Released API: v1.0.0 | v0.13.0 | v0.12.0 | v0.11.0 | v0.10.2 | v0.10.1 | v0.10.0 | v0.9.0

Puppeteer API v1.0.0-post

Table of Contents

Overview

Puppeteer is a Node library which provides a high-level API to control Chromium or Chrome over the DevTools Protocol.

Puppeteer API is hierarchical and mirrors browser structure. On the following diagram, faded entities are not currently represented in Puppeteer.

- `Puppeteer` communicates with browser using devtools protocol.
- `Browser` instance owns multiple pages.
- `Page` has at least one frame: main frame. There might be other frames created by iframe or frame tags.
- `Frame` has at least one execution context - default execution context - where frame's JavaScript is executed. Frame might have additional execution contexts associated with extensions.

(Diagram source: link)

Environment Variables

Puppeteer looks for certain environment variables to aid its operations. These variables can either be set in the environment or in the npm config.

- `HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY` - defines HTTP proxy settings that are used to download and run Chromium.
- `PUPPETEER_SKIP_CHROMIUM_DOWNLOAD` - do not download bundled Chromium during installation step.
- `PUPPETEER_DOWNLOAD_HOST` - overwrite host part of URL that is used to download Chromium

class: Puppeteer

Puppeteer module provides a method to launch a Chromium instance. The following is a typical example of using a Puppeteer to drive automation:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  await page.goto('https://www.google.com');
  // other actions...
  await browser.close();
});
```

puppeteer.connect(options)

- `options` <Object>
  - `browserWSEndpoint` <string> a browser websocket endpoint to connect to.
  - `ignoreHTTPSErrors` <boolean> Whether to ignore HTTPS errors during navigation. Defaults to `false`.
  - `slowMo` <number> Slows down Puppeteer operations by the specified amount of milliseconds. Useful so that you can see what is going on.
- returns: <Promise<Browser>>

This methods attaches Puppeteer to an existing Chromium instance.

puppeteer.defaultArgs()

- returns: <Array<string>> The default flags that Chromium will be launched with.

puppeteer.executablePath()

- returns: <string> A path where Puppeteer expects to find bundled Chromium. Chromium might not exist there if the download was skipped with `PUPPETEER_SKIP_CHROMIUM_DOWNLOAD`.

puppeteer.launch([options])

- `options` <Object> Set of configurable options to set on the browser. Can have the following fields:
  - `ignoreHTTPSErrors` <boolean> Whether to ignore HTTPS errors during navigation. Defaults to `false`.
  - `headless` <boolean> Whether to run browser in headless mode. Defaults to `true` unless the `devtools` option is `true`.
  - `executablePath` <string> Path to a Chromium or Chrome executable to run instead of bundled Chromium. If `executablePath` is a relative path, then it is resolved relative to current working directory.
  - `slowMo` <number> Slows down Puppeteer operations by the specified amount of milliseconds. Useful so that you can see what is going on.
  - `args` <Array<string>> Additional arguments to pass to the browser instance. List of Chromium flags can be found here.
  - `ignoreDefaultArgs` <boolean> Do not use `puppeteer.defaultArgs()`. Dangerous option; use with care. Defaults to `false`.
  - `handleSIGINT` <boolean> Close browser process on Ctrl-C. Defaults to `true`.
  - `handleSIGTERM` <boolean> Close browser process on SIGTERM. Defaults to `true`.
  - `handleSIGHUP` <boolean> Close browser process on SIGHUP. Defaults to `true`.
  - `timeout` <number> Maximum time in milliseconds to wait for the browser instance to start. Defaults to `30000` (30 seconds). Pass `0` to disable timeout.

- `dumpio` <boolean> Whether to pipe browser process stdout and stderr into `process.stdout` and `process.stderr`. Defaults to `false`.
- `userDataDir` <string> Path to a User Data Directory.
- `env` <Object> Specify environment variables that will be visible to browser. Defaults to `process.env`.
- `devtools` <boolean> Whether to auto-open DevTools panel for each tab. If this option is `true`, the `headless` option will be set `false`.

- returns: <Promise<Browser>> Promise which resolves to browser instance.

The method launches a browser instance with given arguments. The browser will be closed when the parent node.js process is closed.

> **NOTE** Puppeteer can also be used to control the Chrome browser, but it works best with the version of Chromium it is bundled with. There is no guarantee it will work with any other version. Use `executablePath` option with extreme caution. If Google Chrome (rather than Chromium) is preferred, a Chrome Canary or Dev Channel build is suggested.
>
> In puppeteer.launch([options]) above, any mention of Chromium also applies to Chrome.
>
> See `this article` for a description of the differences between Chromium and Chrome. `This article` describes some differences for Linux users.

class: Browser

- extends: `EventEmitter`

A Browser is created when Puppeteer connects to a Chromium instance, either through `puppeteer.launch` or `puppeteer.connect`.

An example of using a Browser to create a Page:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  await page.goto('https://example.com');
```

```
  await browser.close();
});
```

An example of disconnecting from and reconnecting to a Browser:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  // Store the endpoint to be able to reconnect to Chromium
  const browserWSEndpoint = browser.wsEndpoint();
  // Disconnect puppeteer from Chromium
  browser.disconnect();

  // Use the endpoint to reestablish a connection
  const browser2 = await puppeteer.connect({browserWSEndpoint});
  // Close Chromium
  await browser2.close();
});
```

event: 'disconnected'

Emitted when puppeteer gets disconnected from the Chromium instance. This might happen because one of the following:

- Chromium is closed or crashed
- `browser.disconnect` method was called

event: 'targetchanged'

- <Target>

Emitted when the url of a target changes.

event: 'targetcreated'

- <Target>

Emitted when a target is created, for example when a new page is opened by `window.open` or `browser.newPage`.

event: 'targetdestroyed'

- <Target>

Emitted when a target is destroyed, for example when a page is closed.

browser.close()

- returns: <Promise>

Closes Chromium and all of its pages (if any were opened). The browser object itself is considered disposed and cannot be used anymore.

browser.disconnect()

Disconnects Puppeteer from the browser, but leaves the Chromium process running. After calling `disconnect`, the browser object is considered disposed and cannot be used anymore.

browser.newPage()

- returns: <Promise<Page>> Promise which resolves to a new Page object.

browser.pages()

- returns: <Promise<Array<Page>>> Promise which resolves to an array of all open pages.

browser.process()

- returns: <?ChildProcess> Spawned browser process. Returns `null` if the browser instance was created with `puppeteer.connect` method.

browser.targets()

- returns: <Array<Target>> An array of all active targets.

browser.userAgent()

- returns: <Promise<string>> Promise which resolves to the browser's original user agent.

> **NOTE** Pages can override browser user agent with page.setUserAgent

browser.version()

- returns: <Promise<string>> For headless Chromium, this is similar to `HeadlessChrome/61.0.3153.0`. For non-headless, this is similar to `Chrome/61.0.3153.0`.

> **NOTE** the format of browser.version() might change with future releases of Chromium.

browser.wsEndpoint()

- returns: <string> Browser websocket url.

Browser websocket endpoint which can be used as an argument to puppeteer.connect. The format is `ws://${host}:${port}/devtools/browser/<id>`

You can find the `webSocketDebuggerUrl` from `http://${host}:${port}/json/version`. Learn more about the devtools protocol and the browser endpoint.

class: Page

- extends: `EventEmitter`

Page provides methods to interact with a single tab in Chromium. One Browser instance might have multiple Page instances.

This example creates a page, navigates it to a URL, and then saves a screenshot:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  await page.goto('https://example.com');
```

```
  await page.screenshot({path: 'screenshot.png'});
  await browser.close();
});
```

The Page class emits various events (described below) which can be handled using any of Node's native EventEmitter methods, such as `on` or `once`.

This example logs a message for a single page `load` event:

```
page.once('load', () => console.log('Page loaded!'));
```

event: 'console'

- <ConsoleMessage>

Emitted when JavaScript within the page calls one of console API methods, e.g. `console.log` or `console.dir`. Also emitted if the page throws an error or a warning.

The arguments passed into `console.log` appear as arguments on the event handler.

An example of handling `console` event:

```
page.on('console', msg => {
  for (let i = 0; i < msg.args().length; ++i)
    console.log(`${i}: ${msg.args()[i]}`);
});
page.evaluate(() => console.log('hello', 5, {foo: 'bar'}));
```

event: 'dialog'

- <Dialog>

Emitted when a JavaScript dialog appears, such as `alert`, `prompt`, `confirm` or `beforeunload`. Puppeteer can respond to the dialog via Dialog's accept or dismiss methods.

event: 'domcontentloaded'

Emitted when the JavaScript `DOMContentLoaded` event is dispatched.

event: 'error'

- <Error>

Emitted when the page crashes.

> **NOTE** `error` event has a special meaning in Node, see error events for details.

event: 'frameattached'

- <Frame>

Emitted when a frame is attached.

event: 'framedetached'

- <Frame>

Emitted when a frame is detached.

event: 'framenavigated'

- <Frame>

Emitted when a frame is navigated to a new url.

event: 'load'

Emitted when the JavaScript `load` event is dispatched.

event: 'metrics'

- <Object>

- `title` <string> The title passed to `console.timeStamp`.
- `metrics` <Object> Object containing metrics as key/value pairs. The values of metrics are of <number> type.

Emitted when the JavaScript code makes a call to `console.timeStamp`. For the list of metrics see `page.metrics`.

event: 'pageerror'

- <Error> The exception message

Emitted when an uncaught exception happens within the page.

event: 'request'

- <Request>

Emitted when a page issues a request. The request object is read-only. In order to intercept and mutate requests, see `page.setRequestInterception`.

event: 'requestfailed'

- <Request>

Emitted when a request fails, for example by timing out.

event: 'requestfinished'

- <Request>

Emitted when a request finishes successfully.

event: 'response'

- <Response>

Emitted when a response is received.

page.$(selector)

- `selector` <string> A selector to query page for
- returns: <Promise<?ElementHandle>>

The method runs `document.querySelector` within the page. If no element matches the selector, the return value resolve to `null`.

Shortcut for page.mainFrame().$(selector).

page.$$(selector)

- `selector` <string> A selector to query page for
- returns: <Promise<Array<ElementHandle>>>

The method runs `document.querySelectorAll` within the page. If no elements match the selector, the return value resolve to `[]`.

