

Dismiss

## Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.


Sign up

Tag: **v1.20.0** ▼

Find file

Copy path

[puppeteer](#) / [docs](#) / [api.md](#)

 **mathiasbynens** chore: mark version v1.20.0 (#4928)

a5f03ce on 12 Sep

119 contributors



and others

Raw

Blame

History



3807 lines (2927 sloc) 184 KB

# Puppeteer API v1.20.0

- Interactive Documentation: <https://pptr.dev>
- API Translations: [中文|Chinese](#)
- Troubleshooting: [troubleshooting.md](#)
- Releases per Chromium Version:
  - Chromium 77.0.3803.0 - [Puppeteer v1.19.0](#)
  - Chromium 76.0.3803.0 - [Puppeteer v1.17.0](#)
  - Chromium 75.0.3765.0 - [Puppeteer v1.15.0](#)
  - Chromium 74.0.3723.0 - [Puppeteer v1.13.0](#)
  - Chromium 73.0.3679.0 - [Puppeteer v1.12.2](#)
  - [All releases](#)

## Table of Contents

- [Overview](#)
- [puppeteer vs puppeteer-core](#)
- [Environment Variables](#)
- [Working with Chrome Extensions](#)
- [class: Puppeteer](#)

- [puppeteer.connect\(options\)](#)
- [puppeteer.createBrowserFetcher\(\[options\]\)](#)
- [puppeteer.defaultArgs\(\[options\]\)](#)
- [puppeteer.devices](#)
- [puppeteer.errors](#)
- [puppeteer.executablePath\(\)](#)
- [puppeteer.launch\(\[options\]\)](#)
- [class: BrowserFetcher](#)
  - [browserFetcher.canDownload\(revision\)](#)
  - [browserFetcher.download\(revision\[, progressCallback\]\)](#)
  - [browserFetcher.localRevisions\(\)](#)
  - [browserFetcher.platform\(\)](#)
  - [browserFetcher.remove\(revision\)](#)
  - [browserFetcher.revisionInfo\(revision\)](#)
- [class: Browser](#)
  - [event: 'disconnected'](#)
  - [event: 'targetchanged'](#)
  - [event: 'targetcreated'](#)
  - [event: 'targetdestroyed'](#)
  - [browser.browserContexts\(\)](#)
  - [browser.close\(\)](#)
  - [browser.createIncognitoBrowserContext\(\)](#)
  - [browser.defaultBrowserContext\(\)](#)
  - [browser.disconnect\(\)](#)
  - [browser.isConnected\(\)](#)
  - [browser.newPage\(\)](#)
  - [browser.pages\(\)](#)
  - [browser.process\(\)](#)
  - [browser.target\(\)](#)
  - [browser.targets\(\)](#)
  - [browser.userAgent\(\)](#)
  - [browser.version\(\)](#)
  - [browser.waitForTarget\(predicate\[, options\]\)](#)
  - [browser.wsEndpoint\(\)](#)
- [class: BrowserContext](#)
  - [event: 'targetchanged'](#)
  - [event: 'targetcreated'](#)
  - [event: 'targetdestroyed'](#)
  - [browserContext.browser\(\)](#)
  - [browserContext.clearPermissionOverrides\(\)](#)

- `browserContext.close()`
- `browserContext.isIncognito()`
- `browserContext.newPage()`
- `browserContext.overridePermissions(origin, permissions)`
- `browserContext.pages()`
- `browserContext.targets()`
- `browserContext.waitForTarget(predicate[, options])`
- `class: Page`
  - `event: 'close'`
  - `event: 'console'`
  - `event: 'dialog'`
  - `event: 'domcontentloaded'`
  - `event: 'error'`
  - `event: 'frameattached'`
  - `event: 'framedetached'`
  - `event: 'framenavigated'`
  - `event: 'load'`
  - `event: 'metrics'`
  - `event: 'pageerror'`
  - `event: 'popup'`
  - `event: 'request'`
  - `event: 'requestfailed'`
  - `event: 'requestfinished'`
  - `event: 'response'`
  - `event: 'workercreated'`
  - `event: 'workerdestroyed'`
  - `page.$(selector)`
  - `page.$$ (selector)`
  - `page.$$eval(selector, pageFunction[, ...args])`
  - `page.$eval(selector, pageFunction[, ...args])`
  - `page.$x(expression)`
  - `page.accessibility`
  - `page.addScriptTag(options)`
  - `page.addStyleTag(options)`
  - `page.authenticate(credentials)`
  - `page.bringToFront()`
  - `page.browser()`
  - `page.browserContext()`
  - `page.click(selector[, options])`
  - `page.close([options])`

- `page.content()`
- `page.cookies([...urls])`
- `page.coverage`
- `page.deleteCookie(...cookies)`
- `page.emulate(options)`
- `page.emulateMedia(mediaType)`
- `page.evaluate(pageFunction[, ...args])`
- `page.evaluateHandle(pageFunction[, ...args])`
- `page.evaluateOnNewDocument(pageFunction[, ...args])`
- `page.exposeFunction(name, puppeteerFunction)`
- `page.focus(selector)`
- `page.frames()`
- `page.goBack([options])`
- `page.goForward([options])`
- `page.goto(url[, options])`
- `page.hover(selector)`
- `page.isClosed()`
- `page.keyboard`
- `page.mainFrame()`
- `page.metrics()`
- `page.mouse`
- `page.pdf([options])`
- `page.queryObjects(prototypeHandle)`
- `page.reload([options])`
- `page.screenshot([options])`
- `page.select(selector, ...values)`
- `page.setBypassCSP(enabled)`
- `page.setCacheEnabled([enabled])`
- `page.setContent(html[, options])`
- `page.setCookie(...cookies)`
- `page.setDefaultNavigationTimeout(timeout)`
- `page.setDefaultTimeout(timeout)`
- `page.setExtraHTTPHeaders(headers)`
- `page.setGeolocation(options)`
- `page.setJavaScriptEnabled(enabled)`
- `page.setOfflineMode(enabled)`
- `page.setRequestInterception(value)`
- `page.setUserAgent(userAgent)`
- `page.setViewport(viewport)`
- `page.tap(selector)`

- `page.target()`
- `page.title()`
- `page.touchscreen`
- `page.tracing`
- `page.type(selector, text[, options])`
- `page.url()`
- `page.viewport()`
- `page.waitFor(selectorOrFunctionOrTimeout[, options[, ...args]])`
- `page.waitForFileChooser([options])`
- `page.waitForFunction(pageFunction[, options[, ...args]])`
- `page.waitForNavigation([options])`
- `page.waitForRequest(urlOrPredicate[, options])`
- `page.waitForResponse(urlOrPredicate[, options])`
- `page.waitForSelector(selector[, options])`
- `page.waitForXPath(xpath[, options])`
- `page.workers()`
- **class: Worker**
  - `worker.evaluate(pageFunction[, ...args])`
  - `worker.evaluateHandle(pageFunction[, ...args])`
  - `worker.executionContext()`
  - `worker.url()`
- **class: Accessibility**
  - `accessibility.snapshot([options])`
- **class: Keyboard**
  - `keyboard.down(key[, options])`
  - `keyboard.press(key[, options])`
  - `keyboard.sendCharacter(char)`
  - `keyboard.type(text[, options])`
  - `keyboard.up(key)`
- **class: Mouse**
  - `mouse.click(x, y[, options])`
  - `mouse.down([options])`
  - `mouse.move(x, y[, options])`
  - `mouse.up([options])`
- **class: Touchscreen**
  - `touchscreen.tap(x, y)`
- **class: Tracing**
  - `tracing.start([options])`
  - `tracing.stop()`
- **class: FileChooser**
  - `fileChooser.accept(filePaths)`

- `fileChooser.cancel()`
- `fileChooser.isMultiple()`
- class: `Dialog`
  - `dialog.accept([promptText])`
  - `dialog.defaultValue()`
  - `dialog.dismiss()`
  - `dialog.message()`
  - `dialog.type()`
- class: `ConsoleMessage`
  - `consoleMessage.args()`
  - `consoleMessage.location()`
  - `consoleMessage.text()`
  - `consoleMessage.type()`
- class: `Frame`
  - `frame.$(selector)`
  - `frame.$$ (selector)`
  - `frame.$$eval(selector, pageFunction[, ...args])`
  - `frame.$eval(selector, pageFunction[, ...args])`
  - `frame.$x(expression)`
  - `frame.addScriptTag(options)`
  - `frame.addStyleTag(options)`
  - `frame.childFrames()`
  - `frame.click(selector[, options])`
  - `frame.content()`
  - `frame.evaluate(pageFunction[, ...args])`
  - `frame.evaluateHandle(pageFunction[, ...args])`
  - `frame.executionContext()`
  - `frame.focus(selector)`
  - `frame.goto(url[, options])`
  - `frame.hover(selector)`
  - `frame.isDetached()`
  - `frame.name()`
  - `frame.parentFrame()`
  - `frame.select(selector, ...values)`
  - `frame.setContent(html[, options])`
  - `frame.tap(selector)`
  - `frame.title()`
  - `frame.type(selector, text[, options])`
  - `frame.url()`
  - `frame.waitFor(selectorOrFunctionOrTimeout[, options[, ...args]])`

