorithms*** that matches the *source type* and the *action object*'s `subtype` property, to a dispatch action algorithm.

source type	`subtype` property	Dispatch action algorithm
`"none"`	`"pause"`	Dispatch a pause action
`"key"`	`"pause"`	Dispatch a pause action
`"key"`	`"keyDown"`	Dispatch a keyDown action
`"key"`	`"keyUp"`	Dispatch a keyUp action
`"pointer"`	`"pause"`	Dispatch a pause action
`"pointer"`	`"pointerDown"`	Dispatch a pointerDown action
`"pointer"`	`"pointerUp"`	Dispatch a pointerUp action
`"pointer"`	`"pointerMove"`	Dispatch a pointerMove action
`"pointer"`	`"pointerCancel"`	Dispatch a pointerCancel action

2. Return the result of running *algorithm* with arguments *source id*, *action object*, *device state* and *tick duration*.

17.4.1 General Actions

When required to ***dispatch a pause action*** with arguments *source id*, *action object*, *input state* and *tick duration* a remote end must run the following steps:

1. Return success with data null.

17.4.2 Keyboard Actions

The ***normalised key value*** for a raw key *key* is, if *key* appears in the table below, the string value in the second column on the row containing *key*'s unicode code point in the first column, otherwise it is *key*.

key's codepoint	Normalised key value

\uE000	"Unidentified"
\uE001	"Cancel"
\uE002	"Help"
\uE003	"Backspace"
\uE004	"Tab"
\uE005	"Clear"
\uE006	"Return"
\uE007	"Enter"
\uE008	"Shift"
\uE009	"Control"
\uE00A	"Alt"
\uE00B	"Pause"
\uE00C	"Escape"
\uE00D	" "
\uE00E	"PageUp"
\uE00F	"PageDown"
\uE010	"End"
\uE011	"Home"
\uE012	"ArrowLeft"
\uE013	"ArrowUp"
\uE014	"ArrowRight"
\uE015	"ArrowDown"
\uE016	"Insert"
\uE017	"Delete"
\uE018	";"
\uE019	"="
\uE01A	"0"
\uE01B	"1"
\uE01C	"2"
\uE01D	"3"
\uE01E	"4"
\uE01F	"5"

\uE020	"6"
\uE021	"7"
\uE022	"8"
\uE023	"9"
\uE024	"*"
\uE025	"+"
\uE026	","
\uE027	"-"
\uE028	"."
\uE029	"/"
\uE031	"F1"
\uE032	"F2"
\uE033	"F3"
\uE034	"F4"
\uE035	"F5"
\uE036	"F6"
\uE037	"F7"
\uE038	"F8"
\uE039	"F9"
\uE03A	"F10"
\uE03B	"F11"
\uE03C	"F12"
\uE03D	"Meta"
\uE040	"ZenkakuHankaku"
\uE050	"Shift"
\uE051	"Control"
\uE052	"Alt"
\uE053	"Meta"
\uE054	"PageUp"
\uE055	"PageDown"
\uE056	"End"
\uE057	"Home"

\uE058	"ArrowLeft"
\uE059	"ArrowUp"
\uE05A	"ArrowRight"
\uE05B	"ArrowDown"
\uE05C	"Insert"
\uE05D	"Delete"

The ***code*** for *key* is the value in the last column of the following table on the row with *key* in either the first or second column, if any such row exists, otherwise it is undefined.

A ***shifted character*** is one that appears in the second column of the following table.

Key	Alternate Key	code
"`"	"~"	"Backquote"
"\"	"|"	"Backslash"
"\uE003"		"Backspace"
"["	"{"	"BracketLeft"
"}"	"]"	"BracketRight"
","	"<"	"Comma"
"0"	")"	"Digit0"
"1"	"!"	"Digit1"
"2"	"@"	"Digit2"
"3"	"#"	"Digit3"
"4"	"$"	"Digit4"
"5"	"%"	"Digit5"
"6"	"^"	"Digit6"
"7"	"&"	"Digit7"
"8"	"*"	"Digit8"
"9"	"("	"Digit9"
"="	"+"	"Equal"
"<"	">"	"IntlBackslash"
"a"	"A"	"KeyA"
"b"	"B"	"KeyB"
"c"	"C"	"KeyC"

"d"	"D"	"KeyD"
"e"	"E"	"KeyE"
"f"	"F"	"KeyF"
"g"	"G"	"KeyG"
"h"	"H"	"KeyH"
"i"	"I"	"KeyI"
"j"	"J"	"KeyJ"
"k"	"K"	"KeyK"
"l"	"L"	"KeyL"
"m"	"M"	"KeyM"
"n"	"N"	"KeyN"
"o"	"O"	"KeyO"
"p"	"P"	"KeyP"
"q"	"Q"	"KeyQ"
"r"	"R"	"KeyR"
"s"	"S"	"KeyS"
"t"	"T"	"KeyT"
"u"	"U"	"KeyU"
"v"	"V"	"KeyV"
"w"	"W"	"KeyW"
"x"	"X"	"KeyX"
"y"	"Y"	"KeyY"
"z"	"Z"	"KeyZ"
"-"	"_"	"Minus"
"."	">"	"Period"
"'"	"\""	"Quote"
";"	":"	"Semicolon"
"/"	"?"	"Slash"
"\uE00A"		"AltLeft"
"\uE052"		"AltRight"
"\uE009"		"ControlLeft"
"\uE051"		"ControlRight"