Shortcut for page.mainFrame().$$(selector).

page.$$eval(selector, pageFunction[, ...args])

- `selector` <string> A selector to query frame for
- `pageFunction` <function> Function to be evaluated in browser context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<Serializable>> Promise which resolves to the return value of `pageFunction`

This method runs `document.querySelectorAll` within the page and passes it as the first argument to `pageFunction`.

If `pageFunction` returns a Promise, then `page.$$eval` would wait for the promise to resolve and return its value.

Examples:

```
const divsCounts = await page.$$eval('div', divs => divs.length);
```

page.$eval(selector, pageFunction[, ...args])

- `selector` <string> A selector to query page for
- `pageFunction` <function> Function to be evaluated in browser context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<Serializable>> Promise which resolves to the return value of `pageFunction`

This method runs `document.querySelector` within the page and passes it as the first argument to `pageFunction`. If there's no element matching `selector`, the method throws an error.

If `pageFunction` returns a Promise, then `page.$eval` would wait for the promise to resolve and return its value.

Examples:

```
const searchValue = await page.$eval('#search', el => el.value);
const preloadHref = await page.$eval('link[rel=preload]', el => el.href);
const html = await page.$eval('.main-container', e => e.outerHTML);
```

Shortcut for page.mainFrame().$eval(selector, pageFunction).

page.$x(expression)

- `expression` <string> Expression to evaluate.
- returns: <Promise<Array<ElementHandle>>>

The method evaluates the XPath expression.

Shortcut for page.mainFrame().$x(expression)

page.addScriptTag(options)

- `options` <Object>
  - `url` <string> Url of a script to be added.

    - `path` <string> Path to the JavaScript file to be injected into frame. If `path` is a relative path, then it is resolved relative to current working directory.
    - `content` <string> Raw JavaScript content to be injected into frame.
- returns: <Promise<ElementHandle>> which resolves to the added tag when the script's onload fires or when the script content was injected into frame.

Adds a `<script>` tag into the page with the desired url or content.

Shortcut for page.mainFrame().addScriptTag(options).

page.addStyleTag(options)

- `options` <Object>
    - `url` <string> Url of the `<link>` tag.
    - `path` <string> Path to the CSS file to be injected into frame. If `path` is a relative path, then it is resolved relative to current working directory.
    - `content` <string> Raw CSS content to be injected into frame.
- returns: <Promise<ElementHandle>> which resolves to the added tag when the stylesheet's onload fires or when the CSS content was injected into frame.

Adds a `<link rel="stylesheet">` tag into the page with the desired url or a `<style type="text/css">` tag with the content.

Shortcut for page.mainFrame().addStyleTag(options).

page.authenticate(credentials)

- `credentials` <?Object>
    - `username` <string>
    - `password` <string>
- returns: <Promise>

Provide credentials for http authentication.

To disable authentication, pass `null`.

page.bringToFront()

- returns: <Promise>

Brings page to front (activates tab).

page.click(selector[, options])

- `selector` <string> A selector to search for element to click. If there are multiple elements satisfying the selector, the first will be clicked.
- `options` <Object>
  - `button` <string> `left`, `right`, or `middle`, defaults to `left`.
  - `clickCount` <number> defaults to 1. See UIEvent.detail.
  - `delay` <number> Time to wait between `mousedown` and `mouseup` in milliseconds. Defaults to 0.
- returns: <Promise> Promise which resolves when the element matching `selector` is successfully clicked. The Promise will be rejected if there is no element matching `selector`.

This method fetches an element with `selector`, scrolls it into view if needed, and then uses page.mouse to click in the center of the element. If there's no element matching `selector`, the method throws an error.

Bare in mind that if `click()` triggers a navigation event and there's a separate `page.waitForNavigation()` promise to be resolved, you may end up with a race condition that yields unexpected results. The correct pattern for click and wait for navigation is the following:

```
const [response] = await Promise.all([
  page.waitForNavigation(waitOptions),
  page.click(selector, clickOptions),
]);
```

Shortcut for page.mainFrame().click(selector[, options]).

page.close()

- returns: <Promise>

page.content()

- returns: <Promise<String>>

Gets the full HTML contents of the page, including the doctype.

page.cookies(...urls)

- `...urls` <...string>
- returns: <Promise<Array<Object>>>
  - `name` <string>
  - `value` <string>
  - `domain` <string>
  - `path` <string>
  - `expires` <number> Unix time in seconds.
  - `httpOnly` <boolean>
  - `secure` <boolean>
  - `session` <boolean>
  - `sameSite` <string> `"Strict"` or `"Lax"`.

If no URLs are specified, this method returns cookies for the current page URL. If URLs are specified, only cookies for those URLs are returned.

page.coverage

- returns: <Coverage>

page.deleteCookie(...cookies)

- `...cookies` <...Object>
  - `name` <string> **required**
  - `url` <string>
  - `domain` <string>
  - `path` <string>
  - `secure` <boolean>
- returns: <Promise>

page.emulate(options)

- `options` <Object>
  - `viewport` <Object>
    - `width` <number> page width in pixels.
    - `height` <number> page height in pixels.
    - `deviceScaleFactor` <number> Specify device scale factor (can be thought of as dpr). Defaults to `1`.
    - `isMobile` <boolean> Whether the `meta viewport` tag is taken into account. Defaults to `false`.
    - `hasTouch` <boolean> Specifies if viewport supports touch events. Defaults to `false`
    - `isLandscape` <boolean> Specifies if viewport is in landscape mode. Defaults to `false`.
  - `userAgent` <string>
- returns: <Promise>

Emulates given device metrics and user agent. This method is a shortcut for calling two methods:

- page.setUserAgent(userAgent)
- page.setViewport(viewport)

To aid emulation, puppeteer provides a list of device descriptors which can be obtained via the `require('puppeteer/DeviceDescriptors')` command. Below is an example of emulating an iPhone 6 in puppeteer:

```
const puppeteer = require('puppeteer');
const devices = require('puppeteer/DeviceDescriptors');
```

```
const iPhone = devices['iPhone 6'];

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  await page.emulate(iPhone);
  await page.goto('https://www.google.com');
  // other actions...
  await browser.close();
});
```

List of all available devices is available in the source code: DeviceDescriptors.js.

page.emulateMedia(mediaType)

- `mediaType` <?string> Changes the CSS media type of the page. The only allowed values are `'screen'`, `'print'` and `null`. Passing `null` disables media emulation.
- returns: <Promise>

page.evaluate(pageFunction, ...args)

- `pageFunction` <function|string> Function to be evaluated in the page context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<Serializable>> Resolves to the return value of `pageFunction`

If the function, passed to the `page.evaluate`, returns a Promise, then `page.evaluate` would wait for the promise to resolve and return its value.

If the function passed into `page.evaluate` returns a non-Serializable value, then `page.evaluate` resolves to `undefined`. Passing arguments to `pageFunction`.

```
const result = await page.evaluate(x => {
  return Promise.resolve(8 * x);
}, 7);
console.log(result); // prints "56"
```

A string can also be passed in instead of a function.

```
console.log(await page.evaluate('1 + 2')); // prints "3"
const x = 10;
console.log(await page.evaluate(`1 + ${x}`)); // prints "11"
```

ElementHandle instances can be passed as arguments to the `page.evaluate`:

```
const bodyHandle = await page.$('body');
const html = await page.evaluate(body => body.innerHTML, bodyHandle);
await bodyHandle.dispose();
```

Shortcut for page.mainFrame().evaluate(pageFunction, ...args).

page.evaluateHandle(pageFunction, ...args)

- `pageFunction` <function|string> Function to be evaluated in the page context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<JSHandle>> Resolves to the return value of `pageFunction`

If the function, passed to the `page.evaluateHandle`, returns a Promise, then `page.evaluateHandle` would wait for the promise to resolve and return its value.

```
const aWindowHandle = await page.evaluateHandle(() => Promise.resolve(window));
aWindowHandle; // Handle for the window object.
```

A string can also be passed in instead of a function.

```
const aHandle = await page.evaluateHandle('document'); // Handle for the 'document'.
```

JSHandle instances can be passed as arguments to the `page.evaluateHandle`:

```
const aHandle = await page.evaluateHandle(() => document.body);
const resultHandle = await page.evaluateHandle(body => body.innerHTML, aHandle);
console.log(await resultHandle.jsonValue());
await resultHandle.dispose();
```

Shortcut for page.mainFrame().executionContext().evaluateHandle(pageFunction, ...args).

page.evaluateOnNewDocument(pageFunction, ...args)

- `pageFunction` <function|string> Function to be evaluated in browser context
- `...args` <...Serializable> Arguments to pass to `pageFunction`
- returns: <Promise>

Adds a function which would be invoked in one of the following scenarios:

- whenever the page is navigated
- whenever the child frame is attached or navigated. In this case, the function is invoked in the context of the newly attached frame

The function is invoked after the document was created but before any of its scripts were run. This is useful to amend JavaScript environment, e.g. to seed `Math.random`.

An example of overriding the navigator.languages property before the page loads:

```
// preload.js

// overwrite the `languages` property to use a custom getter
Object.defineProperty(navigator, "languages", {
  get: function() {
    return ["en-US", "en", "bn"];
  };
});

// In your puppeteer script, assuming the preload.js file is in same folder of our script
```

```
const preloadFile = fs.readFileSync('./preload.js', 'utf8');
await page.evaluateOnNewDocument(preloadFile);
```

page.exposeFunction(name, puppeteerFunction)

- `name` <string> Name of the function on the window object
- `puppeteerFunction` <function> Callback function which will be called in Puppeteer's context.
- returns: <Promise>

The method adds a function called `name` on the page's `window` object. When called, the function executes `puppeteerFunction` in node.js and returns a Promise which resolves to the return value of `puppeteerFunction`.

If the `puppeteerFunction` returns a Promise, it will be awaited.

> **NOTE** Functions installed via `page.exposeFunction` survive navigations.