- `frame.waitForFunction(pageFunction[, options[, ...args]])`
- `frame.waitForNavigation([options])`
- `frame.waitForSelector(selector[, options])`
- `frame.waitForXPath(xpath[, options])`
- **class: ExecutionContext**
  - `executionContext.evaluate(pageFunction[, ...args])`
  - `executionContext.evaluateHandle(pageFunction[, ...args])`
  - `executionContext.frame()`
  - `executionContext.queryObjects(prototypeHandle)`
- **class: JSHandle**
  - `jsHandle.asElement()`
  - `jsHandle.dispose()`
  - `jsHandle.evaluate(pageFunction[, ...args])`
  - `jsHandle.evaluateHandle(pageFunction[, ...args])`
  - `jsHandle.executionContext()`
  - `jsHandle.getProperties()`
  - `jsHandle.getProperty(propertyName)`
  - `jsHandle.jsonValue()`
- **class: ElementHandle**
  - `elementHandle.$(selector)`
  - `elementHandle.$$$(selector)`
  - `elementHandle.$$eval(selector, pageFunction[, ...args])`
  - `elementHandle.$eval(selector, pageFunction[, ...args])`
  - `elementHandle.$x(expression)`
  - `elementHandle.asElement()`
  - `elementHandle.boundingBox()`
  - `elementHandle.boxModel()`
  - `elementHandle.click([options])`
  - `elementHandle.contentFrame()`
  - `elementHandle.dispose()`
  - `elementHandle.evaluate(pageFunction[, ...args])`
  - `elementHandle.evaluateHandle(pageFunction[, ...args])`
  - `elementHandle.executionContext()`
  - `elementHandle.focus()`
  - `elementHandle.getProperties()`
  - `elementHandle.getProperty(propertyName)`
  - `elementHandle.hover()`
  - `elementHandle.isIntersectingViewport()`
  - `elementHandle.jsonValue()`
  - `elementHandle.press(key[, options])`

- `elementHandle.screenshot([options])`
- `elementHandle.select(...values)`
- `elementHandle.tap()`
- `elementHandle.toString()`
- `elementHandle.type(text[, options])`
- `elementHandle.uploadFile(...filePaths)`
- **class: Request**
  - `request.abort([errorCode])`
  - `request.continue([overrides])`
  - `request.failure()`
  - `request.frame()`
  - `request.headers()`
  - `request.isNavigationRequest()`
  - `request.method()`
  - `request.postData()`
  - `request.redirectChain()`
  - `request.resourceType()`
  - `request.respond(response)`
  - `request.response()`
  - `request.url()`
- **class: Response**
  - `response.buffer()`
  - `response.frame()`
  - `response.fromCache()`
  - `response.fromServiceWorker()`
  - `response.headers()`
  - `response.json()`
  - `response.ok()`
  - `response.remoteAddress()`
  - `response.request()`
  - `response.securityDetails()`
  - `response.status()`
  - `response.statusText()`
  - `response.text()`
  - `response.url()`
- **class: SecurityDetails**
  - `securityDetails.issuer()`
  - `securityDetails.protocol()`
  - `securityDetails.subjectName()`
  - `securityDetails.validFrom()`



- `securityDetails.validTo()`
- class: `Target`
  - `target.browser()`
  - `target.browserContext()`
  - `target.createCDPSession()`
  - `target.opener()`
  - `target.page()`
  - `target.type()`
  - `target.url()`
  - `target.worker()`
- class: `CDPSession`
  - `cdpSession.detach()`
  - `cdpSession.send(method[, params])`
- class: `Coverage`
  - `coverage.startCSSCoverage([options])`
  - `coverage.startJSCoverage([options])`
  - `coverage.stopCSSCoverage()`
  - `coverage.stopJSCoverage()`
- class: `TimeoutError`

## 🔗 Overview

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

The Puppeteer API is hierarchical and mirrors the browser structure.

**NOTE** On the following diagram, faded entities are not currently represented in Puppeteer.

- [Puppeteer](#) communicates with the browser using [DevTools Protocol](#).
- [Browser](#) instance can own multiple browser contexts.
- [BrowserContext](#) instance defines a browsing session and can own 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 - the default execution context - where the frame's JavaScript is executed. A Frame might have additional execution contexts that are associated with [extensions](#).
- [Worker](#) has a single execution context and facilitates interacting with [WebWorkers](#).

(Diagram source: [link](#))

## puppeteer vs puppeteer-core

Every release since v1.7.0 we publish two packages:

- [puppeteer](#)
- [puppeteer-core](#)

`puppeteer` is a *product* for browser automation. When installed, it downloads a version of Chromium, which it then drives using `puppeteer-core`. Being an end-user product, `puppeteer` supports a bunch of convenient `PUPPETEER_*` env variables to tweak its behavior.

`puppeteer-core` is a *library* to help drive anything that supports DevTools protocol. `puppeteer-core` doesn't download Chromium when installed. Being a library, `puppeteer-core` is fully driven through its programmatic interface and disregards all the `PUPPETEER_*` env variables.

To sum up, the only differences between `puppeteer-core` and `puppeteer` are:

- `puppeteer-core` doesn't automatically download Chromium when installed.
- `puppeteer-core` ignores all `PUPPETEER_*` env variables.

In most cases, you'll be fine using the `puppeteer` package.

However, you should use `puppeteer-core` if:

- you're building another end-user product or library atop of DevTools protocol. For example, one might build a PDF generator using `puppeteer-core` and write a custom `install.js` script that downloads `headless_shell` instead of Chromium to save disk space.
- you're bundling Puppeteer to use in Chrome Extension / browser with the DevTools protocol where downloading an additional Chromium binary is unnecessary.

When using `puppeteer-core`, remember to change the *include* line:

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

You will then need to call `puppeteer.connect([options])` or `puppeteer.launch([options])` with an explicit `executablePath` option.

## 🔗 Environment Variables

Puppeteer looks for certain [environment variables](#) to aid its operations. If Puppeteer doesn't find them in the environment during the installation step, a lowercased variant of these variables will be used from 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 URL prefix that is used to download Chromium. Note: this includes protocol and might even include path prefix. Defaults to `https://storage.googleapis.com`.
- `PUPPETEER_CHROMIUM_REVISION` - specify a certain version of Chromium you'd like Puppeteer to use. See [puppeteer.launch\(\[options\]\)](#) on how executable path is inferred.

**BEWARE:** Puppeteer is only [guaranteed to work](#) with the bundled Chromium, use at your own risk.

- `PUPPETEER_EXECUTABLE_PATH` - specify an executable path to be used in `puppeteer.launch`. See [puppeteer.launch\(\[options\]\)](#) on how the executable path is inferred. **BEWARE:** Puppeteer is only [guaranteed to work](#) with the bundled Chromium, use at your own risk.

**NOTE** `PUPPETEER_*` env variables are not accounted for in the [puppeteer-core](#) package.

## 🔗 Working with Chrome Extensions

Puppeteer can be used for testing Chrome Extensions.

**NOTE** Extensions in Chrome / Chromium currently only work in non-headless mode.

The following is code for getting a handle to the [background page](#) of an extension whose source is located in `./my-extension`:

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

(async () => {
  const pathToExtension = require('path').join(__dirname, 'my-extension');
  const browser = await puppeteer.launch({
    headless: false,
    args: [
      '--disable-extensions-except=${pathToExtension}',
      '--load-extension=${pathToExtension}'
    ]
  });
  const targets = await browser.targets();
  const backgroundPageTarget = targets.find(target => target.type() === 'background');
  const backgroundPage = await backgroundPageTarget.page();
  // Test the background page as you would any other page.
  await browser.close();
})();
```

**NOTE** It is not yet possible to test extension popups or content scripts.

## 🔗 class: Puppeteer

Puppeteer module provides a method to launch a Chromium instance. The following is a typical example of using 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.
  - browserURL <?string> a browser url to connect to, in format `http://${host}:${port}` . Use interchangeably with `browserWSEndpoint` to let Puppeteer fetch it from [metadata endpoint](#).
  - ignoreHTTPSErrors <boolean> Whether to ignore HTTPS errors during navigation. Defaults to `false` .
  - defaultViewport <?Object> Sets a consistent viewport for each page. Defaults to an 800x600 viewport. `null` disables the default viewport.
    - 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` .
  - slowMo <number> Slows down Puppeteer operations by the specified amount of milliseconds. Useful so that you can see what is going on.
  - transport <ConnectionTransport> **Experimental** Specify a custom transport object for Puppeteer to use.
- returns: <Promise<Browser>>

This methods attaches Puppeteer to an existing Chromium instance.

## puppeteer.createBrowserFetcher([options])

- options <Object>
  - host <string> A download host to be used. Defaults to `https://storage.googleapis.com` .
  - path <string> A path for the downloads folder. Defaults to `<root>/local-chromium` , where `<root>` is puppeteer's package root.
  - platform <string> Possible values are: `mac` , `win32` , `win64` , `linux` . Defaults to the current platform.
- returns: <BrowserFetcher>

## puppeteer.defaultArgs([options])

- options <Object> Set of configurable options to set on the browser. Can have the following fields:

- `headless` <boolean> Whether to run browser in [headless mode](#). Defaults to `true` unless the `devtools` option is `true`.
- `args` <Array<string>> Additional arguments to pass to the browser instance. The list of Chromium flags can be found [here](#).
- `userDataDir` <string> Path to a [User Data Directory](#).
- `devtools` <boolean> Whether to auto-open a DevTools panel for each tab. If this option is `true`, the `headless` option will be set `false`.
- returns: <Array<string>>

The default flags that Chromium will be launched with.

## `puppeteer.devices`

- returns: <Object>

Returns a list of devices to be used with `page.emulate(options)`. Actual list of devices can be found in [lib/DeviceDescriptors.js](#).