"\uE006"		"Enter"
"\uE03D"		"OSLeft"
"\uE053"		"OSRight"
"\uE008"		"ShiftLeft"
"\uE050"		"ShiftRight"
" "	"\uE00D"	"Space"
"\uE004"		"Tab"
"\uE017"		"Delete"
"\uE010"		"End"
"\uE002"		"Help"
"\uE011"		"Home"
"\uE016"		"Insert"
"\uE01E"		"PageDown"
"\uE01F"		"PageUp"
"\uE015"		"ArrowDown"
"\uE012"		"ArrowLeft"
"\uE014"		"ArrowRight"
"\uE013"		"ArrowUp"
"\uE00C"		"Escape"
"\uE031"		"F1"
"\uE032"		"F2"
"\uE033"		"F3"
"\uE034"		"F4"
"\uE035"		"F5"
"\uE036"		"F6"
"\uE037"		"F7"
"\uE038"		"F8"
"\uE039"		"F9"
"\uE03A"		"F10"
"\uE03B"		"F11"
"\uE03C"		"F12"
"\uE01A"	"\uE05C"	"Numpad0"

"\uE01B"	"\uE056"	"Numpad1"
"\uE01C"	"\uE05B"	"Numpad2"
"\uE01D"	"\uE055"	"Numpad3"
"\uE01E"	"\uE058"	"Numpad4"
"\uE01F"		"Numpad5"
"\uE020"	"\uE05A"	"Numpad6"
"\uE021"	"\uE057"	"Numpad7"
"\uE022"	"\uE059"	"Numpad8"
"\uE023"	"\uE054"	"Numpad9"
"\uE024"		"NumpadAdd"
"\uE026"		"NumpadComma"
"\uE028"	"\uE05D"	"NumpadDecimal"
"\uE029"		"NumpadDivide"
"\uE007"		"NumpadEnter"
"\uE024"		"NumpadMultiply"
"\uE026"		"NumpadSubtract"

The ***key location*** for *key* is the value in the last column in the table below on the row with *key* appears in the first column, if such a row exists, otherwise it is `0`.

key's codepoint	Description	Location
\uE007	Enter	1
\uE008	Left Shift	1
\uE009	Left Control	1
\uE00A	Left Alt	1
\uE01A	Numpad 0	3
\uE01B	Numpad 1	3
\uE01C	Numpad 2	3
\uE01D	Numpad 3	3
\uE01E	Numpad 4	3
\uE01F	Numpad 5	3
\uE020	Numpad 6	3
\uE021	Numpad 7	3
\uE022	Numpad 8	3

\uE023	Numpad 9	3
\uE024	Numpad *	3
\uE025	Numpad +	3
\uE026	Numpad ,	3
\uE027	Numpad -	3
\uE028	Numpad .	3
\uE029	Numpad /	3
\uE03D	Left Meta	1
\uE050	Right Shift	2
\uE051	Right Control	2
\uE052	Right Alt	2
\uE053	Right Meta	2
\uE054	Numpad PageUp	3
\uE055	Numpad PageDown	3
\uE056	Numpad End	3
\uE057	Numpad Home	3
\uE058	Numpad ArrowLeft	3
\uE059	Numpad ArrowUp	3
\uE05A	Numpad ArrowRight	3
\uE05B	Numpad ArrowDown	3
\uE05C	Numpad Insert	3
\uE05D	Numpad Delete	3

When required to ***dispatch a keyDown action*** with arguments *source id*, *action object*, *input state* and *tick duration* a remote end must run the following steps:

1. Let *raw key* be equal to the *action object*'s `value` property.
2. Let *key* be equal to the normalised key value for *raw key*.
3. If the *input state*'s `pressed` property contains *key*, let *repeat* be true, otherwise let *repeat* be false.
4. Let *code* be the code for *raw key*.
5. Let *location* be the key location for *raw key*.

6. Let *charCode*, *keyCode* and *which* be the implementation-specific values of the `charCode`, `keyCode` and `which` properties appropriate for a key with key *key* and location *location* on a 102 key US keyboard, following the guidelines in [UI-EVENTS].

7. If *key* is `"Alt"`, let *device state's* `alt` property be true.

8. If *key* is `"Shift"`, let *device state's* `shift` property be true.

9. If *key* is `"Control"`, let *device state's* `ctrl` property be true.

10. If *key* is `"Meta"`, let *device state's* `meta` property be true.

11. Add *key* to the set corresponding to *input state*'s `pressed` property.

12. Append a copy of *action object* with the *subtype* property changed to *keyUp* to current session's input cancel list.

13. Perform implementation-specific action dispatch steps equivalent to pressing a key on the keyboard in accordance with the requirements of [UI-EVENTS], and producing the following events, as appropriate, with the specified properties. This will always produce events including at least a `keyDown` event.

