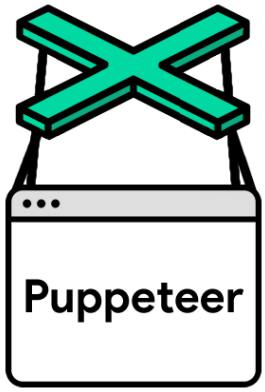
Puppeteer

API | **FAQ** | **CONTRIBUTING** | **TROUBLESHOOTING**



> *Puppeteer is a Node library which provides a high-level API to control Chrome or Chromium over the DevTools Protocol. Puppeteer runs headless by default, but can be configured to run full (non-headless) Chrome or Chromium.*

WHAT CAN I DO?

Most things that you can do manually in the browser can be done using Puppeteer! Here are a few examples to get you started:

- Generate screenshots and PDFs of pages.
- Crawl a SPA (Single-Page Application) and generate pre-rendered content (i.e. "SSR" (Server-Side Rendering)).
- Automate form submission, UI testing, keyboard input, etc.
- Create an up-to-date, automated testing environment. Run your tests directly in the latest version of Chrome using the latest JavaScript and browser features.
- Capture a timeline trace of your site to help diagnose performance issues.
- Test Chrome Extensions.

Give it a spin: https://try-puppeteer.appspot.com/

Getting Started

Installation

To use Puppeteer in your project, run:

```
npm i puppeteer
# or "yarn add puppeteer"
```

Note: When you install Puppeteer, it downloads a recent version of Chromium (~170MB Mac, ~282MB Linux, ~280MB Win) that is guaranteed to work with the API. To skip the download, or to download a different browser, see Environment variables.

puppeteer-core

Since version 1.7.0 we publish the `puppeteer-core` package, a version of Puppeteer that doesn't download any browser by default.

```
npm i puppeteer-core
# or "yarn add puppeteer-core"
```

`puppeteer-core` is intended to be a lightweight version of Puppeteer for launching an existing browser installation or for connecting to a remote one. Be sure that the version of puppeteer-core you install is compatible with the browser you intend to connect to.

See puppeteer vs puppeteer-core.

Usage

Puppeteer follows the latest maintenance LTS version of Node.

Note: Prior to v1.18.1, Puppeteer required at least Node v6.4.0. Versions from v1.18.1 to v2.1.0 rely on Node 8.9.0+. Starting from v3.0.0 Puppeteer starts to rely on Node 10.18.1+. All examples below use async/await which is only supported in Node v7.6.0 or greater.

Puppeteer will be familiar to people using other browser testing frameworks. You create an instance of `Browser`, open pages, and then manipulate them with Puppeteer's API.

Example - navigating to https://example.com and saving a screenshot as *example.png*:

Save file as **example.js**

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

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  await page.screenshot({ path: 'example.png' });

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

Execute script on the command line

```
node example.js
```

Puppeteer sets an initial page size to 800×600px, which defines the screenshot size. The page size can be customized with `Page.setViewport()`.

Example - create a PDF.

Save file as **hn.js**

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

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('https://news.ycombinator.com', {
    waitUntil: 'networkidle2',
  });
  await page.pdf({ path: 'hn.pdf', format: 'a4' });

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

Execute script on the command line

```
node hn.js
```

See `Page.pdf()` for more information about creating pdfs.

Example - evaluate script in the context of the page

Save file as **get-dimensions.js**

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

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

  // Get the "viewport" of the page, as reported by the page.
  const dimensions = await page.evaluate(() => {
    return {
      width: document.documentElement.clientWidth,
      height: document.documentElement.clientHeight,
      deviceScaleFactor: window.devicePixelRatio,
    };
  });

  console.log('Dimensions:', dimensions);

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

Execute script on the command line

```
node get-dimensions.js
```

See `Page.evaluate()` for more information on `evaluate` and related methods like `evaluateOnNewDocument` and `exposeFunction`.