```
const puppeteer = require('puppeteer');
const iPhone = puppeteer.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();
});
```

**NOTE** The old way (Puppeteer versions <= v1.14.0) devices can be obtained with `require('puppeteer/DeviceDescriptors')`.

## `puppeteer.errors`

- returns: <Object>
  - `TimeoutError` <function> A class of [TimeoutError](#).

Puppeteer methods might throw errors if they are unable to fulfill a request. For example, `page.waitForSelector(selector[, options])` might fail if the selector doesn't match any nodes during the given timeframe.

For certain types of errors Puppeteer uses specific error classes. These classes are available via [puppeteer.errors](#)

An example of handling a timeout error:

```
try {
  await page.waitForSelector('.foo');
} catch (e) {
  if (e instanceof puppeteer.errors.TimeoutError) {
```

```
    // Do something if this is a timeout.  
  }  
}
```

**NOTE** The old way (Puppeteer versions <= v1.14.0) errors can be obtained with `require('puppeteer/Errors')` .

## `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` .

**NOTE** `puppeteer.executablePath()` is affected by the `PUPPETEER_EXECUTABLE_PATH` and `PUPPETEER_CHROMIUM_REVISION` env variables. See [Environment Variables](#) for details.

## `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 the bundled Chromium. If `executablePath` is a relative path, then it is resolved relative to [current working directory](#). **BEWARE:** Puppeteer is only [guaranteed to work](#) with the bundled Chromium, use at your own risk.
  - `slowMo` `<number>` Slows down Puppeteer operations by the specified amount of milliseconds. Useful so that you can see what is going on.
  - `defaultViewport` `<?Object>` Sets a consistent viewport for each page. Defaults to an 800x600 viewport. `null` disables the default viewport.
    - `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` .
  - `args` `<Array<string>>` Additional arguments to pass to the browser instance. The list of Chromium flags can be found [here](#).

- `ignoreDefaultArgs` [<boolean|Array<string>>](#) If `true`, then do not use `puppeteer.defaultArgs()`. If an array is given, then filter out the given default arguments. Dangerous option; use with care. Defaults to `false`.
  - `handleSIGINT` [<boolean>](#) Close the browser process on Ctrl-C. Defaults to `true`.
  - `handleSIGTERM` [<boolean>](#) Close the browser process on SIGTERM. Defaults to `true`.
  - `handleSIGHUP` [<boolean>](#) Close the 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 the 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 the browser. Defaults to `process.env`.
  - `devtools` [<boolean>](#) Whether to auto-open a DevTools panel for each tab. If this option is `true`, the `headless` option will be set `false`.
  - `pipe` [<boolean>](#) Connects to the browser over a pipe instead of a WebSocket. Defaults to `false`.
- `returns`: [<Promise<Browser>>](#) Promise which resolves to browser instance.

You can use `ignoreDefaultArgs` to filter out `--mute-audio` from default arguments:

```
const browser = await puppeteer.launch({
  ignoreDefaultArgs: ['--mute-audio']
});
```

**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: BrowserFetcher

BrowserFetcher can download and manage different versions of Chromium.

BrowserFetcher operates on revision strings that specify a precise version of Chromium, e.g. `"533271"`. Revision strings can be obtained from [omahaproxy.appspot.com](#).



An example of using BrowserFetcher to download a specific version of Chromium and running Puppeteer against it:

```
const browserFetcher = puppeteer.createBrowserFetcher();
const revisionInfo = await browserFetcher.download('533271');
const browser = await puppeteer.launch({executablePath: revisionInfo.executablePath});
```

**NOTE** BrowserFetcher is not designed to work concurrently with other instances of BrowserFetcher that share the same downloads directory.

### browserFetcher.canDownload(revision)

- revision `<string>` a revision to check availability.
- returns: `<Promise<boolean>>` returns `true` if the revision could be downloaded from the host.

The method initiates a HEAD request to check if the revision is available.

### browserFetcher.download(revision[, progressCallback])

- revision `<string>` a revision to download.
- progressCallback `<function(number, number)>` A function that will be called with two arguments:
  - downloadedBytes `<number>` how many bytes have been downloaded
  - totalBytes `<number>` how large is the total download.
- returns: `<Promise<Object>>` Resolves with revision information when the revision is downloaded and extracted
  - revision `<string>` the revision the info was created from
  - folderPath `<string>` path to the extracted revision folder
  - executablePath `<string>` path to the revision executable
  - url `<string>` URL this revision can be downloaded from
  - local `<boolean>` whether the revision is locally available on disk

The method initiates a GET request to download the revision from the host.

### browserFetcher.localRevisions()

- returns: `<Promise<Array<string>>>` A list of all revisions available locally on disk.

### browserFetcher.platform()

- returns: `<string>` One of `mac` , `linux` , `win32` or `win64` .

### browserFetcher.remove(revision)

- revision `<string>` a revision to remove. The method will throw if the revision has not been downloaded.
- returns: `<Promise>` Resolves when the revision has been removed.

## 🔗 `browserFetcher.revisionInfo(revision)`

- `revision` `<string>` a revision to get info for.
- returns: `<Object>`
  - `revision` `<string>` the revision the info was created from
  - `folderPath` `<string>` path to the extracted revision folder
  - `executablePath` `<string>` path to the revision executable
  - `url` `<string>` URL this revision can be downloaded from
  - `local` `<boolean>` whether the revision is locally available on disk

## 🔗 **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 of one of the following:

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

### 🔗 event: 'targetchanged'

- [<Target>](#)

Emitted when the url of a target changes.

**NOTE** This includes target changes in incognito browser contexts.

### 🔗 event: 'targetcreated'

- [<Target>](#)

Emitted when a target is created, for example when a new page is opened by [window.open](#) or [browser.newPage](#) .

**NOTE** This includes target creations in incognito browser contexts.

### 🔗 event: 'targetdestroyed'

- [<Target>](#)

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

**NOTE** This includes target destructions in incognito browser contexts.

### 🔗 browser.browserContexts()

- returns: [<Array<BrowserContext>>](#)

Returns an array of all open browser contexts. In a newly created browser, this will return a single instance of [BrowserContext](#).

### 🔗 browser.close()

- returns: [<Promise>](#)

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

### 🔗 browser.createIncognitoBrowserContext()

- returns: [<Promise<BrowserContext>>](#)

Creates a new incognito browser context. This won't share cookies/cache with other browser contexts.

```
const browser = await puppeteer.launch();
// Create a new incognito browser context.
const context = await browser.createIncognitoBrowserContext();
// Create a new page in a pristine context.
const page = await context.newPage();
// Do stuff
await page.goto('https://example.com');
```

## 🔗 **browser.defaultBrowserContext()**

- returns: <[BrowserContext](#)>

Returns the default browser context. The default browser context can not be closed.

## 🔗 **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.isConnected()**

- returns: <[boolean](#)>

Indicates that the browser is connected.

## 🔗 **browser.newPage()**

- returns: <[Promise](#)<[Page](#)>>

Promise which resolves to a new [Page](#) object. The [Page](#) is created in a default browser context.

## 🔗 **browser.pages()**

- returns: <[Promise](#)<[Array](#)<[Page](#)>>> Promise which resolves to an array of all open pages. Non visible pages, such as "background\_page", will not be listed here. You can find them using [target.page\(\)](#).

An array of all pages inside the Browser. In case of multiple browser contexts, the method will return an array with all the pages in all browser contexts.

## 🔗 **browser.process()**

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

## 🔗 **browser.target()**

- returns: <[Target](#)>

A target associated with the browser.

## 🔗 **browser.targets()**

- returns: <[Array](#)<[Target](#)>>

An array of all active targets inside the Browser. In case of multiple browser contexts, the method will return an array with all the targets in all browser contexts.

## 🔗 **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.waitForTarget(predicate[, options])**

- predicate `<function(Target):boolean>` A function to be run for every target
- options `<Object>`
  - timeout `<number>` Maximum wait time in milliseconds. Pass `0` to disable the timeout. Defaults to 30 seconds.
- returns: `<Promise<Target>>` Promise which resolves to the first target found that matches the predicate function.

This searches for a target in all browser contexts.

An example of finding a target for a page opened via `window.open` :

```
await page.evaluate(() => window.open('https://www.example.com/'));
const newWindowTarget = await browser.waitForTarget(target => target.url() ===
```

## 🔗 **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: BrowserContext**

- extends: [EventEmitter](#)

BrowserContexts provide a way to operate multiple independent browser sessions. When a browser is launched, it has a single BrowserContext used by default. The method `browser.newPage()` creates a page in the default browser context.

If a page opens another page, e.g. with a `window.open` call, the popup will belong to the parent page's browser context.

Puppeteer allows creation of "incognito" browser contexts with

`browser.createIncognitoBrowserContext()` method. "Incognito" browser contexts don't write any browsing data to disk.