    - `keyDown` with properties:

Attribute	Value
`key`	*key*
`code`	*code*
`location`	*location*
`altKey`	*device state*'s `alt` property
`shiftKey`	*device state*'s `shift` property
`ctrlKey`	*device state*'s `ctrl` property
`metaKey`	*device state*'s `meta` property
`repeat`	*repeat*
`isComposing`	*false*
`charCode`	*charCode*
`keyCode`	*keyCode*
`which`	*which*

    - `keyPress` with properties:

Attribute	Value
`key`	*key*

`code`	*code*
`location`	*location*
`altKey`	*device state*'s `alt` property
`shiftKey`	*device state*'s `shift` property
`ctrlKey`	*device state*'s `ctrl` property
`metaKey`	*device state*'s `meta` property
`repeat`	*repeat*
`isComposing`	*false*
`charCode`	*charCode*
`keyCode`	*keyCode*
`which`	*which*

14. Return success with data null.

> NOTE
>
> A single keyDown action produces a single key input, irrespective of how long the key is held down; there is no implicit key repetition.

When required to ***dispatch a keyUp action*** with arguments *source id*, *action object*, *input state* and *tick duration* a remote end must run the following steps:

1. Let *raw key* be equal to *action object*'s `value` property.
2. Let *key* be equal to the normalised key value for *raw key*.
3. If the *input state*'s `pressed` property does not contain *key*, return.
4. Let *code* be the code for *raw key*.
5. Let *location* be the key location for *raw key*.
6. Let *charCode*, *keyCode* and *which* be the implementation-specific values of the `charCode`, `keyCode` and `which` properties appropriate for a key with key *key* and location *location* on a 102 key US keyboard, following the guidelines in [UI-EVENTS].
7. If *key* is `"Alt"`, let *device state's* `alt` property be false.
8. If *key* is `"Shift"`, let *device state's* `shift` property be false.
9. If *key* is `"Control"`, let *device state's* `ctrl` property be false.

10. If *key* is `"Meta"`, let *device state's* `meta` property be false.

11. Remove *key* from the set corresponding to *input state*'s `pressed` property.

12. Perform implementation-specific action dispatch steps equivalent to releasing a key on the keyboard in accordance with the requirements of [UI-EVENTS], and producing at least the following events with the specified properties:

    - `keyup`, with properties:

Attribute	Value
`key`	*key*
`code`	*code*
`location`	*location*
`altKey`	*device state*'s `alt` property
`shiftKey`	*device state*'s `shift` property
`ctrlKey`	*device state*'s `ctrl` property
`metaKey`	*device state*'s `meta` property
`repeat`	*false*
`isComposing`	*false*
`charCode`	*charCode*
`keyCode`	*keyCode*
`which`	*which*

13. Return success with data null.

17.4.3 Pointer Actions

When required to ***dispatch a pointerDown action*** with arguments *source id*, *action object*, *input state* and *tick duration* a remote end must run the following steps:

1. Let *pointerType* be equal to *action object*'s `pointerType` property.

2. Let *button* be equal to *action object*'s `button` property.

3. If the *input state*'s `pressed` property contains *button* return success with data null.

4. Let *x* be equal to *input state*'s `x` property.

5. Let *y* be equal to *input state*'s `y` property.

6. Add *button* to the set corresponding to *input state*'s `pressed` property, and let *buttons* be the resulting value of that property.

7. Append a copy of *action object* with the *subtype* property changed to *pointerUp* to the current session's input cancel list.

8. Perform implementation-specific action dispatch steps equivalent to pressing the button numbered *button* on the pointer with ID *source id*, having type type *pointerType* at viewport x coordinate *x*, viewport y coordinate *y*, with buttons *buttons* depressed in accordance with the requirements of [UI-EVENTS] and [POINTER-EVENTS]. The generated events must set `ctrlKey`, `shiftKey`, `altKey`, and `metaKey` from the calculated global key state. Type specific properties for the pointer that are not exposed through the webdriver API must be set to the default value specified for hardware that doesn't support that property.

9. Return success with data null.

When required to ***dispatch a pointerUp action*** with arguments *source id*, *action object*, *input state* and *tick duration* a remote end must run the following steps:

1. Let *pointerType* be equal to *action object*'s `pointerType` property.

2. Let *button* be equal to *action object*'s `button` property.

3. If the *input state*'s `pressed` property does not contain *button*, return success with data null.

4. Let *x* be equal to *input state*'s `x` property.

5. Let *y* be equal to *input state*'s `y` property.

6. Remove *button* from the set corresponding to *input state*'s `pressed` property, and let *buttons* be the resulting value of that property.

7. Perform implementation-specific action dispatch steps equivalent to releasing the button numbered *button* on the pointer of ID *source id* having type type *pointerType* at viewport x coordinate *x*, viewport y coordinate *y*, with buttons *buttons* depressed, in accordance with the requirements of [UI-EVENTS] and [POINTER-EVENTS]. The generated events must set `ctrlKey`, `shiftKey`, `altKey`, and `metaKey` from the calculated global key state. Type specific properties for the pointer that are not exposed through the webdriver API must be set to the default value specified for hardware that doesn't support that property.