Default runtime settings

1. Uses Headless mode

Puppeteer launches Chromium in headless mode. To launch a full version of Chromium, set the `headless` option when launching a browser:

```
const browser = await puppeteer.launch({ headless: false }); // default is true
```

2. Runs a bundled version of Chromium

By default, Puppeteer downloads and uses a specific version of Chromium so its API is guaranteed to work out of the box. To use Puppeteer with a different version of Chrome or Chromium, pass in the executable's path when creating a `Browser` instance:

```
const browser = await puppeteer.launch({ executablePath: '/path/to/Chrome' });
```

You can also use Puppeteer with Firefox Nightly (experimental support). See `Puppeteer.launch()` for more information.

See `this article` for a description of the differences between Chromium and Chrome. `This article` describes some differences for Linux users.

3. Creates a fresh user profile

Puppeteer creates its own browser user profile which it **cleans up on every run**.

Resources

- API Documentation
- Examples
- Community list of Puppeteer resources

Debugging tips

1. Turn off headless mode - sometimes it's useful to see what the browser is displaying. Instead of launching in headless mode, launch a full version of the browser using `headless: false`:

   ```
   const browser = await puppeteer.launch({ headless: false });
   ```

2. Slow it down - the `slowMo` option slows down Puppeteer operations by the specified amount of milliseconds. It's another way to help see what's going on.

   ```
   const browser = await puppeteer.launch({
     headless: false,
     slowMo: 250, // slow down by 250ms
   });
   ```

3. Capture console output - You can listen for the `console` event. This is also handy when debugging code in `page.evaluate()`:

   ```
   page.on('console', (msg) => console.log('PAGE LOG:', msg.text()));

   await page.evaluate(() => console.log(`url is ${location.href}`));
   ```

4. Use debugger in application code browser

   There are two execution context: node.js that is running test code, and the browser running application code being tested. This lets you debug code in the application code browser; ie code inside `evaluate()`.

   - Use `{devtools: true}` when launching Puppeteer:

     ```
     const browser = await puppeteer.launch({ devtools: true });
     ```

   - Change default test timeout:

jest: `jest.setTimeout(100000);`

jasmine: `jasmine.DEFAULT_TIMEOUT_INTERVAL = 100000;`

mocha: `this.timeout(100000);` (don't forget to change test to use function and not '=>')

- Add an evaluate statement with `debugger` inside / add `debugger` to an existing evaluate statement:

```
await page.evaluate(() => {
  debugger;
});
```

The test will now stop executing in the above evaluate statement, and chromium will stop in debug mode.

5. Use debugger in node.js

This will let you debug test code. For example, you can step over `await page.click()` in the node.js script and see the click happen in the application code browser.

Note that you won't be able to run `await page.click()` in DevTools console due to this Chromium bug. So if you want to try something out, you have to add it to your test file.

- Add `debugger;` to your test, eg:

```
debugger;
await page.click('a[target=_blank]');
```

- Set `headless` to `false`
- Run `node --inspect-brk`, eg `node --inspect-brk node_modules/.bin/jest tests`
- In Chrome open `chrome://inspect/#devices` and click `inspect`
- In the newly opened test browser, type `F8` to resume test execution
- Now your `debugger` will be hit and you can debug in the test browser

6. Enable verbose logging - internal DevTools protocol traffic will be logged via the `debug` module under the `puppeteer` namespace.

```
# Basic verbose logging
env DEBUG="puppeteer:*" node script.js

# Protocol traffic can be rather noisy. This example filters out all Network domain messages
env DEBUG="puppeteer:*" env DEBUG_COLORS=true node script.js 2>&1 | grep -v '"Network'
```

7. Debug your Puppeteer (node) code easily, using ndb

- `npm install -g ndb` (or even better, use npx!)
- add a `debugger` to your Puppeteer (node) code
- add `ndb` (or `npx ndb`) before your test command. For example:

  `ndb jest` or `ndb mocha` (or `npx ndb jest` / `npx ndb mocha`)
- debug your test inside chromium like a boss!

Usage with TypeScript

We have recently completed a migration to move the Puppeteer source code from JavaScript to TypeScript and as of version 7.0.1 we ship our own built-in type definitions.

If you are on a version older than 7, we recommend installing the Puppeteer type definitions from the DefinitelyTyped repository:

```
npm install --save-dev @types/puppeteer
```

The types that you'll see appearing in the Puppeteer source code are based off the great work of those who have contributed to the `@types/puppeteer` package. We really appreciate the hard work those people put in to providing high quality TypeScript definitions for Puppeteer's users.