```
// Create a new incognito browser context
const context = await browser.createIncognitoBrowserContext();
// Create a new page inside context.
const page = await context.newPage();
// ... do stuff with page ...
await page.goto('https://example.com');
// Dispose context once it's no longer needed.
await context.close();
```

### **event: 'targetchanged'**

- `<Target>`

Emitted when the url of a target inside the browser context changes.

### **event: 'targetcreated'**

- `<Target>`

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

### **event: 'targetdestroyed'**

- `<Target>`

Emitted when a target inside the browser context is destroyed, for example when a page is closed.

### **browserContext.browser()**

- returns: `<Browser>`

The browser this browser context belongs to.

### **browserContext.clearPermissionOverrides()**

- returns: `<Promise>`

Clears all permission overrides for the browser context.

```
const context = browser.defaultBrowserContext();
context.overridePermissions('https://example.com', ['clipboard-read']);
// do stuff ..
context.clearPermissionOverrides();
```

## 🔗 `browserContext.close()`

- returns: `<Promise>`

Closes the browser context. All the targets that belong to the browser context will be closed.

**NOTE** only incognito browser contexts can be closed.

## 🔗 `browserContext.isIncognito()`

- returns: `<boolean>`

Returns whether BrowserContext is incognito. The default browser context is the only non-incognito browser context.

**NOTE** the default browser context cannot be closed.

## 🔗 `browserContext.newPage()`

- returns: `<Promise<Page>>`

Creates a new page in the browser context.

## 🔗 `browserContext.overridePermissions(origin, permissions)`

- origin `<string>` The [origin](#) to grant permissions to, e.g. "<https://example.com>".
- permissions `<Array<string>>` An array of permissions to grant. All permissions that are not listed here will be automatically denied. Permissions can be one of the following values:
  - `'geolocation'`
  - `'midi'`
  - `'midi-sysex'` (system-exclusive midi)
  - `'notifications'`
  - `'push'`
  - `'camera'`
  - `'microphone'`
  - `'background-sync'`
  - `'ambient-light-sensor'`
  - `'accelerometer'`
  - `'gyroscope'`
  - `'magnetometer'`
  - `'accessibility-events'`
  - `'clipboard-read'`
  - `'clipboard-write'`
  - `'payment-handler'`

- returns: [<Promise>](#)

```
const context = browser.defaultBrowserContext();
await context.overridePermissions('https://html5demos.com', ['geolocation']);
```

## 🔗 **browserContext.pages()**

- returns: [<Promise<Array<Page>>>](#) Promise which resolves to an array of all open pages. Non visible pages, such as "background\_page" , will not be listed here. You can find them using [target.page\(\)](#).

An array of all pages inside the browser context.

## 🔗 **browserContext.targets()**

- returns: [<Array<Target>>](#)

An array of all active targets inside the browser context.

## 🔗 **browserContext.waitForTarget(predicate[, options])**

- predicate [<function\(Target\):boolean>](#) A function to be run for every target
- options [<Object>](#)
  - timeout [<number>](#) Maximum wait time in milliseconds. Pass 0 to disable the timeout. Defaults to 30 seconds.
- returns: [<Promise<Target>>](#) Promise which resolves to the first target found that matches the predicate function.

This searches for a target in this specific browser context.

An example of finding a target for a page opened via `window.open` :

```
await page.evaluate(() => window.open('https://www.example.com/'));
const newWindowTarget = await browserContext.waitForTarget(target => target.url === 'https://www.example.com/');
```

## 🔗 **class: Page**

- extends: [EventEmitter](#)

Page provides methods to interact with a single tab or [extension background page](#) 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`, `once` or `removeListener`.

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

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

To unsubscribe from events use the `removeListener` method:

```
function logRequest(interceptedRequest) {
  console.log('A request was made:', interceptedRequest.url());
}
page.on('request', logRequest);
// Sometime later...
page.removeListener('request', logRequest);
```

#### **event: 'close'**

Emitted when the page closes.

#### **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: 'popup'

- `<Page>` Page corresponding to "popup" window

Emitted when the page opens a new tab or window.

```
const [popup] = await Promise.all([
  new Promise(resolve => page.once('popup', resolve)),
  page.click('a[target=_blank]'),
]);

const [popup] = await Promise.all([
  new Promise(resolve => page.once('popup', resolve)),
  page.evaluate(() => window.open('https://example.com')),
]);
```

## 🔗 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.

**NOTE** HTTP Error responses, such as 404 or 503, are still successful responses from HTTP standpoint, so request will complete with `'requestfinished'` event and not with `'requestfailed'`.

## 🔗 event: 'requestfinished'

- [<Request>](#)

Emitted when a request finishes successfully.

## 🔗 event: 'response'

- [<Response>](#)

Emitted when a [response](#) is received.

## 🔗 event: 'workercreated'

- [<Worker>](#)

Emitted when a dedicated [WebWorker](#) is spawned by the page.

## 🔗 event: 'workerdestroyed'

- [<Worker>](#)

Emitted when a dedicated [WebWorker](#) is terminated.

## 🔗 `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 resolves 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 resolves to `[]`.

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

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

- `selector` `<string>` A `selector` to query page for
- `pageFunction` `<function(Array<Element>)>` 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 `Array.from(document.querySelectorAll(selector))` 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(Element)>` 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.accessibility**

- returns: [<Accessibility>](#)

### **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.
  - `type` [<string>](#) Script type. Use 'module' in order to load a Javascript ES6 module. See [script](#) for more details.
- 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.browser()**

- returns: [<Browser>](#)

Get the browser the page belongs to.

## 🔗 **page.browserContext()**

- returns: [<BrowserContext>](#)

Get the browser context that the page belongs to.

## 🔗 **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 [<"left"|"right"|"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.

Bear 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([options])**

- options [<Object>](#)
  - `runBeforeUnload` [<boolean>](#) Defaults to `false`. Whether to run the [before unload](#) page handlers.
- returns: [<Promise>](#)

By default, `page.close()` **does not** run `beforeunload` handlers.

**NOTE** if `runBeforeUnload` is passed as `true`, a `beforeunload` dialog might be summoned and should be handled manually via page's ['dialog'](#) event.

## **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.
  - `size` [<number>](#)

- httpOnly <[boolean](#)>
- secure <[boolean](#)>
- session <[boolean](#)>
- sameSite <"Strict"|"Lax"|"Extended"|"None">

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](#)>
- 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 [puppeteer.devices](#) .



`page.emulate` will resize the page. A lot of websites don't expect phones to change size, so you should emulate before navigating to the page.

```
const puppeteer = require('puppeteer');
const iPhone = puppeteer.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>>](#) Promise which 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 to the `page.evaluate` returns a non-[Serializable](#) value, then `page.evaluate` resolves to `undefined`. DevTools Protocol also supports transferring some additional values that are not serializable by JSON: `-0`, `NaN`, `Infinity`, `-Infinity`, and bigint literals.

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>>` Promise which resolves to the return value of `pageFunction` as in-page object (JSHandle)

The only difference between `page.evaluate` and `page.evaluateHandle` is that `page.evaluateHandle` returns in-page object (JSHandle).

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.

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

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

[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 the 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
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)` or `page.setDefaultTimeout(timeout)` methods.
  - `waitFor` `<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.goBackward([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)` or `page.setDefaultTimeout(timeout)` methods.
  - `waitFor` `<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 forward, 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)` or `page.setDefaultTimeout(timeout)` methods.
  - `waitFor` `<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.
- `referrer` [<string>](#) Referer header value. If provided it will take preference over the referer header value set by [page.setExtraHTTPHeaders\(\)](#).
- 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.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 remote server does not respond or is unreachable.
- the main resource failed to load.

`page.goto` will not throw an error when any valid HTTP status code is returned by the remote server, including 404 "Not Found" and 500 "Internal Server Error". The status code for such responses can be retrieved by calling [response.status\(\)](#).

**NOTE** `page.goto` either throws an error or returns a main resource response. The only exceptions are navigation to `about:blank` or navigation to the same URL with a different hash, which would succeed and return `null`.

**NOTE** Headless mode doesn't support navigation to a PDF document. See the [upstream issue](#).

Shortcut for [page.mainFrame\(\).goto\(url, options\)](#)

## **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.isClosed()`

- returns: `<boolean>`

Indicates that the page has been closed.