8. Return success with data null.

When required to ***dispatch a pointerMove action*** with arguments *source id*, *action object*, *input state* and *tick duration* a remote end must run the following steps:

1. Let *x offset* be equal to the `x` property of *action object*.

2. Let *y offset* be equal to the `y` property of *action object*.

3. Let *start x* be equal to the `x` property of *input state*.

4. Let *start y* be equal to the `y` property of *input state*.

5. Let *origin* be equal to the `origin` property of *action object*.

6. Run the substeps of the first matching value of *origin*:

   `"viewport"`
   : Let *x* equal *x offset* and *y* equal *y offset*.

   `"pointer"`
   : Let *x* equal *start x* + *x offset* and *y* equal *start y* + *y offset*.

   An object that represents a web element

   1. Let *element* be equal to the result of trying to get a known connected element with argument *origin*.

   2. Let *x element* and *y element* be the result of calculating the in-view center point of *element*.

   3. Let *x* equal *x element* + *x offset*, and *y* equal *y element* + *y offset*.

7. If *x* is less than 0 or greater than the width of the viewport in CSS pixels, then return error with error code move target out of bounds.

8. If *y* is less than 0 or greater than the height of the viewport in CSS pixels, then return error with error code move target out of bounds.

9. Let *duration* be equal to *action object*'s `duration` property if it is not undefined, or *tick duration* otherwise.

10. If *duration* is greater than 0 and inside any implementation-defined bounds, asynchronously wait for an implementation defined amount of time to pass.

    This wait allows the implementation to model the overall pointer move as a series of small movements occurring at an implementation defined rate (e.g. one movement per vsync).

11. Perform a pointer move with arguments *source id*, *input state*, *duration*, *start x*, *start y*, *x*, *y*.

When required to ***perform a pointer move*** with arguments *source id*, *input state*, *duration*, *start x*, *start y*, *target x* and *target y*, an implementation must run the following steps:

1. Let *subtype* equal the `subtype` property of *input state*.

2. Let *time delta* be the time since the beginning of the current tick, measured in milliseconds on a monotonic clock.

3. Let *duration ratio* be the ratio of *time delta* and *duration*, if *duration* is greater than 0, or 1 otherwise.

4. If *duration ratio* is 1, or close enough to 1 that the implementation will not further subdivide the move action, let *last* be true. Otherwise let *last* be `false`.

5. If *last* is true, let *x* equal *target x* and *y* equal *target y*.

   Otherwise let *x* equal an approximation to *duration ratio* × (*target x* - *start x*) + *start x*, and *y* equal an approximation to *duration ratio* × (*target y* - *start y*) + *start y*.

6. Let *current x* equal the `x` property of *input state*.

7. Let *current y* equal the `y` property of *input state*.

8. If *x* is not equal to *current x* or *y* is not equal to *current y*, run the following steps:

   1. Let *buttons* be equal to input state's `buttons` property.

   2. Perform implementation-specific action dispatch steps equivalent to moving the pointer with ID *source id* having type *pointerType* from viewport x coordinate *current x*, viewport y coordinate *y* to viewport x coordinate *x* and viewport y coordinate *y*, with buttons *buttons* depressed, in accordance with the requirements of [UI-EVENTS] and [POINTER-EVENTS]. The generated events must set `ctrlKey`, `shiftKey`, `altKey`, and `metaKey` from the calculated global key state. Type specific properties for the pointer that are not exposed through the WebDriver API must be set to the default value specified for hardware that doesn't support that property. In the case where the *pointerType* is `"pen"` or `"touch"`, and *buttons* is empty, this may be a no-op. For a pointer of type `"mouse"`, this will always produce events including at least a `pointerMove` event.

   3. Let *input state*'s `x` property equal *x* and `y` property equal *y*.

9. If *last* is true, return.

10. Asynchronously wait for an implementation defined amount of time to pass.

    This wait allows the implementation to model the overall pointer move as a series of small movements occurring at an implementation defined rate (e.g. one movement per vsync).

11. Perform a pointer move with arguments *source id*, *input state*, *duration*, *start x*, *start y*, *target x*, *target y*.

When required to ***dispatch a pointerCancel action*** with arguments *source id*, *action object*, *input state* and *tick duration* a remote end must run the following steps:

1. Perform implementation-specific action dispatch steps equivalent to cancelling the any action of the pointer with ID *source id* having type *pointerType*, in accordance with the requirements of [UI-EVENTS] and [POINTER-EVENTS].

17.5 *Perform Actions*

HTTP Method	URI Template
POST	/session/{*session id*}/actions

The remote end steps are:

1. Let *actions by tick* be the result of trying to extract an action sequence with argument *parameters*.
2. If the current browsing context is no longer open, return error with error code no such window.
3. Handle any user prompts. If this results in an error, return that error.
4. Dispatch actions with argument *actions by tick*. If this results in an error return that error.
5. Return success with data null.

17.6 *Release Actions*

HTTP Method	URI Template
DELETE	/session/{*session id*}/actions

> NOTE
>
> The Release Actions command is used to release all the keys and pointer buttons that are currently depressed. This causes events to be fired as if the state was released by an explicit series of actions. It also clears all the internal state of the virtual devices.