An example of adding an `md5` function into the page:

```
const puppeteer = require('puppeteer');
const crypto = require('crypto');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  page.on('console', msg => console.log(msg.text()));
  await page.exposeFunction('md5', text =>
    crypto.createHash('md5').update(text).digest('hex')
  );
  await page.evaluate(async () => {
    // use window.md5 to compute hashes
    const myString = 'PUPPETEER';
    const myHash = await window.md5(myString);
    console.log(`md5 of ${myString} is ${myHash}`);
  });
  await browser.close();
});
```

An example of adding a `window.readfile` function into the page:

```
const puppeteer = require('puppeteer');
const fs = require('fs');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  page.on('console', msg => console.log(msg.text()));
  await page.exposeFunction('readfile', async filePath => {
    return new Promise((resolve, reject) => {
      fs.readFile(filePath, 'utf8', (err, text) => {
        if (err)
          reject(err);
        else
          resolve(text);
      });
    });
  });
  await page.evaluate(async () => {
    // use window.readfile to read contents of a file
    const content = await window.readfile('/etc/hosts');
    console.log(content);
  });
  await browser.close();
});
```

page.focus(selector)

- `selector` <string> A selector of an element to focus. If there are multiple elements satisfying the selector, the first will be focused.
- returns: <Promise> Promise which resolves when the element matching `selector` is successfully focused. The promise will be rejected if there is no element matching `selector`.

This method fetches an element with `selector` and focuses it. If there's no element matching `selector`, the method throws an error.

Shortcut for page.mainFrame().focus(selector).

page.frames()

- returns: <Array<Frame>> An array of all frames attached to the page.

page.goBack(options)

- `options` <Object> Navigation parameters which might have the following properties:
  - `timeout` <number> Maximum navigation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the page.setDefaultNavigationTimeout(timeout) method.
  - `waitUntil` <string|Array<string>> When to consider navigation succeeded, defaults to `load`. Given an array of event strings, navigation is considered to be successful after all events have been fired. Events can be either:
    - `load` - consider navigation to be finished when the `load` event is fired.
    - `domcontentloaded` - consider navigation to be finished when the `DOMContentLoaded` event is fired.
    - `networkidle0` - consider navigation to be finished when there are no more than 0 network connections for at least `500` ms.
    - `networkidle2` - consider navigation to be finished when there are no more than 2 network connections for at least `500` ms.
- returns: <Promise<?Response>> Promise which resolves to the main resource response. In case of multiple redirects, the navigation will resolve with the response of the last redirect. If can not go back, resolves to `null`.

Navigate to the previous page in history.

page.goForward(options)

- `options` <Object> Navigation parameters which might have the following properties:
  - `timeout` <number> Maximum navigation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the page.setDefaultNavigationTimeout(timeout) method.
  - `waitUntil` <string|Array<string>> When to consider navigation succeeded, defaults to `load`. Given an array of event strings, navigation is considered to be successful after all events have been fired. Events can be either:
    - `load` - consider navigation to be finished when the `load` event is fired.

    - `domcontentloaded` - consider navigation to be finished when the `DOMContentLoaded` event is fired.
    - `networkidle0` - consider navigation to be finished when there are no more than 0 network connections for at least `500` ms.
    - `networkidle2` - consider navigation to be finished when there are no more than 2 network connections for at least `500` ms.
- returns: <Promise<?Response>> Promise which resolves to the main resource response. In case of multiple redirects, the navigation will resolve with the response of the last redirect. If can not go back, resolves to `null`.

Navigate to the next page in history.

page.goto(url, options)

- `url` <string> URL to navigate page to. The url should include scheme, e.g. `https://`.
- `options` <Object> Navigation parameters which might have the following properties:
  - `timeout` <number> Maximum navigation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the page.setDefaultNavigationTimeout(timeout) method.
  - `waitUntil` <string|Array<string>> When to consider navigation succeeded, defaults to `load`. Given an array of event strings, navigation is considered to be successful after all events have been fired. Events can be either:
    - `load` - consider navigation to be finished when the `load` event is fired.
    - `domcontentloaded` - consider navigation to be finished when the `DOMContentLoaded` event is fired.
    - `networkidle0` - consider navigation to be finished when there are no more than 0 network connections for at least `500` ms.
    - `networkidle2` - consider navigation to be finished when there are no more than 2 network connections for at least `500` ms.
- returns: <Promise<?Response>> Promise which resolves to the main resource response. In case of multiple redirects, the navigation will resolve with the response of the last redirect.

The `page.goto` will throw an error if:

- there's an SSL error (e.g. in case of self-signed certificates).
- target URL is invalid.

- the `timeout` is exceeded during navigation.
- the main resource failed to load.

> **NOTE** `page.goto` either throw or return a main resource response. The only exception is navigation to `about:blank`, which would succeed and return `null`.

> **NOTE** Headless mode doesn't support navigating to a PDF document. See the upstream issue.

page.hover(selector)

- `selector` <string> A selector to search for element to hover. If there are multiple elements satisfying the selector, the first will be hovered.
- returns: <Promise> Promise which resolves when the element matching `selector` is successfully hovered. Promise gets rejected if there's no element matching `selector`.

This method fetches an element with `selector`, scrolls it into view if needed, and then uses page.mouse to hover over the center of the element. If there's no element matching `selector`, the method throws an error.

Shortcut for page.mainFrame().hover(selector).

page.keyboard

- returns: <Keyboard>

page.mainFrame()

- returns: <Frame> returns page's main frame.

Page is guaranteed to have a main frame which persists during navigations.

page.metrics()

- returns: <Promise<Object>> Object containing metrics as key/value pairs.
  - `Timestamp` <number> The timestamp when the metrics sample was taken.
  - `Documents` <number> Number of documents in the page.

- `Frames` <number> Number of frames in the page.
- `JSEventListeners` <number> Number of events in the page.
- `Nodes` <number> Number of DOM nodes in the page.
- `LayoutCount` <number> Total number of full or partial page layout.
- `RecalcStyleCount` <number> Total number of page style recalculations.
- `LayoutDuration` <number> Combined durations of all page layouts.
- `RecalcStyleDuration` <number> Combined duration of all page style recalculations.
- `ScriptDuration` <number> Combined duration of JavaScript execution.
- `TaskDuration` <number> Combined duration of all tasks performed by the browser.
- `JSHeapUsedSize` <number> Used JavaScript heap size.
- `JSHeapTotalSize` <number> Total JavaScript heap size.

> **NOTE** All timestamps are in monotonic time: monotonically increasing time in seconds since an arbitrary point in the past.

page.mouse

- returns: <Mouse>

page.pdf(options)

- `options` <Object> Options object which might have the following properties:
  - `path` <string> The file path to save the PDF to. If `path` is a relative path, then it is resolved relative to current working directory. If no path is provided, the PDF won't be saved to the disk.
  - `scale` <number> Scale of the webpage rendering. Defaults to `1` .
  - `displayHeaderFooter` <boolean> Display header and footer. Defaults to `false` .
  - `headerTemplate` <string> HTML template for the print header. Should be valid HTML markup with following classes used to inject printing values into them:
    - `date` formatted print date
    - `title` document title

        - `url` document location
        - `pageNumber` current page number
        - `totalPages` total pages in the document
    - `footerTemplate` <string> HTML template for the print footer. Should use the same format as the `headerTemplate`.
    - `printBackground` <boolean> Print background graphics. Defaults to `false`.
    - `landscape` <boolean> Paper orientation. Defaults to `false`.
    - `pageRanges` <string> Paper ranges to print, e.g., '1-5, 8, 11-13'. Defaults to the empty string, which means print all pages.
    - `format` <string> Paper format. If set, takes priority over `width` or `height` options. Defaults to 'Letter'.
    - `width` <string> Paper width, accepts values labeled with units.
    - `height` <string> Paper height, accepts values labeled with units.
    - `margin` <Object> Paper margins, defaults to none.
        - `top` <string> Top margin, accepts values labeled with units.
        - `right` <string> Right margin, accepts values labeled with units.
        - `bottom` <string> Bottom margin, accepts values labeled with units.
        - `left` <string> Left margin, accepts values labeled with units.
- returns: <Promise<Buffer>> Promise which resolves with PDF buffer.

> **NOTE** Generating a pdf is currently only supported in Chrome headless.

`page.pdf()` generates a pdf of the page with `print` css media. To generate a pdf with `screen` media, call page.emulateMedia('screen') before calling `page.pdf()`:

```
// Generates a PDF with 'screen' media type.
await page.emulateMedia('screen');
await page.pdf({path: 'page.pdf'});
```

The `width`, `height`, and `margin` options accept values labeled with units. Unlabeled values are treated as pixels.

A few examples:

- `page.pdf({width: 100})` - prints with width set to 100 pixels
- `page.pdf({width: '100px'})` - prints with width set to 100 pixels
- `page.pdf({width: '10cm'})` - prints with width set to 10 centimeters.

All possible units are:

- `px` - pixel
- `in` - inch
- `cm` - centimeter
- `mm` - millimeter

The `format` options are:

- `Letter` : 8.5in x 11in
- `Legal` : 8.5in x 14in
- `Tabloid` : 11in x 17in
- `Ledger` : 17in x 11in
- `A0` : 33.1in x 46.8in
- `A1` : 23.4in x 33.1in
- `A2` : 16.5in x 23.4in
- `A3` : 11.7in x 16.5in
- `A4` : 8.27in x 11.7in
- `A5` : 5.83in x 8.27in
- `A6` : 4.13in x 5.83in

page.queryObjects(prototypeHandle)

- `prototypeHandle` <JSHandle> A handle to the object prototype.
- returns: <Promise<JSHandle>> Promise which resolves to a handle to an array of objects with this prototype.

The method iterates JavaScript heap and finds all the objects with the given prototype.

```
// Create a Map object
await page.evaluate(() => window.map = new Map());
// Get a handle to the Map object prototype
const mapPrototype = await page.evaluateHandle(() => Map.prototype);
// Query all map instances into an array
const mapInstances = await page.queryObjects(mapPrototype);
// Count amount of map objects in heap
const count = await page.evaluate(maps => maps.length, mapInstances);
await mapInstances.dispose();
await mapPrototype.dispose();
```

Shortcut for page.mainFrame().executionContext().queryObjects(prototypeHandle).

page.reload(options)

- `options` <Object> Navigation parameters which might have the following properties:
  - `timeout` <number> Maximum navigation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the page.setDefaultNavigationTimeout(timeout) method.
  - `waitUntil` <string|Array<string>> When to consider navigation succeeded, defaults to `load`. Given an array of event strings, navigation is considered to be successful after all events have been fired. Events can be either:
    - `load` - consider navigation to be finished when the `load` event is fired.
    - `domcontentloaded` - consider navigation to be finished when the `DOMContentLoaded` event is fired.
    - `networkidle0` - consider navigation to be finished when there are no more than 0 network connections for at least `500` ms.
    - `networkidle2` - consider navigation to be finished when there are no more than 2 network connections for at least `500` ms.
- returns: <Promise<Response>> Promise which resolves to the main resource response. In case of multiple redirects, the navigation will resolve with the response of the last redirect.

page.screenshot([options])

- `options` <Object> Options object which might have the following properties:

- `path` <string> The file path to save the image to. The screenshot type will be inferred from file extension. If `path` is a relative path, then it is resolved relative to current working directory. If no path is provided, the image won't be saved to the disk.
- `type` <string> Specify screenshot type, can be either `jpeg` or `png`. Defaults to 'png'.
- `quality` <number> The quality of the image, between 0-100. Not applicable to `png` images.
- `fullPage` <boolean> When true, takes a screenshot of the full scrollable page. Defaults to `false`.
- `clip` <Object> An object which specifies clipping region of the page. Should have the following fields:
  - `x` <number> x-coordinate of top-left corner of clip area
  - `y` <number> y-coordinate of top-left corner of clip area
  - `width` <number> width of clipping area
  - `height` <number> height of clipping area
- `omitBackground` <boolean> Hides default white background and allows capturing screenshots with transparency. Defaults to `false`.