## 🔗 `page.keyboard`

- returns: `<Keyboard>`

## 🔗 `page.mainFrame()`

- returns: `<Frame>` The 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`. Scale amount must be between 0.1 and 2.
  - `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|number>](#) Paper width, accepts values labeled with units.
  - `height` [<string|number>](#) Paper height, accepts values labeled with units.
  - `margin` [<Object>](#) Paper margins, defaults to none.
    - `top` [<string|number>](#) Top margin, accepts values labeled with units.
    - `right` [<string|number>](#) Right margin, accepts values labeled with units.
    - `bottom` [<string|number>](#) Bottom margin, accepts values labeled with units.
    - `left` [<string|number>](#) Left margin, accepts values labeled with units.
  - `preferCSSPageSize` [<boolean>](#) Give any CSS `@page` size declared in the page priority over what is declared in `width` and `height` or `format` options. Defaults to `false`, which will scale the content to fit the paper size.
- 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()`:

**NOTE** By default, `page.pdf()` generates a pdf with modified colors for printing. Use the `-webkit-print-color-adjust` property to force rendering of exact colors.

```
// 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 : 16.54in x 23.4in
- A3 : 11.7in x 16.54in
- A4 : 8.27in x 11.7in
- A5 : 5.83in x 8.27in
- A6 : 4.13in x 5.83in

**NOTE** `headerTemplate` and `footerTemplate` markup have the following limitations:

1. Script tags inside templates are not evaluated.
2. Page styles are not visible inside templates.

## `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 the 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\(timeout\)](#) or [page.setDefaultTimeout\(timeout\)](#) methods.
  - 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`.
- `encoding` `<string>` The encoding of the image, can be either `base64` or `binary`. Defaults to `binary`.
- returns: `<Promise<string|Buffer>>` Promise which resolves to buffer or a base64 string (depending on the value of `encoding`) with captured screenshot.

**NOTE** Screenshots take at least 1/6 second on OS X. See <https://crbug.com/741689> for discussion.

### 🔗 `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>>>` 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.setBypassCSP(enabled)`

- `enabled` `<boolean>` sets bypassing of page's Content-Security-Policy.
- returns: `<Promise>`

Toggles bypassing page's Content-Security-Policy.

**NOTE** CSP bypassing happens at the moment of CSP initialization rather than evaluation. Usually this means that `page.setBypassCSP` should be called before navigating to the domain.

### 🔗 `page.setCacheEnabled([enabled])`

- `enabled` `<boolean>` sets the `enabled` state of the cache.
- returns: `<Promise>`

Toggles ignoring cache for each request based on the enabled state. By default, caching is enabled.

### 🔗 `page.setContent(html[, options])`

- `html` `<string>` HTML markup to assign to the page.
- `options` `<Object>` Parameters which might have the following properties:

- `timeout` [<number>](#) Maximum time in milliseconds for resources to load, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the [`page.setDefaultNavigationTimeout\(timeout\)`](#) or [`page.setDefaultTimeout\(timeout\)`](#) methods.
- `waitFor` [<string|Array<string>>](#) When to consider setting markup succeeded, defaults to `load`. Given an array of event strings, setting content is considered to be successful after all events have been fired. Events can be either:
  - `load` - consider setting content to be finished when the `load` event is fired.
  - `domcontentloaded` - consider setting content to be finished when the `DOMContentLoaded` event is fired.
  - `networkidle0` - consider setting content to be finished when there are no more than 0 network connections for at least 500 ms.
  - `networkidle2` - consider setting content to be finished when there are no more than 2 network connections for at least 500 ms.
- 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` [<"Strict"|"Lax">](#)
- returns: [<Promise>](#)

```
await page.setCookie(cookieObject1, cookieObject2);
```

## [page.setDefaultNavigationTimeout\(timeout\)](#)

- `timeout` [<number>](#) Maximum navigation time in milliseconds

This setting will change the default maximum navigation time for the following methods and related shortcuts:

- [page.goBack\(\[options\]\)](#)
- [page.goForward\(\[options\]\)](#)
- [page.goto\(url\[, options\]\)](#)
- [page.reload\(\[options\]\)](#)
- [page.setContent\(html\[, options\]\)](#)

- `page.waitForNavigation([options])`

**NOTE** `page.setDefaultNavigationTimeout` takes priority over `page.setDefaultTimeout`

## 🔗 `page.setDefaultTimeout(timeout)`

- `timeout` <number> Maximum time in milliseconds

This setting will change the default maximum time for the following methods and related shortcuts:

- `page.goBack([options])`
- `page.goForward([options])`
- `page.goto(url[, options])`
- `page.reload([options])`
- `page.setContent(html[, options])`
- `page.waitFor(selectorOrFunctionOrTimeout[, options[, ...args]])`
- `page.waitForFileChooser([options])`
- `page.waitForFunction(pageFunction[, options[, ...args]])`
- `page.waitForNavigation([options])`
- `page.waitForRequest(urlOrPredicate[, options])`
- `page.waitForResponse(urlOrPredicate[, options])`
- `page.waitForSelector(selector[, options])`
- `page.waitForXPath(xpath[, options])`

**NOTE** `page.setDefaultNavigationTimeout` takes priority over `page.setDefaultTimeout`

## 🔗 `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.setGeolocation(options)`

- `options` <Object>
  - `latitude` <number> Latitude between -90 and 90.
  - `longitude` <number> Longitude between -180 and 180.
  - `accuracy` <number> Optional non-negative accuracy value.
- returns: <Promise>

Sets the page's geolocation.

```
await page.setGeolocation({latitude: 59.95, longitude: 30.31667});
```

**NOTE** Consider using [browserContext.overridePermissions](#) to grant permissions for the page to read its geolocation.

### 🔗 `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. This provides the capability to modify network requests that are made by a page.

Once request interception is enabled, every request will stall unless it's continued, responded or aborted. 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()
        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. **required**
  - `height` `<number>` page height in pixels. **required**
  - `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.setViewport` will resize the page. A lot of websites don't expect phones to change size, so you should set the viewport before navigating to the page.

```
const page = await browser.newPage();
await page.setViewport({
  width: 640,
  height: 480,
  deviceScaleFactor: 1,
});
await page.goto('https://example.com');
```

## 🔗 `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>>` The 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 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`.

## 🔗 `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

```
// wait for selector
await page.waitFor('.foo');
// wait for 1 second
await page.waitFor(1000);
// wait for predicate
await page.waitFor(() => !!document.querySelector('.foo'));
```

To pass arguments from node.js to the predicate of `page.waitFor` function:

```
const selector = '.foo';
await page.waitFor(selector => !!document.querySelector(selector), {}, selector);
```

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

## 🔗 `page.waitForFileChooser([options])`

- `options` <[Object](#)> Optional waiting parameters
  - `timeout` <[number](#)> Maximum wait time in milliseconds, defaults to 30 seconds, pass `0` to disable the timeout. The default value can be changed by using the

[page.setDefaultTimeout\(timeout\)](#) method.

- returns: `<Promise<FileChooser>>` A promise that resolves after a page requests a file picker.

**NOTE** In non-headless Chromium, this method results in the native file picker dialog **not showing up** for the user.

This method is typically coupled with an action that triggers file choosing. The following example clicks a button that issues a file chooser, and then responds with `/tmp/myfile.pdf` as if a user has selected this file.

```
const [fileChooser] = await Promise.all([
  page.waitForFileChooser(),
  page.click('#upload-file-button'), // some button that triggers file select:
]);
await fileChooser.accept(['/tmp/myfile.pdf']);
```

**NOTE** This must be called *before* the file chooser is launched. It will not return a currently active file chooser.

## 🔗 `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). Pass `0` to disable timeout. The default value can be changed by using the [page.setDefaultTimeout\(timeout\)](#) method.
- `...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');
  await page.setViewport({width: 50, height: 50});
  await watchDog;
```

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

To pass arguments from node.js to the predicate of `page.waitForFunction` function:

```
const selector = '.foo';
await page.waitForFunction(selector => !!document.querySelector(selector), {},
```

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)` or `page.setDefaultTimeout(timeout)` methods.
  - `waitFor` <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. In case of navigation to a different anchor or navigation due to History API usage, the navigation will resolve with `null`.

This resolves when the page navigates to a new URL or reloads. It is useful for when you run code which will indirectly cause the page to navigate. Consider this example:

```
const [response] = await Promise.all([
  page.waitForNavigation(), // The promise resolves after navigation has finished
  page.click('a.my-link'), // Clicking the link will indirectly cause a navigation
]);
```

**NOTE** Usage of the [History API](#) to change the URL is considered a navigation.

Shortcut for `page.mainFrame().waitForNavigation(options)`.

## 🔗 `page.waitForRequest(urlOrPredicate[, options])`

- `urlOrPredicate` <string|Function> A URL or predicate to wait for.