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. Let *undo actions* be equal to the current session's input cancel list in reverse order.

3. Dispatch tick actions with arguments *undo actions* and duration 0.

4. Let the current session's input cancel list be an empty List.

5. Let the current session's input state table be an empty map.

6. Let the current session's active input sources be an empty list.

7. Return success with data null.

18. User Prompts

This chapter describes interaction with various types of user prompts. The common denominator for user prompts is that they are modal windows requiring users to interact with them before the event loop is unpaused and control is returned to the current top-level browsing context.

By default user prompts are not handled automatically unless a user prompt handler has been defined. When a user prompt appears, it is the task of the subsequent command to handle it. If the subsequent requested command is not one listed in this chapter, an unexpected alert open error will be returned.

User prompts that are spawned from `beforeunload` event handlers, are dismissed implicitly upon navigation or close window, regardless of the defined user prompt handler.

A user prompt has an associated ***user prompt message*** that is the string message shown to the user, or null if the message length is `0`.

The following ***table of simple dialogs*** enumerates all supported simple dialogs, along with the commands that are allowed to interact with it as a non-normative reference:

Definition	Dialog	Interactions
Alert	window.`alert`	Accept Alert Dismiss Alert Get Alert Text
Confirm	window.`confirm`	Dismiss Alert Accept Alert Get Alert Text
Prompt	window.`prompt`	Dismiss Alert Accept Alert Get Alert Text Send Alert Text

The ***current user prompt*** is said to be the active user prompt, which can be one of the entries on the table of simple dialogs.

To ***dismiss*** the current user prompt, do so as if the user would click the **Cancel** or **OK** button, whichever is present, in that order.

To ***accept*** the current user prompt, do so as if the user would click the **OK** button.

A ***user prompt handler*** is an enumerated attribute defining what action the remote end must take when a user prompt is encountered. This is defined by the unhandled prompt behavior capability. The following ***known prompt handling approaches table*** lists the keywords and states for the attribute:

Keyword	State	Description
"`dismiss`"	***Dismiss state***	All simple dialogs encountered should be dismissed.
"`accept`"	***Accept state***	All simple dialogs encountered should be accepted.
"`dismiss and notify`"	***Dismiss and notify state***	All simple dialogs encountered should be dismissed, and an error returned that the dialog was handled.
"`accept and notify`"	***Accept and notify state***	All simple dialogs encountered should be accepted, and an error returned that the dialog was handled.
"`ignore`"	***Ignore state***	All simple dialogs encountered should be left to the user to handle.

When required to ***deserialize as an unhandled prompt behavior*** an argument *value*:

1. If *value* is not a string return an error with error code invalid argument.
2. If *value* is not present as a `keyword` in the known prompt handling approaches table return an error with error code invalid argument.
3. Return success with data *value*.

An ***annotated unexpected alert open error*** is an error with error code unexpected alert open and an optional error data dictionary with the following entries:

"`text`"
The current user prompt's message.

In order to ***handle any user prompts*** a remote end must take the following steps:

1. If there is no current user prompt, abort these steps and return success.
2. Perform the following substeps based on the current session's user prompt handler:

   ↪ **dismiss state**

Dismiss the current user prompt.

↪ **accept state**

Accept the current user prompt.

↪ **dismiss and notify state**

1. Dismiss the current user prompt.

2. Return an annotated unexpected alert open error.

↪ **accept and notify state**

1. Accept the current user prompt.

2. Return an annotated unexpected alert open error.

↪ **ignore state**

Return an annotated unexpected alert open error.

↪ **missing value default state**
↪ **not in the table of simple dialogs**

1. Dismiss the current user prompt.

2. Return an annotated unexpected alert open error.

3. Return success.

EXAMPLE 12

When returning an error with unexpected alert open, a remote end may choose to return the user prompt message as part of an additional "`data`" Object on the error representation:

```
{
        "error": "unexpected alert open",
        "message": "implementation defined",
        "stacktrace": "",
        "data": {
                "text": "the text from the alert"
        }
}
```

18.1 *Dismiss Alert*

HTTP Method	URI Template
POST	/session/{*session id*}/alert/dismiss

> NOTE
>
> The Dismiss Alert command dismisses a simple dialog if present. A request to dismiss an alert user prompt, which may not necessarily have a dismiss button, has the same effect as accepting it.

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. If there is no current user prompt, return error with error code no such alert.
3. Dismiss the current user prompt.
4. Return success with data null.

18.2 *Accept Alert*

HTTP Method	URI Template
POST	/session/{*session id*}/alert/accept

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. If there is no current user prompt, return error with error code no such alert.
3. Accept the current user prompt.
4. Return success with data null.

18.3 *Get Alert Text*

HTTP Method	URI Template
GET	/session/{*session id*}/alert/text

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.
2. If there is no current user prompt, return error with error code no such alert.
3. Let *message* be the text message associated with the current user prompt, or otherwise be null.
4. Return success with data *message*.

18.4 *Send Alert Text*

HTTP Method	URI Template
POST	/session/{*session id*}/alert/text

> NOTE
>
> The Send Alert Text command sets the text field of a window.`prompt` user prompt to the given value.