- returns: <Promise<Buffer>> Promise which resolves to buffer with captured screenshot

page.select(selector, ...values)

- `selector` <string> A selector to query page for
- `...values` <...string> Values of options to select. If the `<select>` has the `multiple` attribute, all values are considered, otherwise only the first one is taken into account.
- returns: <Promise<Array<string>>> Returns an array of option values that have been successfully selected.

Triggers a `change` and `input` event once all the provided options have been selected. If there's no `<select>` element matching `selector`, the method throws an error.

```
page.select('select#colors', 'blue'); // single selection
page.select('select#colors', 'red', 'green', 'blue'); // multiple selections
```

Shortcut for page.mainFrame().select()

page.setContent(html)

- `html` <string> HTML markup to assign to the page.
- returns: <Promise>

page.setCookie(...cookies)

- `...cookies` <...Object>
  - `name` <string> **required**
  - `value` <string> **required**
  - `url` <string>
  - `domain` <string>
  - `path` <string>
  - `expires` <number> Unix time in seconds.
  - `httpOnly` <boolean>
  - `secure` <boolean>
  - `sameSite` <string> `"Strict"` or `"Lax"`.
- returns: <Promise>

page.setDefaultNavigationTimeout(timeout)

- `timeout` <number> Maximum navigation time in milliseconds

This setting will change the default maximum navigation time of 30 seconds for the following methods:

- page.goto(url, options)
- page.goBack(options)
- page.goForward(options)
- page.reload(options)
- page.waitForNavigation(options)

page.setExtraHTTPHeaders(headers)

- `headers` <Object> An object containing additional http headers to be sent with every request. All header values must be strings.
- returns: <Promise>

The extra HTTP headers will be sent with every request the page initiates.

> **NOTE** page.setExtraHTTPHeaders does not guarantee the order of headers in the outgoing requests.

page.setJavaScriptEnabled(enabled)

- `enabled` <boolean> Whether or not to enable JavaScript on the page.
- returns: <Promise>

> **NOTE** changing this value won't affect scripts that have already been run. It will take full effect on the next navigation.

page.setOfflineMode(enabled)

- `enabled` <boolean> When `true`, enables offline mode for the page.
- returns: <Promise>

page.setRequestInterception(value)

- `value` <boolean> Whether to enable request interception.
- returns: <Promise>

Activating request interception enables `request.abort`, `request.continue` and `request.respond` methods.

An example of a naïve request interceptor that aborts all image requests:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
```

```
  await page.setRequestInterception(true);
  page.on('request', interceptedRequest => {
    if (interceptedRequest.url().endsWith('.png') || interceptedRequest.url().endsWith('.jpg'))
      interceptedRequest.abort();
    else
      interceptedRequest.continue();
  });
  await page.goto('https://example.com');
  await browser.close();
});
```

> **NOTE** Enabling request interception disables page caching.

page.setUserAgent(userAgent)

- `userAgent` <string> Specific user agent to use in this page
- returns: <Promise> Promise which resolves when the user agent is set.

page.setViewport(viewport)

- `viewport` <Object>
  - `width` <number> page width in pixels.
  - `height` <number> page height in pixels.
  - `deviceScaleFactor` <number> Specify device scale factor (can be thought of as dpr). Defaults to `1`.
  - `isMobile` <boolean> Whether the `meta viewport` tag is taken into account. Defaults to `false`.
  - `hasTouch` <boolean> Specifies if viewport supports touch events. Defaults to `false`
  - `isLandscape` <boolean> Specifies if viewport is in landscape mode. Defaults to `false`.
- returns: <Promise>

> **NOTE** in certain cases, setting viewport will reload the page in order to set the `isMobile` or `hasTouch` properties.

In the case of multiple pages in a single browser, each page can have its own viewport size.

page.tap(selector)

- `selector` <string> A selector to search for element to tap. If there are multiple elements satisfying the selector, the first will be tapped.
- returns: <Promise>

This method fetches an element with `selector`, scrolls it into view if needed, and then uses page.touchscreen to tap in the center of the element. If there's no element matching `selector`, the method throws an error.

Shortcut for page.mainFrame().tap(selector).

page.target()

- returns: <Target> a target this page was created from.

page.title()

- returns: <Promise<string>> Returns page's title.

Shortcut for page.mainFrame().title().

page.touchscreen

- returns: <Touchscreen>

page.tracing

- returns: <Tracing>

page.type(selector, text[, options])

- `selector` <string> A selector of an element to type into. If there are multiple elements satisfying the selector, the first will be used.
- `text` <string> A text to type into a focused element.
- `options` <Object>
  - `delay` <number> Time to wait between key presses in milliseconds. Defaults to 0.

- returns: <Promise>

Sends a `keydown`, `keypress`/`input`, and `keyup` event for each character in the text.

To press a special key, like `Control` or `ArrowDown`, use `keyboard.press`.

```
page.type('#mytextarea', 'Hello'); // Types instantly
page.type('#mytextarea', 'World', {delay: 100}); // Types slower, like a user
```

Shortcut for page.mainFrame().type(selector, text[, options]).

page.url()

- returns: <string>

This is a shortcut for page.mainFrame().url()

page.viewport()

- returns: <Object>
  - `width` <number> page width in pixels.
  - `height` <number> page height in pixels.
  - `deviceScaleFactor` <number> Specify device scale factor (can be though of as dpr). Defaults to `1`.
  - `isMobile` <boolean> Whether the `meta viewport` tag is taken into account. Defaults to `false`.
  - `hasTouch` <boolean> Specifies if viewport supports touch events. Defaults to `false`
  - `isLandscape` <boolean> Specifies if viewport is in landscape mode. Defaults to `false`.

page.waitFor(selectorOrFunctionOrTimeout[, options[, ...args]])

- `selectorOrFunctionOrTimeout` <string|number|function> A selector, predicate or timeout to wait for
- `options` <Object> Optional waiting parameters
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`

- returns: <Promise<JSHandle>> Promise which resolves to a JSHandle of the success value

This method behaves differently with respect to the type of the first parameter:

- if `selectorOrFunctionOrTimeout` is a `string`, then the first argument is treated as a selector or xpath, depending on whether or not it starts with '//', and the method is a shortcut for page.waitForSelector or page.waitForXPath
- if `selectorOrFunctionOrTimeout` is a `function`, then the first argument is treated as a predicate to wait for and the method is a shortcut for page.waitForFunction().
- if `selectorOrFunctionOrTimeout` is a `number`, then the first argument is treated as a timeout in milliseconds and the method returns a promise which resolves after the timeout
- otherwise, an exception is thrown

Shortcut for page.mainFrame().waitFor(selectorOrFunctionOrTimeout[, options[, ...args]]).

page.waitForFunction(pageFunction[, options[, ...args]])

- `pageFunction` <function|string> Function to be evaluated in browser context
- `options` <Object> Optional waiting parameters
  - `polling` <string|number> An interval at which the `pageFunction` is executed, defaults to `raf`. If `polling` is a number, then it is treated as an interval in milliseconds at which the function would be executed. If `polling` is a string, then it can be one of the following values:
    - `raf` - to constantly execute `pageFunction` in `requestAnimationFrame` callback. This is the tightest polling mode which is suitable to observe styling changes.
    - `mutation` - to execute `pageFunction` on every DOM mutation.
  - `timeout` <number> maximum time to wait for in milliseconds. Defaults to `30000` (30 seconds).
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<JSHandle>> Promise which resolves when the `pageFunction` returns a truthy value. It resolves to a JSHandle of the truthy value.

The `waitForFunction` can be used to observe viewport size change:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  const watchDog = page.waitForFunction('window.innerWidth < 100');
  page.setViewport({width: 50, height: 50});
  await watchDog;
  await browser.close();
});
```

Shortcut for page.mainFrame().waitForFunction(pageFunction[, options[, ...args]]).

page.waitForNavigation(options)

- `options` <Object> Navigation parameters which might have the following properties:
  - `timeout` <number> Maximum navigation time in milliseconds, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the page.setDefaultNavigationTimeout(timeout) method.
  - `waitUntil` <string|Array<string>> When to consider navigation succeeded, defaults to `load`. Given an array of event strings, navigation is considered to be successful after all events have been fired. Events can be either:
    - `load` - consider navigation to be finished when the `load` event is fired.
    - `domcontentloaded` - consider navigation to be finished when the `DOMContentLoaded` event is fired.
    - `networkidle0` - consider navigation to be finished when there are no more than 0 network connections for at least `500` ms.
    - `networkidle2` - consider navigation to be finished when there are no more than 2 network connections for at least `500` ms.
- returns: <Promise<Response>> Promise which resolves to the main resource response. In case of multiple redirects, the navigation will resolve with the response of the last redirect.

page.waitForSelector(selector[, options])

- `selector` <string> A selector of an element to wait for,
- `options` <Object> Optional waiting parameters

- `visible` <boolean> wait for element to be present in DOM and to be visible, i.e. to not have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.
- `hidden` <boolean> wait for element to not be found in the DOM or to be hidden, i.e. have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.
- `timeout` <number> maximum time to wait for in milliseconds. Defaults to `30000` (30 seconds).

- returns: <Promise<ElementHandle>> Promise which resolves when element specified by selector string is added to DOM.

Wait for the `selector` to appear in page. If at the moment of calling the method the `selector` already exists, the method will return immediately. If the selector doesn't appear after the `timeout` milliseconds of waiting, the function will throw.