- options <Object> Optional waiting parameters
  - timeout <number> Maximum wait time in milliseconds, defaults to 30 seconds, pass 0 to disable the timeout. The default value can be changed by using the `page.setDefaultTimeout(timeout)` method.
- returns: <Promise<Request>> Promise which resolves to the matched request.

```
const firstRequest = await page.waitForRequest('http://example.com/resource'),
const finalRequest = await page.waitForRequest(request => request.url() === 'http://example.com/resource'),
return firstRequest.url();
```

## 🔗 `page.waitForResponse(urlOrPredicate[, options])`

- urlOrPredicate <string|Function> A URL or predicate to wait for.
- options <Object> Optional waiting parameters
  - timeout <number> Maximum wait time in milliseconds, defaults to 30 seconds, pass 0 to disable the timeout. The default value can be changed by using the `page.setDefaultTimeout(timeout)` method.
- returns: <Promise<Response>> Promise which resolves to the matched response.

```
const firstResponse = await page.waitForResponse('https://example.com/resource'),
const finalResponse = await page.waitForResponse(response => response.url() === 'https://example.com/resource'),
return finalResponse.ok();
```

## 🔗 `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). Pass 0 to disable timeout. The default value can be changed by using the `page.setDefaultTimeout(timeout)` method.
- returns: <Promise<?ElementHandle>> Promise which resolves when element specified by selector string is added to DOM. Resolves to `null` if waiting for `hidden: true` and selector is not found in 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://bl
    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). Pass `0` to disable timeout. The default value can be changed by using the `page.setDefaultTimeout(timeout)` method.
- `returns`: <Promise<?ElementHandle>> Promise which resolves when element specified by `xpath` string is added to DOM. Resolves to `null` if waiting for `hidden: true` and `xpath` is not found in 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://bl
    await page.goto(currentURL);
  await browser.close();
});
```

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

## 🔗 `page.workers()`

- returns: `<Array<Worker>>` This method returns all of the dedicated [WebWorkers](#) associated with the page.

**NOTE** This does not contain ServiceWorkers

## 🔗 **class: Worker**

The `Worker` class represents a [WebWorker](#). The events `workercreated` and `workerdestroyed` are emitted on the page object to signal the worker lifecycle.

```
page.on('workercreated', worker => console.log('Worker created: ' + worker.url));
page.on('workerdestroyed', worker => console.log('Worker destroyed: ' + worker.url));

console.log('Current workers:');
for (const worker of page.workers())
  console.log('  ' + worker.url());
```

## 🔗 `worker.evaluate(pageFunction[, ...args])`

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

If the function passed to the `worker.evaluate` returns a [Promise](#), then `worker.evaluate` would wait for the promise to resolve and return its value.

If the function passed to the `worker.evaluate` returns a non-[Serializable](#) value, then `worker.evaluate` resolves to `undefined`. DevTools Protocol also supports transferring some additional values that are not serializable by JSON: `-0`, `NaN`, `Infinity`, `-Infinity`, and bigint literals.

Shortcut for `(await worker.executionContext()).evaluate(pageFunction, ...args)`.

## 🔗 `worker.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>>` Promise which resolves to the return value of `pageFunction` as in-page object (`JSHandle`)

The only difference between `worker.evaluate` and `worker.evaluateHandle` is that `worker.evaluateHandle` returns in-page object (`JSHandle`).

If the function passed to the `worker.evaluateHandle` returns a [Promise](#), then `worker.evaluateHandle` would wait for the promise to resolve and return its value.

Shortcut for `(await worker.executionContext()).evaluateHandle(pageFunction, ...args)`.

## 🔗 `worker.executionContext()`

- returns: `<Promise<ExecutionContext>>`

## 🔗 `worker.url()`

- returns: `<string>`

## 🔗 **class: Accessibility**

The Accessibility class provides methods for inspecting Chromium's accessibility tree. The accessibility tree is used by assistive technology such as [screen readers](#) or [switches](#).

Accessibility is a very platform-specific thing. On different platforms, there are different screen readers that might have wildly different output.

Blink - Chrome's rendering engine - has a concept of "accessibility tree", which is then translated into different platform-specific APIs. Accessibility namespace gives users access to the Blink Accessibility Tree.

Most of the accessibility tree gets filtered out when converting from Blink AX Tree to Platform-specific AX-Tree or by assistive technologies themselves. By default, Puppeteer tries to approximate this filtering, exposing only the "interesting" nodes of the tree.

## 🔗 `accessibility.snapshot([options])`

- options `<Object>`
  - `interestingOnly` `<boolean>` Prune uninteresting nodes from the tree. Defaults to `true`.
  - `root` `<ElementHandle>` The root DOM element for the snapshot. Defaults to the whole page.
- returns: `<Promise<Object>>` An [AXNode](#) object with the following properties:
  - `role` `<string>` The [role](#).
  - `name` `<string>` A human readable name for the node.
  - `value` `<string|number>` The current value of the node.
  - `description` `<string>` An additional human readable description of the node.
  - `keyshortcuts` `<string>` Keyboard shortcuts associated with this node.
  - `roledescription` `<string>` A human readable alternative to the role.
  - `valuetext` `<string>` A description of the current value.
  - `disabled` `<boolean>` Whether the node is disabled.
  - `expanded` `<boolean>` Whether the node is expanded or collapsed.
  - `focused` `<boolean>` Whether the node is focused.
  - `modal` `<boolean>` Whether the node is [modal](#).
  - `multiline` `<boolean>` Whether the node text input supports multiline.
  - `multiselectable` `<boolean>` Whether more than one child can be selected.



- `readonly` `<boolean>` Whether the node is read only.
- `required` `<boolean>` Whether the node is required.
- `selected` `<boolean>` Whether the node is selected in its parent node.
- `checked` `<boolean|"mixed">` Whether the checkbox is checked, or "mixed".
- `pressed` `<boolean|"mixed">` Whether the toggle button is checked, or "mixed".
- `level` `<number>` The level of a heading.
- `valuemin` `<number>` The minimum value in a node.
- `valuemax` `<number>` The maximum value in a node.
- `autocomplete` `<string>` What kind of autocomplete is supported by a control.
- `haspopup` `<string>` What kind of popup is currently being shown for a node.
- `invalid` `<string>` Whether and in what way this node's value is invalid.
- `orientation` `<string>` Whether the node is oriented horizontally or vertically.
- `children` `<Array<Object>>` Child `AXNodes` of this node, if any.

Captures the current state of the accessibility tree. The returned object represents the root accessible node of the page.

**NOTE** The Chromium accessibility tree contains nodes that go unused on most platforms and by most screen readers. Puppeteer will discard them as well for an easier to process tree, unless `interestingOnly` is set to `false`.

An example of dumping the entire accessibility tree:

```
const snapshot = await page.accessibility.snapshot();
console.log(snapshot);
```

An example of logging the focused node's name:

```
const snapshot = await page.accessibility.snapshot();
const node = findFocusedNode(snapshot);
console.log(node && node.name);

function findFocusedNode(node) {
  if (node.focused)
    return node;
  for (const child of node.children || []) {
    const foundNode = findFocusedNode(child);
    return foundNode;
  }
  return null;
}
```

## 🔗 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 be 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 `keyboard.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**

The `Mouse` class operates in main-frame CSS pixels relative to the top-left corner of the viewport.

Every `page` object has its own `Mouse`, accessible with `page.mouse` .

```
// Using 'page.mouse' to trace a 100x100 square.
await page.mouse.move(0, 0);
await page.mouse.down();
await page.mouse.move(0, 100);
await page.mouse.move(100, 100);
await page.mouse.move(100, 0);
await page.mouse.move(0, 0);
await page.mouse.up();
```

## 🔗 **mouse.click(x, y[, options])**

- x `<number>`
- y `<number>`
- options `<Object>`
  - button `<"left"|"right"|"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 `<"left"|"right"|"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 [<"left"|"right"|"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.
  - 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<Buffer>>](#) Promise which resolves to buffer with trace data.

## 🔗 class: FileChooser

`FileChooser` objects are returned via the `'page.waitForFileChooser'` method.

File choosers let you react to the page requesting for a file.

An example of using `FileChooser`:

```
const [fileChooser] = await Promise.all([
  page.waitForFileChooser(),
  page.click('#upload-file-button'), // some button that triggers file select:
]);
await fileChooser.accept(['/tmp/myfile.pdf']);
```

**NOTE** In browsers, only one file chooser can be opened at a time. All file choosers must be accepted or canceled. Not doing so will prevent subsequent file choosers from appearing.

### 🔗 fileChooser.accept(filePaths)

- `filePaths` `<Array<string>>` Accept the file chooser request with given paths. If some of the `filePaths` are relative paths, then they are resolved relative to the `current working directory`.
- returns: `<Promise>`

### 🔗 fileChooser.cancel()

- returns: `<Promise>`

Closes the file chooser without selecting any files.

### 🔗 fileChooser.isMultiple()

- returns: `<boolean>` Whether file chooser allow for `multiple` file selection.

## 🔗 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.location()**

- returns: <Object>
  - `url` <string> URL of the resource if known or `undefined` otherwise.
  - `lineNumber` <number> 0-based line number in the resource if known or `undefined` otherwise.
  - `columnNumber` <number> 0-based column number in the resource if known or `undefined` otherwise.

### 🔗 **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 + ' ');
  }
});
```

An example of getting text from an iframe element:

```
const frame = page.frames().find(frame => frame.name() === 'myframe');
const text = await frame.$eval('.selector', element => element.textContent);
console.log(text);
```

## 🔗 frame.\$(selector)

- selector [<string>](#) A [selector](#) to query frame 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>` A [selector](#) to query frame 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 resolves to `[]`.