The remote end steps are:

1. Let *text* be the result of getting the property "`text`" from *parameters*.
2. If *text* is not a String, return error with error code invalid argument.
3. If the current top-level browsing context is no longer open, return error with error code no such window.
4. If there is no current user prompt, return error with error code no such alert.
5. Run the substeps of the first matching current user prompt:

   ↪ **alert**
   ↪ **confirm**
   Return error with error code element not interactable.

   ↪ **prompt**
   Do nothing.

   ↪ **Otherwise**
   Return error with error code unsupported operation.

6. Perform user agent dependent steps to set the value of current user prompt's text field to *text*.
7. Return success with data null.

19. Screen Capture

Screenshots are a mechanism for providing additional visual diagnostic information. They work by dumping a snapshot of the initial viewport's framebuffer as a lossless PNG image. It is returned to the local end as a Base64 encoded string.

WebDriver provides the Take Screenshot command to capture the top-level browsing context's initial viewport, and a command Take Element Screenshot for doing the same with the visible region of an element's bounding rectangle after it has been scrolled into view.

In order to ***draw a bounding box from the framebuffer***, given a rectangle:

1. If either the initial viewport's width or height is 0 CSS pixels, return error with error code unable to capture screen.
2. Let *paint width* be the initial viewport's width – min(rectangle x coordinate, rectangle x coordinate + rectangle width dimension).
3. Let *paint height* be the initial viewport's height – min(rectangle y coordinate, rectangle y coordinate + rectangle height dimension).
4. Let *canvas* be a new `canvas` element, and set its width and height to *paint width* and *paint height*, respectively.
5. Let *context*, a canvas context mode, be the result of invoking the 2D context creation algorithm given *canvas* as the target.
6. Complete implementation specific steps equivalent to drawing the region of the framebuffer specified by the following coordinates onto *context*:

   X coordinate
   : rectangle x coordinate

   Y coordinate
   : rectangle y coordinate

   Width
   : *paint width*

   Height
   : *paint height*

7. Return success with *canvas*.

To ***encode as Base64 a `canvas` element***:

1. If the `canvas` element's bitmap's origin-clean flag is set to false, return error with error code unable to capture screen.

2. If the `canvas` element's bitmap has no pixels (i.e. either its horizontal dimension or vertical dimension is zero) then return error with error code unable to capture screen.

3. Let *file* be a serialization of the `canvas` element's bitmap as a file, using "`image/png`" as an argument.

4. Let *data url* be a `data:` URL representing *file*. [RFC2397]

5. Let *index* be the index of "`,`" in *data url*.

6. Let *encoded string* be a substring of *data url* using (*index* + 1) as the *start* argument.

7. Return success with data *encoded string*.

19.1 *Take Screenshot*

HTTP Method	URI Template
GET	/session/{*session id*}/screenshot

The remote end steps are:

1. If the current top-level browsing context is no longer open, return error with error code no such window.

2. When the user agent is next to run the animation frame callbacks:

    1. Let *root rect* be the current top-level browsing context's document element's rectangle.

    2. Let *screenshot* be the result of trying to call draw a bounding box from the framebuffer, given *root rect* as an argument.

    3. Let *canvas* be a `canvas` element of *screenshot result*'s data.

    4. Let *encoding* be the result of trying encoding as Base64 *canvas*.

    5. Let *encoded string* be *encoding result*'s data.

3. Return success with data *encoded string*.

19.2 *Take Element Screenshot*

HTTP Method	URI Template
GET	/session/{*session id*}/element/{*element id*}/screenshot

> NOTE
>
> The Take Element Screenshot command takes a screenshot of the visible region encompassed by the bounding rectangle of an element. If given a parameter argument `scroll` that evaluates to false, the element will not be scrolled into view.

The remote end steps are:

1. Let *scroll* be the result of getting the property `scroll` from *parameters* if it is not undefined. Otherwise let it be true.
2. If the current browsing context is no longer open, return error with error code no such window.
3. Let *element* be the result of trying to get a known connected element with url variable *element id*.
4. If asked to *scroll*, scroll into view the *element*.
5. When the user agent is next to run the animation frame callbacks:
    1. Let *element rect* be *element*'s rectangle.
    2. Let *screenshot result* be the result of trying to call draw a bounding box from the framebuffer, given *element rect* as an argument.
    3. Let *canvas* be a `canvas` element of *screenshot result*'s data.
    4. Let *encoding result* be the result of trying encoding as Base64 *canvas*.
    5. Let *encoded string* be *encoding result*'s data.
6. Return success with data *encoded string*.

A. Privacy Considerations

It is advisable that remote ends create a new profile when creating a new session. This prevents potentially sensitive session data from being accessible to new sessions, ensuring both privacy and preventing state from bleeding through to the next session.

B. Security Considerations

A user agent can rely on a command-line flag or a configuration option to test whether to enable WebDriver, or alternatively make the user agent initiate or confirm the connection through a privileged content document or control widget, in case the user agent does not directly implement the HTTP endpoints.