This method works across navigations:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  let currentURL;
  page
    .waitForSelector('img')
    .then(() => console.log('First URL with image: ' + currentURL));
  for (currentURL of ['https://example.com', 'https://google.com', 'https://bbc.com'])
    await page.goto(currentURL);
  await browser.close();
});
```

Shortcut for page.mainFrame().waitForSelector(selector[, options]).

page.waitForXPath(xpath[, options])

- `xpath` <string> A xpath of an element to wait for,
- `options` <Object> Optional waiting parameters
  - `visible` <boolean> wait for element to be present in DOM and to be visible, i.e. to not have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.

- `hidden` <boolean> wait for element to not be found in the DOM or to be hidden, i.e. have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.
- `timeout` <number> maximum time to wait for in milliseconds. Defaults to `30000` (30 seconds).

- returns: <Promise<ElementHandle>> Promise which resolves when element specified by xpath string is added to DOM.

Wait for the `xpath` to appear in page. If at the moment of calling the method the `xpath` already exists, the method will return immediately. If the xpath doesn't appear after the `timeout` milliseconds of waiting, the function will throw.

This method works across navigations:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  let currentURL;
  page
    .waitForXPath('//img')
    .then(() => console.log('First URL with image: ' + currentURL));
  for (currentURL of ['https://example.com', 'https://google.com', 'https://bbc.com'])
    await page.goto(currentURL);
  await browser.close();
});
```

Shortcut for page.mainFrame().waitForXPath(xpath[, options]).

class: Keyboard

Keyboard provides an api for managing a virtual keyboard. The high level api is `keyboard.type`, which takes raw characters and generates proper keydown, keypress/input, and keyup events on your page.

For finer control, you can use `keyboard.down`, `keyboard.up`, and `keyboard.sendCharacter` to manually fire events as if they were generated from a real keyboard.

An example of holding down `Shift` in order to select and delete some text:

```
await page.keyboard.type('Hello World!');
await page.keyboard.press('ArrowLeft');

await page.keyboard.down('Shift');
for (let i = 0; i < ' World'.length; i++)
  await page.keyboard.press('ArrowLeft');
await page.keyboard.up('Shift');

await page.keyboard.press('Backspace');
// Result text will end up saying 'Hello!'
```

An example of pressing `A`

```
await page.keyboard.down('Shift');
await page.keyboard.press('KeyA');
await page.keyboard.up('Shift');
```

> **NOTE** On MacOS, keyboard shortcuts like `⌘ A` -> Select All do not work. See #1313

keyboard.down(key[, options])

- `key` <string> Name of key to press, such as `ArrowLeft`. See USKeyboardLayout for a list of all key names.
- `options` <Object>
  - `text` <string> If specified, generates an input event with this text.
- returns: <Promise>

Dispatches a `keydown` event.

If `key` is a single character and no modifier keys besides `Shift` are being held down, a `keypress` / `input` event will also generated. The `text` option can be specified to force an input event to be generated.

If `key` is a modifier key, `Shift`, `Meta`, `Control`, or `Alt`, subsequent key presses will be sent with that modifier active. To release the modifier key, use `keyboard.up`.

After the key is pressed once, subsequent calls to `keyboard.down` will have repeat set to true. To release the key, use `keyboard.up`.

> **NOTE** Modifier keys DO influence `keyboard.down`. Holding down `Shift` will type the text in upper case.

keyboard.press(key[, options])

- `key` <string> Name of key to press, such as `ArrowLeft`. See USKeyboardLayout for a list of all key names.
- `options` <Object>
  - `text` <string> If specified, generates an input event with this text.
  - `delay` <number> Time to wait between `keydown` and `keyup` in milliseconds. Defaults to 0.
- returns: <Promise>

If `key` is a single character and no modifier keys besides `Shift` are being held down, a `keypress` / `input` event will also generated. The `text` option can be specified to force an input event to be generated.

> **NOTE** Modifier keys DO effect `elementHandle.press`. Holding down `Shift` will type the text in upper case.

Shortcut for `keyboard.down` and `keyboard.up`.

keyboard.sendCharacter(char)

- `char` <string> Character to send into the page.
- returns: <Promise>

Dispatches a `keypress` and `input` event. This does not send a `keydown` or `keyup` event.

```
page.keyboard.sendCharacter('嗨');
```

> **NOTE** Modifier keys DO NOT effect `keyboard.sendCharacter`. Holding down `Shift` will not type the text in upper case.

keyboard.type(text, options)

- `text` <string> A text to type into a focused element.
- `options` <Object>
  - `delay` <number> Time to wait between key presses in milliseconds. Defaults to 0.
- returns: <Promise>

Sends a `keydown`, `keypress`/`input`, and `keyup` event for each character in the text.

To press a special key, like `Control` or `ArrowDown`, use `keyboard.press`.

```
page.keyboard.type('Hello'); // Types instantly
page.keyboard.type('World', {delay: 100}); // Types slower, like a user
```

> **NOTE** Modifier keys DO NOT effect `keyboard.type`. Holding down `Shift` will not type the text in upper case.

keyboard.up(key)

- `key` <string> Name of key to release, such as `ArrowLeft`. See USKeyboardLayout for a list of all key names.
- returns: <Promise>

Dispatches a `keyup` event.

class: Mouse

mouse.click(x, y, [options])

- `x` <number>
- `y` <number>
- `options` <Object>
  - `button` <string> `left`, `right`, or `middle`, defaults to `left`.
  - `clickCount` <number> defaults to 1. See UIEvent.detail.
  - `delay` <number> Time to wait between `mousedown` and `mouseup` in milliseconds. Defaults to 0.
- returns: <Promise>

Shortcut for `mouse.move`, `mouse.down` and `mouse.up`.

mouse.down([options])

- `options` <Object>
  - `button` <string> `left`, `right`, or `middle`, defaults to `left`.
  - `clickCount` <number> defaults to 1. See UIEvent.detail.
- returns: <Promise>

Dispatches a `mousedown` event.

mouse.move(x, y, [options])

- `x` <number>
- `y` <number>
- `options` <Object>
  - `steps` <number> defaults to 1. Sends intermediate `mousemove` events.
- returns: <Promise>

Dispatches a `mousemove` event.

mouse.up([options])

- `options` <Object>
  - `button` <string> `left`, `right`, or `middle`, defaults to `left`.
  - `clickCount` <number> defaults to 1. See UIEvent.detail.
- returns: <Promise>

Dispatches a `mouseup` event.

class: Touchscreen

touchscreen.tap(x, y)

- `x` <number>
- `y` <number>
- returns: <Promise>

Dispatches a `touchstart` and `touchend` event.

class: Tracing

You can use `tracing.start` and `tracing.stop` to create a trace file which can be opened in Chrome DevTools or timeline viewer.

```
await page.tracing.start({path: 'trace.json'});
await page.goto('https://www.google.com');
await page.tracing.stop();
```

tracing.start(options)

- `options` <Object>
  - `path` <string> A path to write the trace file to. **required**
  - `screenshots` <boolean> captures screenshots in the trace.
  - `categories` <Array<string>> specify custom categories to use instead of default.
- returns: <Promise>

Only one trace can be active at a time per browser.

tracing.stop()

- returns: <Promise>

class: Dialog

Dialog objects are dispatched by page via the 'dialog' event.

An example of using `Dialog` class:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  page.on('dialog', async dialog => {
    console.log(dialog.message());
    await dialog.dismiss();
    await browser.close();
  });
  page.evaluate(() => alert('1'));
});
```

dialog.accept([promptText])

- `promptText` <string> A text to enter in prompt. Does not cause any effects if the dialog's `type` is not prompt.
- returns: <Promise> Promise which resolves when the dialog has been accepted.

dialog.defaultValue()

- returns: <string> If dialog is prompt, returns default prompt value. Otherwise, returns empty string.

dialog.dismiss()

- returns: <Promise> Promise which resolves when the dialog has been dismissed.

dialog.message()

- returns: <string> A message displayed in the dialog.

dialog.type()

- returns: <string> Dialog's type, can be one of `alert`, `beforeunload`, `confirm` or `prompt`.

class: ConsoleMessage

ConsoleMessage objects are dispatched by page via the 'console' event.

consoleMessage.args()

- returns: <Array<JSHandle>>

consoleMessage.text()

- returns: <string>

consoleMessage.type()

- returns: <string>

One of the following values: `'log'`, `'debug'`, `'info'`, `'error'`, `'warning'`, `'dir'`, `'dirxml'`, `'table'`, `'trace'`, `'clear'`, `'startGroup'`, `'startGroupCollapsed'`, `'endGroup'`, `'assert'`, `'profile'`, `'profileEnd'`, `'count'`, `'timeEnd'`.

class: Frame

At every point of time, page exposes its current frame tree via the page.mainFrame() and frame.childFrames() methods.

Frame object's lifecycle is controlled by three events, dispatched on the page object:

- 'frameattached' - fired when the frame gets attached to the page. A Frame can be attached to the page only once.
- 'framenavigated' - fired when the frame commits navigation to a different URL.
- 'framedetached' - fired when the frame gets detached from the page. A Frame can be detached from the page only once.

An example of dumping frame tree:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
```

```
  await page.goto('https://www.google.com/chrome/browser/canary.html');
  dumpFrameTree(page.mainFrame(), '');
  await browser.close();

  function dumpFrameTree(frame, indent) {
    console.log(indent + frame.url());
    for (let child of frame.childFrames())
      dumpFrameTree(child, indent + '  ');
  }
});
```

frame.$(selector)

- `selector` <string> Selector to query page for
- returns: <Promise<?ElementHandle>> Promise which resolves to ElementHandle pointing to the frame element.

The method queries frame for the selector. If there's no such element within the frame, the method will resolve to `null`.

frame.$$(selector)

- `selector` <string> Selector to query page for
- returns: <Promise<Array<ElementHandle>>> Promise which resolves to ElementHandles pointing to the frame elements.

The method runs `document.querySelectorAll` within the frame. If no elements match the selector, the return value resolve to `[]`.

frame.$$eval(selector, pageFunction[, ...args])

- `selector` <string> A selector to query frame for
- `pageFunction` <function> Function to be evaluated in browser context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<Serializable>> Promise which resolves to the return value of `pageFunction`

This method runs `document.querySelectorAll` within the frame and passes it as the first argument to `pageFunction`.

If `pageFunction` returns a Promise, then `frame.$$eval` would wait for the promise to resolve and return its value.