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

- selector `<string>` A [selector](#) to query frame for
- pageFunction `<function(Array<Element>>>` 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 `Array.from(document.querySelectorAll(selector))` 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(Element)>` 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.
  - type [<string>](#) Script type. Use 'module' in order to load a Javascript ES6 module. See [script](#) for more details.
- 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 [<"left"|"right"|"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.

Bear 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 the return value of `pageFunction`

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 to the `frame.evaluate` returns a non-`Serializable` value, then `frame.evaluate` resolves to `undefined`. DevTools Protocol also supports transferring some additional values that are not serializable by JSON: `-0`, `NaN`, `Infinity`, `-Infinity`, and bigint literals.

```
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>>](#) Promise which resolves to the return value of `pageFunction` as in-page object (JSHandle)

The only difference between `frame.evaluate` and `frame.evaluateHandle` is that `frame.evaluateHandle` returns in-page object (JSHandle).

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' object.
```

[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>>](#)

Returns promise that resolves to the frame's default execution context.

### **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.goto(url[, options])`

- `url` [<string>](#) URL to navigate frame 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\(timeout\)](#) or [page.setDefaultTimeout\(timeout\)](#) methods.
  - `waitFor` [<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.
  - `referrer` [<string>](#) Referrer header value. If provided it will take preference over the referer header value set by [page.setExtraHTTPHeaders\(\)](#).
- `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.

`frame.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 remote server does not respond or is unreachable.
- the main resource failed to load.

`frame.goto` will not throw an error when any valid HTTP status code is returned by the remote server, including 404 "Not Found" and 500 "Internal Server Error". The status code for such responses can be retrieved by calling [response.status\(\)](#).

**NOTE** `frame.goto` either throws an error or returns a main resource response. The only exceptions are navigation to `about:blank` or navigation to the same URL with a different hash, which would succeed and return `null`.

**NOTE** Headless mode doesn't support navigation to a PDF document. See the [upstream issue](#).

## 🔗 `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>` 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>>>` 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[, options])`

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

- options `<Object>` Parameters which might have the following properties:
  - timeout `<number>` Maximum time in milliseconds for resources to load, defaults to 30 seconds, pass `0` to disable timeout. The default value can be changed by using the `page.setDefaultNavigationTimeout(timeout)` or `page.setDefaultTimeout(timeout)` methods.
  - waitUntil `<string|Array<string>>` When to consider setting markup succeeded, defaults to `load`. Given an array of event strings, setting content is considered to be successful after all events have been fired. Events can be either:
    - `load` - consider setting content to be finished when the `load` event is fired.
    - `domcontentloaded` - consider setting content to be finished when the `DOMContentLoaded` event is fired.
    - `networkidle0` - consider setting content to be finished when there are no more than 0 network connections for at least 500 ms.
    - `networkidle2` - consider setting content to be finished when there are no more than 2 network connections for at least 500 ms.
- 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>>` The 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

```
// wait for selector
await page.waitFor('.foo');
// wait for 1 second
await page.waitFor(1000);
// wait for predicate
await page.waitFor(() => !!document.querySelector('.foo'));
```

To pass arguments from node.js to the predicate of `page.waitFor` function:

```
const selector = '.foo';
await page.waitFor(selector => !!document.querySelector(selector), {}, selector);
```

## 🔗 `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). Pass `0` to disable timeout. The default value can be changed by using the `page.setDefaultTimeout(timeout)` method.
- `...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();
});
```

To pass arguments from node.js to the predicate of `page.waitForFunction` function:

```
const selector = '.foo';
await page.waitForFunction(selector => !!document.querySelector(selector), {},
```

## 🔗 `frame.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)` or `page.setDefaultTimeout(timeout)` methods.
  - `waitFor` `<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. In case of navigation to a different anchor or navigation due to History API usage, the navigation will resolve with `null`.

This resolves when the frame navigates to a new URL. It is useful for when you run code which will indirectly cause the frame to navigate. Consider this example:

```
const [response] = await Promise.all([
  frame.waitForNavigation(), // The navigation promise resolves after navigat:
  frame.click('a.my-link'), // Clicking the link will indirectly cause a navi
]);
```

**NOTE** Usage of the [History API](#) to change the URL is considered a navigation.

### 🔗 `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). Pass `0` to disable timeout. The default value can be changed by using the [page.setDefaultTimeout\(timeout\)](#) method.
- returns: `<Promise<?ElementHandle>>` Promise which resolves when element specified by selector string is added to DOM. Resolves to `null` if waiting for `hidden: true` and selector is not found in 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://bl
    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). Pass `0` to disable timeout. The default value can be changed by using the `page.setDefaultTimeout(timeout)` method.
- `returns`: <Promise<?ElementHandle>> Promise which resolves when element specified by `xpath` string is added to DOM. Resolves to `null` if waiting for `hidden: true` and `xpath` is not found in 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://bl
    await page.goto(currentURL);
  await browser.close();
});
```

## 🔗 `class: ExecutionContext`

The class represents a context for JavaScript execution. A `Page` might have many execution contexts:

- each `frame` has "default" execution context that is always created after frame is attached to DOM. This context is returned by the `frame.executionContext()` method.
- `Extensions`'s content scripts create additional execution contexts.

Besides pages, execution contexts can be found in [workers](#).

## 🔗 `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 the return value of `pageFunction`

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.

If the function passed to the `executionContext.evaluate` returns a non-[Serializable](#) value, then `executionContext.evaluate` resolves to `undefined`. DevTools Protocol also supports transferring some additional values that are not serializable by JSON: `-0`, `NaN`, `Infinity`, `-Infinity`, and bigint literals.

```
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](#)>> Promise which resolves to the return value of `pageFunction` as in-page object ([JSHandle](#))

The only difference between `executionContext.evaluate` and `executionContext.evaluateHandle` is that `executionContext.evaluateHandle` returns in-page object ([JSHandle](#)).

If the function passed to the `executionContext.evaluateHandle` returns a [Promise](#), then `executionContext.evaluateHandle` 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' (
```

**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.frame()**

- returns: `<?Frame>` Frame associated with this execution context.

**NOTE** Not every execution context is associated with a frame. For example, workers and extensions have execution contexts that are not associated with frames.

### **executionContext.queryObjects(prototypeHandle)**

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

The method iterates the 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 the referenced JavaScript object being garbage collected 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.evaluate\(pageFunction\[, ...args\]\)](#)

- `pageFunction` [<function\(Object\)>](#) 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 passes this handle as the first argument to `pageFunction` .

If `pageFunction` returns a [Promise](#), then `handle.evaluate` would wait for the promise to resolve and return its value.

Examples:

```
const tweetHandle = await page.$('.tweet .retweets');  
expect(await tweetHandle.evaluate(node => node.innerText)).toBe('10');
```

### [jsHandle.evaluateHandle\(pageFunction\[, ...args\]\)](#)

- `pageFunction` [<function|string>](#) Function to be evaluated
- `...args` [<...Serializable|JSHandle>](#) Arguments to pass to `pageFunction`
- returns: [<Promise<JSHandle>>](#) Promise which resolves to the return value of `pageFunction` as in-page object (JSHandle)

This method passes this handle as the first argument to `pageFunction` .

The only difference between `jsHandle.evaluate` and `jsHandle.evaluateHandle` is that `executionContext.evaluateHandle` returns in-page object (JSHandle).

If the function passed to the `jsHandle.evaluateHandle` returns a [Promise](#), then `jsHandle.evaluateHandle` would wait for the promise to resolve and return its value.

See [Page.evaluateHandle](#) for more details.

### `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 object if the referenced object is not stringifiable. It will throw an error if the object has circular references.

### **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://example.com');
  const hrefElement = await page.$('a');
  await hrefElement.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 resolves 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 resolves to `[]`.