It is strongly suggested that user agents require users to take explicit action to enable WebDriver, and that WebDriver remains disabled in publicly consumed versions of the user agent.

To prevent arbitrary machines on the network from connecting and creating sessions, it is suggested that only connections from loopback devices are allowed by default.

The remote end can include a configuration option to limit the accepted IP range allowed to connect and make requests. The default setting for this might be to limit connections to the IPv4 localhost CIDR range `127.0.0.0/8` and the IPv6 localhost address `::1`. [RFC4632]

It is also suggested that user agents make an effort to visually distinguish a user agent session that is under control of WebDriver from those used for normal browsing sessions. This can be done through a browser chrome element such as a “door hanger”, colorful decoration of the OS window, or some widget element that is prevalent in the window so that it easy to identify automation windows.

C. Element Displayedness

Although WebDriver does not define a primitive to ascertain the visibility of an element in the viewport, we acknowledge that it is an important feature for many users. Here we include a recommended approach which will give a simplified approximation of an element’s visibility, but please note that it relies only on tree-traversal, and only covers a subset of visibility checks.

The visibility of an element is guided by what is perceptually visible to the human eye. In this context, an element's displayedness does not relate to the `visibility` or `display` style properties.

The approach recommended to implementors to ascertain an element's visibility was originally developed by the Selenium project, and is based on crude approximations about an element's nature and relationship in the tree. An element is in general to be considered visible if any part of it is drawn on the canvas within the boundaries of the viewport.

The ***element displayed*** algorithm is a boolean state where `true` signifies that the element is displayed and `false` signifies that the element is not displayed. To compute the state on *element*, invoke the Call(`bot.dom.isShown`, null, *element*). If doing so does not produce an error, return the return value from this function call. Otherwise return an error with error code unknown error.

This function is typically exposed to `GET` requests with a URI Template of `/session/{session id}/element/{element id}/displayed`.

D. Acknowledgements

There have been a lot of people that have helped make browser automation possible over the years and thereby furthered the goals of this standard. In particular, thanks goes to the Selenium Open Source community, without which this standard would never have been possible.

This standard is authored by Aleksey Chemakin, Andreas Tolfsen, Andrey Botalov, Clayton Martin, Daniel Wagner-Hall, David Burns, Eran Messeri, Erik Wilde, Gábor Csárdi, Sam Sneddon, James Graham, Jason Juang, Jason Leyba, Jim Evans, John Jansen, Luke Inman-Semerau, Maja Frydrychowicz, Malini Das, Marc Fisher, Mike Pennisi, Ondřej Machulda, Randall Kent, Seva Lotoshnikov, Simon Stewart, Titus Fortner, and Vangelis Katsikaros. The work is coordinated and edited by David Burns and Simon Stewart.

Thanks to Berge Schwebs Bjørlo, Lukas Tetzlaff, Malcolm Rowe, Michael[tm] Smith, Nathan Bloomfield, Philippe Le Hégaret, Robin Berjon, Ross Patterson, and Wilhelm Joys Andersen for proofreading and suggesting areas for improvement.

E. References

E.1 Normative references

[CSP3]
Content Security Policy Level 3. Mike West. W3C. 13 September 2016. W3C Working Draft. URL: https://www.w3.org/TR/CSP3/

[CSS-CASCADE-4]
CSS Cascading and Inheritance Level 4. Elika Etemad; Tab Atkins Jr.. W3C. 14 January 2016. W3C Candidate Recommendation. URL: https://www.w3.org/TR/css-cascade-4/

[CSS-DEVICE-ADAPT]
CSS Device Adaptation Module Level 1. Rune Lillesveen; Florian Rivoal; Matt Rakow. W3C. 29 March 2016. W3C Working Draft. URL: https://www.w3.org/TR/css-device-adapt-1/

[CSS21]
Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. Bert Bos; Tantek Çelik; Ian Hickson; Håkon Wium Lie et al. W3C. 7 June 2011. W3C Recommendation. URL: https://www.w3.org/TR/CSS2/

[CSS3-BOX]
CSS basic box model. Bert Bos. W3C. 9 August 2007. W3C Working Draft. URL: https://www.w3.org/TR/css3-box/

[CSS3-DISPLAY]
CSS Display Module Level 3. Tab Atkins Jr.; Elika Etemad. W3C. 20 April 2018. W3C Working Draft. URL: https://www.w3.org/TR/css-display-3/

[CSS3-VALUES]
CSS Values and Units Module Level 3. Tab Atkins Jr.; Elika Etemad. W3C. 29 September 2016. W3C Candidate Recommendation. URL: https://www.w3.org/TR/css-values-3/

[CSSOM]
CSS Object Model (CSSOM). Simon Pieters; Glenn Adams. W3C. 17 March 2016. W3C Working Draft. URL: https://www.w3.org/TR/cssom-1/

[CSSOM-VIEW]
CSSOM View Module. Simon Pieters. W3C. 17 March 2016. W3C Working Draft. URL: https://www.w3.org/TR/cssom-view-1/

[DOM]
DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/

[DOM-PARSING]
DOM Parsing and Serialization. Travis Leithead. W3C. 17 May 2016. W3C Working Draft. URL: https://www.w3.org/TR/DOM-Parsing/