Examples:

```
const divsCounts = await frame.$$eval('div', divs => divs.length);
```

frame.$eval(selector, pageFunction[, ...args])

- `selector` <string> A selector to query frame for
- `pageFunction` <function> Function to be evaluated in browser context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<Serializable>> Promise which resolves to the return value of `pageFunction`

This method runs `document.querySelector` within the frame and passes it as the first argument to `pageFunction`. If there's no element matching `selector`, the method throws an error.

If `pageFunction` returns a Promise, then `frame.$eval` would wait for the promise to resolve and return its value.

Examples:

```
const searchValue = await frame.$eval('#search', el => el.value);
const preloadHref = await frame.$eval('link[rel=preload]', el => el.href);
const html = await frame.$eval('.main-container', e => e.outerHTML);
```

frame.$x(expression)

- `expression` <string> Expression to evaluate.
- returns: <Promise<Array<ElementHandle>>>

The method evaluates the XPath expression.

frame.addScriptTag(options)

- `options` <Object>
  - `url` <string> Url of a script to be added.
  - `path` <string> Path to the JavaScript file to be injected into frame. If `path` is a relative path, then it is resolved relative to current working directory.
  - `content` <string> Raw JavaScript content to be injected into frame.
- returns: <Promise<ElementHandle>> which resolves to the added tag when the script's onload fires or when the script content was injected into frame.

Adds a `<script>` tag into the page with the desired url or content.

frame.addStyleTag(options)

- `options` <Object>
  - `url` <string> Url of the `<link>` tag.
  - `path` <string> Path to the CSS file to be injected into frame. If `path` is a relative path, then it is resolved relative to current working directory.
  - `content` <string> Raw CSS content to be injected into frame.
- returns: <Promise<ElementHandle>> which resolves to the added tag when the stylesheet's onload fires or when the CSS content was injected into frame.

Adds a `<link rel="stylesheet">` tag into the page with the desired url or a `<style type="text/css">` tag with the content.

frame.childFrames()

- returns: <Array<Frame>>

frame.click(selector[, options])

- `selector` <string> A selector to search for element to click. If there are multiple elements satisfying the selector, the first will be clicked.
- `options` <Object>

  - `button` <string> `left`, `right`, or `middle`, defaults to `left`.
  - `clickCount` <number> defaults to 1. See UIEvent.detail.
  - `delay` <number> Time to wait between `mousedown` and `mouseup` in milliseconds. Defaults to 0.
- returns: <Promise> Promise which resolves when the element matching `selector` is successfully clicked. The Promise will be rejected if there is no element matching `selector`.

This method fetches an element with `selector`, scrolls it into view if needed, and then uses page.mouse to click in the center of the element. If there's no element matching `selector`, the method throws an error.

Bare in mind that if `click()` triggers a navigation event and there's a separate `page.waitForNavigation()` promise to be resolved, you may end up with a race condition that yields unexpected results. The correct pattern for click and wait for navigation is the following:

```
const [response] = await Promise.all([
  page.waitForNavigation(waitOptions),
  frame.click(selector, clickOptions),
]);
```

frame.content()

- returns: <Promise<String>>

Gets the full HTML contents of the frame, including the doctype.

frame.evaluate(pageFunction, ...args)

- `pageFunction` <function|string> Function to be evaluated in browser context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<Serializable>> Promise which resolves to function return value

If the function, passed to the `frame.evaluate`, returns a Promise, then `frame.evaluate` would wait for the promise to resolve and return its value.

If the function passed into `frame.evaluate` returns a non-Serializable value, then `frame.evaluate` resolves to `undefined`.

```
const result = await frame.evaluate(() => {
  return Promise.resolve(8 * 7);
});
console.log(result); // prints "56"
```

A string can also be passed in instead of a function.

```
console.log(await frame.evaluate('1 + 2')); // prints "3"
```

ElementHandle instances can be passed as arguments to the `frame.evaluate`:

```
const bodyHandle = await frame.$('body');
const html = await frame.evaluate(body => body.innerHTML, bodyHandle);
await bodyHandle.dispose();
```

frame.evaluateHandle(pageFunction, ...args)

- `pageFunction` <function|string> Function to be evaluated in the page context
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<JSHandle>> Resolves to the return value of `pageFunction`

If the function, passed to the `frame.evaluateHandle`, returns a Promise, then `frame.evaluateHandle` would wait for the promise to resolve and return its value.

```
const aWindowHandle = await frame.evaluateHandle(() => Promise.resolve(window));
aWindowHandle; // Handle for the window object.
```

A string can also be passed in instead of a function.

```
const aHandle = await frame.evaluateHandle('document'); // Handle for the 'document'.
```

JSHandle instances can be passed as arguments to the `frame.evaluateHandle`:

```
const aHandle = await frame.evaluateHandle(() => document.body);
const resultHandle = await frame.evaluateHandle(body => body.innerHTML, aHandle);
console.log(await resultHandle.jsonValue());
await resultHandle.dispose();
```

frame.executionContext()

- returns: <Promise<ExecutionContext>> Execution context associated with this frame.

frame.focus(selector)

- `selector` <string> A selector of an element to focus. If there are multiple elements satisfying the selector, the first will be focused.
- returns: <Promise> Promise which resolves when the element matching `selector` is successfully focused. The promise will be rejected if there is no element matching `selector`.

This method fetches an element with `selector` and focuses it. If there's no element matching `selector`, the method throws an error.

frame.hover(selector)

- `selector` <string> A selector to search for element to hover. If there are multiple elements satisfying the selector, the first will be hovered.
- returns: <Promise> Promise which resolves when the element matching `selector` is successfully hovered. Promise gets rejected if there's no element matching `selector`.

This method fetches an element with `selector`, scrolls it into view if needed, and then uses page.mouse to hover over the center of the element. If there's no element matching `selector`, the method throws an error.

frame.isDetached()

- returns: <boolean>

Returns `true` if the frame has been detached, or `false` otherwise.

frame.name()

- returns: <string>

Returns frame's name attribute as specified in the tag.

If the name is empty, returns the id attribute instead.

> **NOTE** This value is calculated once when the frame is created, and will not update if the attribute is changed later.

frame.parentFrame()

- returns: <?Frame> Returns parent frame, if any. Detached frames and main frames return `null`.

frame.select(selector, ...values)

- `selector` <string> A selector to query frame for
- `...values` <...string> Values of options to select. If the `<select>` has the `multiple` attribute, all values are considered, otherwise only the first one is taken into account.
- returns: <Promise<Array<string>>> Returns an array of option values that have been successfully selected.

Triggers a `change` and `input` event once all the provided options have been selected. If there's no `<select>` element matching `selector`, the method throws an error.

```
frame.select('select#colors', 'blue'); // single selection
frame.select('select#colors', 'red', 'green', 'blue'); // multiple selections
```

frame.setContent(html)

- `html` <string> HTML markup to assign to the page.
- returns: <Promise>

frame.tap(selector)

- `selector` <string> A selector to search for element to tap. If there are multiple elements satisfying the selector, the first will be tapped.
- returns: <Promise>

This method fetches an element with `selector`, scrolls it into view if needed, and then uses page.touchscreen to tap in the center of the element. If there's no element matching `selector`, the method throws an error.

frame.title()

- returns: <Promise<string>> Returns page's title.

frame.type(selector, text[, options])

- `selector` <string> A selector of an element to type into. If there are multiple elements satisfying the selector, the first will be used.
- `text` <string> A text to type into a focused element.
- `options` <Object>
  - `delay` <number> Time to wait between key presses in milliseconds. Defaults to 0.
- returns: <Promise>

Sends a `keydown`, `keypress`/`input`, and `keyup` event for each character in the text.

To press a special key, like `Control` or `ArrowDown`, use `keyboard.press`.

```
frame.type('#mytextarea', 'Hello'); // Types instantly
frame.type('#mytextarea', 'World', {delay: 100}); // Types slower, like a user
```

frame.url()

- returns: <string>

Returns frame's url.

frame.waitFor(selectorOrFunctionOrTimeout[, options[, ...args]])

- `selectorOrFunctionOrTimeout` <string|number|function> A selector, predicate or timeout to wait for
- `options` <Object> Optional waiting parameters
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<JSHandle>> Promise which resolves to a JSHandle of the success value

This method behaves differently with respect to the type of the first parameter:

- if `selectorOrFunctionOrTimeout` is a `string`, then the first argument is treated as a selector or xpath, depending on whether or not it starts with '//', and the method is a shortcut for frame.waitForSelector or frame.waitForXPath
- if `selectorOrFunctionOrTimeout` is a `function`, then the first argument is treated as a predicate to wait for and the method is a shortcut for frame.waitForFunction().
- if `selectorOrFunctionOrTimeout` is a `number`, then the first argument is treated as a timeout in milliseconds and the method returns a promise which resolves after the timeout
- otherwise, an exception is thrown

frame.waitForFunction(pageFunction[, options[, ...args]])

- `pageFunction` <function|string> Function to be evaluated in browser context
- `options` <Object> Optional waiting parameters
  - `polling` <string|number> An interval at which the `pageFunction` is executed, defaults to `raf`. If `polling` is a number, then it is treated as an interval in milliseconds at which the function would be executed. If `polling` is a string, then it can be one of the following values:
    - `raf` - to constantly execute `pageFunction` in `requestAnimationFrame` callback. This is the tightest polling mode which is suitable to observe styling changes.
    - `mutation` - to execute `pageFunction` on every DOM mutation.
  - `timeout` <number> maximum time to wait for in milliseconds. Defaults to `30000` (30 seconds).

- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<JSHandle>> Promise which resolves when the `pageFunction` returns a truthy value. It resolves to a JSHandle of the truthy value.