### 🔗 **elementHandle.\$\$eval(selector, pageFunction[, ...args])**

- selector `<string>` A [selector](#) to query page for
- pageFunction `<function(Array<Element>>>` 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 element 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:



```
<div class="feed">
  <div class="tweet">Hello!</div>
  <div class="tweet">Hi!</div>
</div>
```

```
const feedHandle = await page.$('.feed');
expect(await feedHandle.$$eval('.tweet', nodes => nodes.map(n => n.innerText))
```

## 🔗 `elementHandle.$eval(selector, pageFunction[, ...args])`

- selector `<string>` A `selector` to query page for
- pageFunction `<function(Element)>` 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 element 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 tweetHandle = await page.$('.tweet');
expect(await tweetHandle.$eval('.like', node => node.innerText)).toBe('100');
expect(await tweetHandle.$eval('.retweets', node => node.innerText)).toBe('10
```

## 🔗 `elementHandle.$x(expression)`

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

The method evaluates the XPath expression relative to the `elementHandle`. If there are no such elements, the method will resolve to an empty array.

## 🔗 `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.boxModel()**

- returns: `<Promise<?Object>>`
  - content `<Array<Object>>` Content box.
    - x `<number>`
    - y `<number>`
  - padding `<Array<Object>>` Padding box.
    - x `<number>`
    - y `<number>`
  - border `<Array<Object>>` Border box.
    - x `<number>`
    - y `<number>`
  - margin `<Array<Object>>` Margin box.
    - x `<number>`
    - y `<number>`
  - width `<number>` Element's width.
  - height `<number>` Element's height.

This method returns boxes of the element, or `null` if the element is not visible. Boxes are represented as an array of points; each Point is an object `{x, y}`. Box points are sorted clock-wise.

### **elementHandle.click([options])**

- options `<Object>`
  - button `<"left"|"right"|"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.contentFrame()**

- returns: `<Promise<?Frame>>` Resolves to the content frame for element handles referencing iframe nodes, or null otherwise

### **elementHandle.dispose()**

- returns: [<Promise>](#) Promise which resolves when the element handle is successfully disposed.

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

### `elementHandle.evaluate(pageFunction[, ...args])`

- `pageFunction` [<function\(Object\)>](#) 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 passes this handle as the first argument to `pageFunction`.

If `pageFunction` returns a [Promise](#), then `handle.evaluate` would wait for the promise to resolve and return its value.

Examples:

```
const tweetHandle = await page.$('.tweet .retweets');
expect(await tweetHandle.evaluate(node => node.innerText)).toBe('10');
```

### `elementHandle.evaluateHandle(pageFunction[, ...args])`

- `pageFunction` [<function|string>](#) Function to be evaluated
- `...args` [<...Serializable|JSHandle>](#) Arguments to pass to `pageFunction`
- returns: [<Promise<JSHandle>>](#) Promise which resolves to the return value of `pageFunction` as in-page object (JSHandle)

This method passes this handle as the first argument to `pageFunction`.

The only difference between `evaluateHandle.evaluate` and `evaluateHandle.evaluateHandle` is that `executionContext.evaluateHandle` returns in-page object (JSHandle).

If the function passed to the `evaluateHandle.evaluateHandle` returns a [Promise](#), then `evaluateHandle.evaluateHandle` would wait for the promise to resolve and return its value.

See [Page.evaluateHandle](#) for more details.

### `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.isIntersectingViewport()`

- returns: `<Promise<boolean>>` Resolves to true if the element is visible in the current viewport.

## 🔗 `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<string|Buffer>>](#) Promise which resolves to buffer or a base64 string (depending on the value of `options.encoding` ) 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.select(...values)`

- `...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>>>](#) 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.

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

## 🔗 `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 to these paths. If some of the `filePaths` are relative paths, then they are resolved relative to the [current working directory](#).
- returns: <Promise>

This method expects `elementHandle` to point to an [input element](#).

## 🔗 **class: Request**

Whenever the page sends a request, such as for a network resource, 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.

**NOTE** HTTP Error responses, such as 404 or 503, are still successful responses from HTTP standpoint, so request will complete with `'requestfinished'` event.

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.
  - `blockedbyclient` - The client chose to block the request.
  - `blockedbyresponse` - The request failed because the response was delivered along with requirements which are not met ('X-Frame-Options' and 'Content-Security-Policy' ancestor checks, for instance).
  - `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. This is not a redirect. The request will be silently forwarded to the new url. For example, the address bar will show the original url.
  - `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. Header values will be converted to a string.
- 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.

```
await page.setRequestInterception(true);
page.on('request', request => {
  // Override headers
  const headers = Object.assign({}, request.headers(), {
    foo: 'bar', // set "foo" header
    origin: undefined, // remove "origin" header
  });
  request.continue({headers});
});
```

### `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 `Frame` that initiated this request, 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.isNavigationRequest()`

- returns: `<boolean>`

Whether this request is driving frame's navigation.

### `request.method()`

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

### `request.postData()`

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

## 🔗 `request.redirectChain()`

- returns: `<Array<Request>>`

A `redirectChain` is a chain of requests initiated to fetch a resource.

- If there are no redirects and the request was successful, the chain will be empty.
- If a server responds with at least a single redirect, then the chain will contain all the requests that were redirected.

`redirectChain` is shared between all the requests of the same chain.

For example, if the website `http://example.com` has a single redirect to `https://example.com`, then the chain will contain one request:

```
const response = await page.goto('http://example.com');
const chain = response.request().redirectChain();
console.log(chain.length); // 1
console.log(chain[0].url()); // 'http://example.com'
```

If the website `https://google.com` has no redirects, then the chain will be empty:

```
const response = await page.goto('https://google.com');
const chain = response.request().redirectChain();
console.log(chain.length); // 0
```

## 🔗 `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`, `eventsources`, `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. Header values will be converted to a string.
  - `contentType` `<string>` If set, equals to setting `Content-Type` response header
  - `body` `<string|Buffer>` 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.frame()`

- returns: `<?Frame>` A `Frame` that initiated this response, or `null` if navigating to error pages.

### 🔗 `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.remoteAddress()**

- returns: `<Object>`
  - `ip` `<string>` the IP address of the remote server
  - `port` `<number>` the port used to connect to the remote server

## 🔗 **response.request()**

- returns: `<Request>` A matching `Request` object.

## 🔗 **response.securityDetails()**

- returns: `<?SecurityDetails>` Security details if the response was received over the secure connection, or `null` otherwise.

## 🔗 **response.status()**

- returns: `<number>`

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

## 🔗 **response.statusText()**

- returns: `<string>`

Contains the status text of the response (e.g. usually an "OK" 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: SecurityDetails**

[SecurityDetails](#) class represents the security details when response was received over the secure connection.

### 🔗 **securityDetails.issuer()**

- returns: [<string>](#) A string with the name of issuer of the certificate.

### 🔗 **securityDetails.protocol()**

- returns: [<string>](#) String with the security protocol, eg. "TLS 1.2".

### 🔗 **securityDetails.subjectName()**

- returns: [<string>](#) Name of the subject to which the certificate was issued to.

### 🔗 **securityDetails.validFrom()**

- returns: [<number>](#) [UnixTime](#) stating the start of validity of the certificate.

### 🔗 **securityDetails.validTo()**

- returns: [<number>](#) [UnixTime](#) stating the end of validity of the certificate.

## 🔗 **class: Target**

### 🔗 **target.browser()**

- returns: [<Browser>](#)

Get the browser the target belongs to.

### 🔗 **target.browserContext()**

- returns: [<BrowserContext>](#)

The browser context the target belongs to.

### 🔗 **target.createCDPSession()**

- returns: [<Promise<CDPSession>>](#)

Creates a Chrome Devtools Protocol session attached to the target.

### 🔗 **target.opener()**

- returns: [<?Target>](#)

Get the target that opened this target. Top-level targets return `null`.

### 🔗 **target.page()**

- returns: `<Promise<?Page>>`

If the target is not of type "page" or "background\_page", returns null.

### **target.type()**

- returns:  
`<"page"|"background_page"|"service_worker"|"shared_worker"|"other"|"browser">`

Identifies what kind of target this is. Can be "page", "["background\\_page"](#)", "service\_worker", "shared\_worker", "browser" or "other".

### **target.url()**

- returns: `<string>`

### **target.worker()**

- returns: `<Promise<?Worker>>`

If the target is not of type "service\_worker" or "shared\_worker", returns null.

### **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.

Useful links:

- Documentation on DevTools Protocol can be found here: [DevTools Protocol Viewer](#).
- Getting Started with DevTools Protocol: <https://github.com/aslushnikov/getting-started-with-cdp/blob/master/README.md>

```
const client = await page.target().createCDPSession();
await client.send('Animation.enable');
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 the cdpSession from the target. Once detached, the cdpSession object 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}%`);
```

To output coverage in a form consumable by [Istanbul](#), see [puppeteer-to-istanbul](#).

## `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`.

- `reportAnonymousScripts` [<boolean>](#) Whether anonymous scripts generated by the page should be reported. Defaults to `false`.
- returns: [<Promise>](#) Promise that resolves when coverage is started

**NOTE** Anonymous scripts are ones that don't have an associated url. These are scripts that are dynamically created on the page using `eval` or `new Function`. If `reportAnonymousScripts` is set to `true`, anonymous scripts will have `__puppeteer_evaluation_script__` as their URL.

## 🔗 `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 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 by default. However, scripts with sourceURLs are reported.

## 🔗 **class: TimeoutError**

- extends: [Error](#)

TimeoutError is emitted whenever certain operations are terminated due to timeout, e.g. [page.waitForSelector\(selector\[, options\]\)](#) or [puppeteer.launch\(\[options\]\)](#).