[DOMPARSING]
DOM Parsing and Serialization Standard. Ms2ger. WHATWG. Living Standard. URL: https://domparsing.spec.whatwg.org/

[ECMA-262]
ECMAScript Language Specification. Ecma International. URL: https://tc39.github.io/ecma262/

[EDITING]
HTML Editing APIs. A. Gregor. W3C. URL: https://dvcs.w3.org/hg/editing/raw-file/tip/editing.html

[FETCH]
Fetch Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://fetch.spec.whatwg.org/

[FULLSCREEN]
Fullscreen API Standard. Philip Jägenstedt. WHATWG. Living Standard. URL: https://fullscreen.spec.whatwg.org/

[GEOMETRY-1]
Geometry Interfaces Module Level 1. Simon Pieters; Dirk Schulze; Rik Cabanier. W3C. 25 November 2014. W3C Candidate Recommendation. URL: https://www.w3.org/TR/geometry-1/

[HTML]
HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[PAGE-VISIBILITY]

Page Visibility (Second Edition). Jatinder Mann; Arvind Jain. W3C. 29 October 2013. W3C Recommendation. URL: https://www.w3.org/TR/page-visibility/

[POINTER-EVENTS]

Pointer Events. Jacob Rossi; Matt Brubeck. W3C. 24 February 2015. W3C Recommendation. URL: https://www.w3.org/TR/pointerevents/

[PROMISES-GUIDE]

Writing Promise-Using Specifications. Domenic Denicola. W3C. 16 February 2016. Finding of the W3C TAG. URL: https://www.w3.org/2001/tag/doc/promises-guide

[RFC1928]

SOCKS Protocol Version 5. M. Leech; M. Ganis; Y. Lee; R. Kuris; D. Koblas; L. Jones. IETF. March 1996. Proposed Standard. URL: https://tools.ietf.org/html/rfc1928

[RFC1929]

Username/Password Authentication for SOCKS V5. M. Leech. IETF. March 1996. Proposed Standard. URL: https://tools.ietf.org/html/rfc1929

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC2397]

The "data" URL scheme. L. Masinter. IETF. August 1998. Proposed Standard. URL: https://tools.ietf.org/html/rfc2397

[RFC3514]

The Security Flag in the IPv4 Header. S. Bellovin. IETF. 1 April 2003. Informational. URL: https://tools.ietf.org/html/rfc3514

[RFC4122]

A Universally Unique IDentifier (UUID) URN Namespace. P. Leach; M. Mealling; R. Salz. IETF. July 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc4122

[RFC4632]

Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. V. Fuller; T. Li. IETF. August 2006. Best Current Practice. URL: https://tools.ietf.org/html/rfc4632

[RFC6265]

HTTP State Management Mechanism. A. Barth. IETF. April 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6265

[RFC7230]

Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7230

[RFC7231]

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7231

[RFC7232]
Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7232

[RFC7234]
Hypertext Transfer Protocol (HTTP/1.1): Caching. R. Fielding, Ed.; M. Nottingham, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7234

[RFC7235]
Hypertext Transfer Protocol (HTTP/1.1): Authentication. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7235

[UAX29]
Unicode Text Segmentation. Mark Davis; Laurențiu Iancu. Unicode Consortium. 13 June 2017. Unicode Standard Annex #29. URL: https://www.unicode.org/reports/tr29/tr29-31.html

[UAX44]
Unicode Character Database. Mark Davis; Ken Whistler.25 September 2013. URL: http://www.unicode.org/reports/tr44/

[UI-EVENTS]
UI Events. Gary Kacmarcik; Travis Leithead. W3C. 4 August 2016. W3C Working Draft. URL: https://www.w3.org/TR/uievents/

[UIEVENTS-CODE]
UI Events KeyboardEvent code Values. Gary Kacmarcik; Travis Leithead. W3C. 1 June 2017. W3C Candidate Recommendation. URL: https://www.w3.org/TR/uievents-code/

[UIEVENTS-KEY]
UI Events KeyboardEvent key Values. Gary Kacmarcik; Travis Leithead. W3C. 1 June 2017. W3C Candidate Recommendation. URL: https://www.w3.org/TR/uievents-key/

[Unicode]
The Unicode Standard. Unicode Consortium. URL: https://www.unicode.org/versions/latest/

[URI-TEMPLATE]
URI Template. J. Gregorio; R. Fielding; M. Hadley; M. Nottingham; D. Orchard. IETF. March 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6570

[URL]
URL Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://url.spec.whatwg.org/

[WEBIDL]
Web IDL. Cameron McCormack; Boris Zbarsky; Tobie Langel. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

[XML-NAMES]
Namespaces in XML 1.0 (Third Edition). Tim Bray; Dave Hollander; Andrew Layman; Richard Tobin; Henry Thompson et al. W3C. 8 December 2009. W3C Recommendation. URL: https://www.w3.org/TR/xml-names/

[XPATH]

XML Path Language (XPath) Version 1.0. James Clark; Steven DeRose. W3C. 16 November 1999. W3C Recommendation. URL: https://www.w3.org/TR/xpath/