The `waitForFunction` can be used to observe viewport size change:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  const watchDog = page.mainFrame().waitForFunction('window.innerWidth < 100');
  page.setViewport({width: 50, height: 50});
  await watchDog;
  await browser.close();
});
```

frame.waitForSelector(selector[, options])

- `selector` <string> A selector of an element to wait for,
- `options` <Object> Optional waiting parameters
  - `visible` <boolean> wait for element to be present in DOM and to be visible, i.e. to not have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.
  - `hidden` <boolean> wait for element to not be found in the DOM or to be hidden, i.e. have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.
  - `timeout` <number> maximum time to wait for in milliseconds. Defaults to `30000` (30 seconds).
- returns: <Promise<ElementHandle>> Promise which resolves when element specified by selector string is added to DOM.

Wait for the `selector` to appear in page. If at the moment of calling the method the `selector` already exists, the method will return immediately. If the selector doesn't appear after the `timeout` milliseconds of waiting, the function will throw.

This method works across navigations:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  let currentURL;
  page.mainFrame()
    .waitForSelector('img')
    .then(() => console.log('First URL with image: ' + currentURL));
  for (currentURL of ['https://example.com', 'https://google.com', 'https://bbc.com'])
    await page.goto(currentURL);
  await browser.close();
});
```

frame.waitForXPath(xpath[, options])

- `xpath` <string> A xpath of an element to wait for
- `options` <Object> Optional waiting parameters
  - `visible` <boolean> wait for element to be present in DOM and to be visible, i.e. to not have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.
  - `hidden` <boolean> wait for element to not be found in the DOM or to be hidden, i.e. have `display: none` or `visibility: hidden` CSS properties. Defaults to `false`.
  - `timeout` <number> maximum time to wait for in milliseconds. Defaults to `30000` (30 seconds).
- returns: <Promise<ElementHandle>> Promise which resolves when element specified by xpath string is added to DOM.

Wait for the `xpath` to appear in page. If at the moment of calling the method the `xpath` already exists, the method will return immediately. If the xpath doesn't appear after the `timeout` milliseconds of waiting, the function will throw.

This method works across navigations:

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  let currentURL;
```

```
  page.mainFrame()
    .waitForXPath('//img')
    .then(() => console.log('First URL with image: ' + currentURL));
  for (currentURL of ['https://example.com', 'https://google.com', 'https://bbc.com'])
    await page.goto(currentURL);
  await browser.close();
});
```

class: ExecutionContext

The class represents a context for JavaScript execution. Examples of JavaScript contexts are:

- each frame has a separate execution context
- all kind of workers have their own contexts

executionContext.evaluate(pageFunction, ...args)

- `pageFunction` <function|string> Function to be evaluated in `executionContext`
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<Serializable>> Promise which resolves to function return value

If the function, passed to the `executionContext.evaluate`, returns a Promise, then `executionContext.evaluate` would wait for the promise to resolve and return its value.

```
const executionContext = await page.mainFrame().executionContext();
const result = await executionContext.evaluate(() => Promise.resolve(8 * 7));
console.log(result); // prints "56"
```

A string can also be passed in instead of a function.

```
console.log(await executionContext.evaluate('1 + 2')); // prints "3"
```

JSHandle instances can be passed as arguments to the `executionContext.evaluate`:

```
const oneHandle = await executionContext.evaluateHandle(() => 1);
const twoHandle = await executionContext.evaluateHandle(() => 2);
const result = await executionContext.evaluate((a, b) => a + b, oneHandle, twoHandle);
await oneHandle.dispose();
await twoHandle.dispose();
console.log(result); // prints '3'.
```

executionContext.evaluateHandle(pageFunction, ...args)

- `pageFunction` <function|string> Function to be evaluated in the `executionContext`
- `...args` <...Serializable|JSHandle> Arguments to pass to `pageFunction`
- returns: <Promise<JSHandle>> Resolves to the return value of `pageFunction`

If the function, passed to the `executionContext.evaluateHandle`, returns a Promise, then `executionContext.evaluteHandle` would wait for the promise to resolve and return its value.

```
const context = await page.mainFrame().executionContext();
const aHandle = await context.evaluateHandle(() => Promise.resolve(self));
aHandle; // Handle for the global object.
```

A string can also be passed in instead of a function.

```
const aHandle = await context.evaluateHandle('1 + 2'); // Handle for the '3' object.
```

JSHandle instances can be passed as arguments to the `executionContext.evaluateHandle`:

```
const aHandle = await context.evaluateHandle(() => document.body);
const resultHandle = await context.evaluateHandle(body => body.innerHTML, aHandle);
console.log(await resultHandle.jsonValue()); // prints body's innerHTML
await aHandle.dispose();
await resultHandle.dispose();
```

executionContext.queryObjects(prototypeHandle)

- `prototypeHandle` <JSHandle> A handle to the object prototype.
- returns: <JSHandle> A handle to an array of objects with this prototype

The method iterates JavaScript heap and finds all the objects with the given prototype.

```
// Create a Map object
await page.evaluate(() => window.map = new Map());
// Get a handle to the Map object prototype
const mapPrototype = await page.evaluateHandle(() => Map.prototype);
// Query all map instances into an array
const mapInstances = await page.queryObjects(mapPrototype);
// Count amount of map objects in heap
const count = await page.evaluate(maps => maps.length, mapInstances);
await mapInstances.dispose();
await mapPrototype.dispose();
```

class: JSHandle

JSHandle represents an in-page JavaScript object. JSHandles can be created with the page.evaluateHandle method.

```
const windowHandle = await page.evaluateHandle(() => window);
// ...
```

JSHandle prevents references JavaScript objects from garbage collection unless the handle is disposed. JSHandles are auto-disposed when their origin frame gets navigated or the parent context gets destroyed.

JSHandle instances can be used as arguments in `page.$eval()`, `page.evaluate()` and `page.evaluateHandle` methods.

jsHandle.asElement()

- returns: <?ElementHandle>

Returns either `null` or the object handle itself, if the object handle is an instance of ElementHandle.

jsHandle.dispose()

- returns: <Promise> Promise which resolves when the object handle is successfully disposed.

The `jsHandle.dispose` method stops referencing the element handle.

jsHandle.executionContext()

- returns: ExecutionContext

Returns execution context the handle belongs to.

jsHandle.getProperties()

- returns: <Promise<Map<string, JSHandle>>>

The method returns a map with property names as keys and JSHandle instances for the property values.

```
const handle = await page.evaluateHandle(() => ({window, document}));
const properties = await handle.getProperties();
const windowHandle = properties.get('window');
const documentHandle = properties.get('document');
await handle.dispose();
```

jsHandle.getProperty(propertyName)

- `propertyName` <string> property to get
- returns: <Promise<JSHandle>>

Fetches a single property from the referenced object.

jsHandle.jsonValue()

- returns: <Promise<Object>>

Returns a JSON representation of the object. If the object has a `toJSON` function, it **will not be called**.

> **NOTE** The method will return an empty JSON if the referenced object is not stringifiable. It will throw an error if the object has circular references.

class: ElementHandle

> **NOTE** Class ElementHandle extends JSHandle.

ElementHandle represents an in-page DOM element. ElementHandles can be created with the page.$ method.

```
const puppeteer = require('puppeteer');

puppeteer.launch().then(async browser => {
  const page = await browser.newPage();
  await page.goto('https://google.com');
  const inputElement = await page.$('input[type=submit]');
  await inputElement.click();
  // ...
});
```

ElementHandle prevents DOM element from garbage collection unless the handle is disposed. ElementHandles are auto-disposed when their origin frame gets navigated.

ElementHandle instances can be used as arguments in `page.$eval()` and `page.evaluate()` methods.

elementHandle.$(selector)

- `selector` <string> A selector to query element for
- returns: <Promise<?ElementHandle>>

The method runs `element.querySelector` within the page. If no element matches the selector, the return value resolve to `null`.

elementHandle.$$(selector)

- `selector` <string> A selector to query element for
- returns: <Promise<Array<ElementHandle>>>

The method runs `element.querySelectorAll` within the page. If no elements match the selector, the return value resolve to `[]`.

elementHandle.$x(expression)

- `expression` <string> Expression to evaluate.
- returns: <Promise<?ElementHandle>> Promise which resolves to ElementHandle pointing to the frame element.

The method evaluates the XPath expression relative to the elementHandle. If there's no such element, the method will resolve to `null`.

elementHandle.asElement()

- returns: <elementhandle>

elementHandle.boundingBox()

- returns: <Promise<?Object>>
  - x <number> the x coordinate of the element in pixels.
  - y <number> the y coordinate of the element in pixels.
  - width <number> the width of the element in pixels.
  - height <number> the height of the element in pixels.

This method returns the bounding box of the element (relative to the main frame), or `null` if the element is not visible.

elementHandle.click([options])

- `options` <Object>
  - `button` <string> `left`, `right`, or `middle`, defaults to `left`.

- `clickCount` <number> defaults to 1. See UIEvent.detail.
- `delay` <number> Time to wait between `mousedown` and `mouseup` in milliseconds. Defaults to 0.

- returns: <Promise> Promise which resolves when the element is successfully clicked. Promise gets rejected if the element is detached from DOM.

This method scrolls element into view if needed, and then uses page.mouse to click in the center of the element. If the element is detached from DOM, the method throws an error.

elementHandle.dispose()

- returns: <Promise> Promise which resolves when the element handle is successfully disposed.

The `elementHandle.dispose` method stops referencing the element handle.

elementHandle.executionContext()

- returns: ExecutionContext

elementHandle.focus()

- returns: <Promise>

Calls focus on the element.

elementHandle.getProperties()

- returns: <Promise<Map<string, JSHandle>>>

The method returns a map with property names as keys and JSHandle instances for the property values.

```
const listHandle = await page.evaluateHandle(() => document.body.children);
const properties = await listHandle.getProperties();
const children = [];
for (const property of properties.values()) {
  const element = property.asElement();
```

```
  if (element)
    children.push(element);
}
children; // holds elementHandles to all children of document.body
```

elementHandle.getProperty(propertyName)

- `propertyName` <string> property to get
- returns: <Promise<JSHandle>>

Fetches a single property from the objectHandle.

elementHandle.hover()

- returns: <Promise> Promise which resolves when the element is successfully hovered.

This method scrolls element into view if needed, and then uses page.mouse to hover over the center of the element. If the element is detached from DOM, the method throws an error.

elementHandle.jsonValue()

- returns: <Promise<Object>>

Returns a JSON representation of the object. The JSON is generated by running `JSON.stringify` on the object in page and consequent `JSON.parse` in puppeteer.

> **NOTE** The method will throw if the referenced object is not stringifiable.

elementHandle.press(key[, options])

- `key` <string> Name of key to press, such as `ArrowLeft`. See USKeyboardLayout for a list of all key names.
- `options` <Object>
  - `text` <string> If specified, generates an input event with this text.
  - `delay` <number> Time to wait between `keydown` and `keyup` in milliseconds. Defaults to 0.

- returns: <Promise>

Focuses the element, and then uses `keyboard.down` and `keyboard.up`.

If `key` is a single character and no modifier keys besides `Shift` are being held down, a `keypress` / `input` event will also be generated. The `text` option can be specified to force an input event to be generated.

> **NOTE** Modifier keys DO effect `elementHandle.press`. Holding down `Shift` will type the text in upper case.

elementHandle.screenshot([options])

- `options` <Object> Same options as in page.screenshot.
- returns: <Promise<Buffer>> Promise which resolves to buffer with captured screenshot.

This method scrolls element into view if needed, and then uses page.screenshot to take a screenshot of the element. If the element is detached from DOM, the method throws an error.

elementHandle.tap()

- returns: <Promise> Promise which resolves when the element is successfully tapped. Promise gets rejected if the element is detached from DOM.

This method scrolls element into view if needed, and then uses touchscreen.tap to tap in the center of the element. If the element is detached from DOM, the method throws an error.

elementHandle.toString()

- returns: <string>

elementHandle.type(text[, options])

- `text` <string> A text to type into a focused element.
- `options` <Object>
  - `delay` <number> Time to wait between key presses in milliseconds. Defaults to 0.
- returns: <Promise>

Focuses the element, and then sends a `keydown`, `keypress`/`input`, and `keyup` event for each character in the text.

To press a special key, like `Control` or `ArrowDown`, use `elementHandle.press`.

```
elementHandle.type('Hello'); // Types instantly
elementHandle.type('World', {delay: 100}); // Types slower, like a user
```

An example of typing into a text field and then submitting the form:

```
const elementHandle = await page.$('input');
await elementHandle.type('some text');
await elementHandle.press('Enter');
```

elementHandle.uploadFile(...filePaths)

- `...filePaths` <...string> Sets the value of the file input these paths. If some of the `filePaths` are relative paths, then they are resolved relative to current working directory.
- returns: <Promise>

This method expects `elementHandle` to point to an input element.

class: Request

Whenever the page sends a request, the following events are emitted by puppeteer's page:

- 'request' emitted when the request is issued by the page.
- 'response' emitted when/if the response is received for the request.
- 'requestfinished' emitted when the response body is downloaded and the request is complete.

If request fails at some point, then instead of 'requestfinished' event (and possibly instead of 'response' event), the 'requestfailed' event is emitted.

If request gets a 'redirect' response, the request is successfully finished with the 'requestfinished' event, and a new request is issued to a redirected url.

request.abort([errorCode])

- `errorCode` <string> Optional error code. Defaults to `failed`, could be one of the following:
  - `aborted` - An operation was aborted (due to user action)
  - `accessdenied` - Permission to access a resource, other than the network, was denied
  - `addressunreachable` - The IP address is unreachable. This usually means that there is no route to the specified host or network.
  - `connectionaborted` - A connection timed out as a result of not receiving an ACK for data sent.
  - `connectionclosed` - A connection was closed (corresponding to a TCP FIN).
  - `connectionfailed` - A connection attempt failed.
  - `connectionrefused` - A connection attempt was refused.
  - `connectionreset` - A connection was reset (corresponding to a TCP RST).
  - `internetdisconnected` - The Internet connection has been lost.
  - `namenotresolved` - The host name could not be resolved.
  - `timedout` - An operation timed out.
  - `failed` - A generic failure occurred.
- returns: <Promise>

Aborts request. To use this, request interception should be enabled with `page.setRequestInterception`. Exception is immediately thrown if the request interception is not enabled.

request.continue([overrides])

- `overrides` <Object> Optional request overwrites, which can be one of the following:
  - `url` <string> If set, the request url will be changed
  - `method` <string> If set changes the request method (e.g. `GET` or `POST`)
  - `postData` <string> If set changes the post data of request

- `headers` <Object> If set changes the request HTTP headers

- returns: <Promise>

Continues request with optional request overrides. To use this, request interception should be enabled with `page.setRequestInterception`. Exception is immediately thrown if the request interception is not enabled.

request.failure()

- returns: <?Object> Object describing request failure, if any
  - `errorText` <string> Human-readable error message, e.g. `'net::ERR_FAILED'`.

The method returns `null` unless this request was failed, as reported by `requestfailed` event.

Example of logging all failed requests:

```
page.on('requestfailed', request => {
  console.log(request.url() + ' ' + request.failure().errorText);
});
```

request.frame()

- returns: <?Frame> A matching Frame object, or `null` if navigating to error pages.

request.headers()

- returns: <Object> An object with HTTP headers associated with the request. All header names are lower-case.

request.method()

- returns: <string> Request's method (GET, POST, etc.)

request.postData()

- returns: <string> Request's post body, if any.

request.resourceType()

- returns: <string>

Contains the request's resource type as it was perceived by the rendering engine. ResourceType will be one of the following: `document`, `stylesheet`, `image`, `media`, `font`, `script`, `texttrack`, `xhr`, `fetch`, `eventsource`, `websocket`, `manifest`, `other`.

request.respond(response)

- `response` <Object> Response that will fulfill this request
  - `status` <number> Response status code, defaults to `200`.
  - `headers` <Object> Optional response headers
  - `contentType` <string> If set, equals to setting `Content-Type` response header
  - `body` <Buffer|string> Optional response body
- returns: <Promise>

Fulfills request with given response. To use this, request interception should be enabled with `page.setRequestInterception`. Exception is thrown if request interception is not enabled.

An example of fulfilling all requests with 404 responses:

```
await page.setRequestInterception(true);
page.on('request', request => {
  request.respond({
    status: 404,
    contentType: 'text/plain',
    body: 'Not Found!'
  });
});
```

> **NOTE** Mocking responses for dataURL requests is not supported. Calling `request.respond` for a dataURL request is a noop.

request.response()

- returns: <?Response> A matching Response object, or `null` if the response has not been received yet.

request.url()

- returns: <string> URL of the request.

class: Response

Response class represents responses which are received by page.

response.buffer()

- returns: <Promise<Buffer>> Promise which resolves to a buffer with response body.

response.fromCache()

- returns: <boolean>

True if the response was served from either the browser's disk cache or memory cache.

response.fromServiceWorker()

- returns: <boolean>

True if the response was served by a service worker.

response.headers()

- returns: <Object> An object with HTTP headers associated with the response. All header names are lower-case.

response.json()

- returns: <Promise<Object>> Promise which resolves to a JSON representation of response body.

This method will throw if the response body is not parsable via `JSON.parse`.

response.ok()

- returns: <boolean>

Contains a boolean stating whether the response was successful (status in the range 200-299) or not.

response.request()

- returns: <Request> A matching Request object.

response.status()

- returns: <number>

Contains the status code of the response (e.g., 200 for a success).

response.text()

- returns: <Promise<string>> Promise which resolves to a text representation of response body.

response.url()

- returns: <string>

Contains the URL of the response.

class: Target

target.createCDPSession()

- returns: <Promise<CDPSession>>

Creates a Chrome Devtools Protocol session attached to the target.

target.page()

- returns: <Promise<?Page>>

If the target is not of type `"page"`, returns `null`.

target.type()

- returns: <string>

Identifies what kind of target this is. Can be `"page"`, `"service_worker"`, or `"other"`.

target.url()

- returns: <string>

class: CDPSession

- extends: `EventEmitter`

The `CDPSession` instances are used to talk raw Chrome Devtools Protocol:

- protocol methods can be called with `session.send` method.
- protocol events can be subscribed to with `session.on` method.

Documentation on DevTools Protocol can be found here: DevTools Protocol Viewer.

```
const client = await page.target().createCDPSession();
await client.send('Animation.enable');
await client.on('Animation.animationCreated', () => console.log('Animation created!'));
const response = await client.send('Animation.getPlaybackRate');
console.log('playback rate is ' + response.playbackRate);
await client.send('Animation.setPlaybackRate', {
  playbackRate: response.playbackRate / 2
});
```

cdpSession.detach()

- returns: <Promise>

Detaches session from target. Once detached, session won't emit any events and can't be used to send messages.

cdpSession.send(method[, params])

- `method` <string> protocol method name
- `params` <Object> Optional method parameters
- returns: <Promise<Object>>

class: Coverage

Coverage gathers information about parts of JavaScript and CSS that were used by the page.

An example of using JavaScript and CSS coverage to get percentage of initially executed code:

```
// Enable both JavaScript and CSS coverage
await Promise.all([
  page.coverage.startJSCoverage(),
  page.coverage.startCSSCoverage()
]);
// Navigate to page
await page.goto('https://example.com');
// Disable both JavaScript and CSS coverage
const [jsCoverage, cssCoverage] = await Promise.all([
  page.coverage.stopJSCoverage(),
  page.coverage.stopCSSCoverage(),
]);
let totalBytes = 0;
let usedBytes = 0;
const coverage = [...jsCoverage, ...cssCoverage];
for (const entry of coverage) {
  totalBytes += entry.text.length;
  for (const range of entry.ranges)
    usedBytes += range.end - range.start - 1;
```

```
}
console.log(`Bytes used: ${usedBytes / totalBytes * 100}%`);
```

coverage.startCSSCoverage(options)

- `options` <Object> Set of configurable options for coverage
  - `resetOnNavigation` <boolean> Whether to reset coverage on every navigation. Defaults to `true`.
- returns: <Promise> Promise that resolves when coverage is started

coverage.startJSCoverage(options)

- `options` <Object> Set of configurable options for coverage
  - `resetOnNavigation` <boolean> Whether to reset coverage on every navigation. Defaults to `true`.
- returns: <Promise> Promise that resolves when coverage is started

coverage.stopCSSCoverage()

- returns: <Promise<Array<Object>>> Promise that resolves to the array of coverage reports for all stylesheets
  - `url` <string> StyleSheet URL
  - `text` <string> StyleSheet content
  - `ranges` <Array<Object>> StyleSheet ranges that were used. Ranges are sorted and non-overlapping.
    - `start` <number> A start offset in text, inclusive
    - `end` <number> An end offset in text, exclusive

> **NOTE** CSS Coverage doesn't include dynamically injected style tags without sourceURLs.

coverage.stopJSCoverage()

- returns: <Promise<Array<Object>>> Promise that resolves to the array of coverage reports for all non-anonymous scripts
  - `url` <string> Script URL
  - `text` <string> Script content

- `ranges` <Array<Object>> Script ranges that were executed. Ranges are sorted and non-overlapping.
  - `start` <number> A start offset in text, inclusive
  - `end` <number> An end offset in text, exclusive

> **NOTE** JavaScript Coverage doesn't include anonymous scripts; however, scripts with sourceURLs are reported.