Contributing to Puppeteer

Check out contributing guide to get an overview of Puppeteer development.

FAQ

Q: Who maintains Puppeteer?

The Chrome DevTools team maintains the library, but we'd love your help and expertise on the project! See Contributing.

Q: What is the status of cross-browser support?

Official Firefox support is currently experimental. The ongoing collaboration with Mozilla aims to support common end-to-end testing use cases, for which developers expect cross-browser coverage. The Puppeteer team needs input from users to stabilize Firefox support and to bring missing APIs to our attention.

From Puppeteer v2.1.0 onwards you can specify `puppeteer.launch({product: 'firefox'})` to run your Puppeteer scripts in Firefox Nightly, without any additional custom patches. While an older experiment required a patched version of Firefox, the current approach works with “stock” Firefox.

We will continue to collaborate with other browser vendors to bring Puppeteer support to browsers such as Safari. This effort includes exploration of a standard for executing cross-browser commands (instead of relying on the non-standard DevTools Protocol used by Chrome).

Q: What are Puppeteer’s goals and principles?

The goals of the project are:

- Provide a slim, canonical library that highlights the capabilities of the DevTools Protocol.
- Provide a reference implementation for similar testing libraries. Eventually, these other frameworks could adopt Puppeteer as their foundational layer.
- Grow the adoption of headless/automated browser testing.
- Help dogfood new DevTools Protocol features...and catch bugs!
- Learn more about the pain points of automated browser testing and help fill those gaps.

We adapt Chromium principles to help us drive product decisions:

- **Speed**: Puppeteer has almost zero performance overhead over an automated page.
- **Security**: Puppeteer operates off-process with respect to Chromium, making it safe to automate potentially malicious pages.
- **Stability**: Puppeteer should not be flaky and should not leak memory.
- **Simplicity**: Puppeteer provides a high-level API that's easy to use, understand, and debug.

Q: Is Puppeteer replacing Selenium/WebDriver?

No. Both projects are valuable for very different reasons:

- Selenium/WebDriver focuses on cross-browser automation; its value proposition is a single standard API that works across all major browsers.
- Puppeteer focuses on Chromium; its value proposition is richer functionality and higher reliability.

That said, you **can** use Puppeteer to run tests against Chromium, e.g. using the community-driven jest-puppeteer. While this probably shouldn't be your only testing solution, it does have a few good points compared to WebDriver:

- Puppeteer requires zero setup and comes bundled with the Chromium version it works best with, making it very easy to start with. At the end of the day, it's better to have a few tests running chromium-only, than no tests at all.
- Puppeteer has event-driven architecture, which removes a lot of potential flakiness. There's no need for evil "sleep(1000)" calls in puppeteer scripts.
- Puppeteer runs headless by default, which makes it fast to run. Puppeteer v1.5.0 also exposes browser contexts, making it possible to efficiently parallelize test execution.
- Puppeteer shines when it comes to debugging: flip the "headless" bit to false, add "slowMo", and you'll see what the browser is doing. You can even open Chrome DevTools to inspect the test environment.

Q: Why doesn't Puppeteer v.XXX work with Chromium v.YYY?

We see Puppeteer as an **indivisible entity** with Chromium. Each version of Puppeteer bundles a specific version of Chromium – **the only** version it is guaranteed to work with.

This is not an artificial constraint: A lot of work on Puppeteer is actually taking place in the Chromium repository. Here's a typical story:

- A Puppeteer bug is reported: https://github.com/puppeteer/puppeteer/issues/2709
- It turned out this is an issue with the DevTools protocol, so we're fixing it in Chromium: https://chromium-review.googlesource.com/c/chromium/src/+/1102154
- Once the upstream fix is landed, we roll updated Chromium into Puppeteer: https://github.com/puppeteer/puppeteer/pull/2769

However, oftentimes it is desirable to use Puppeteer with the official Google Chrome rather than Chromium. For this to work, you should install a `puppeteer-core` version that corresponds to the Chrome version.

For example, in order to drive Chrome 71 with puppeteer-core, use `chrome-71` npm tag:

```
npm install puppeteer-core@chrome-71
```

Q: Which Chromium version does Puppeteer use?

Look for the `chromium` entry in revisions.ts. To find the corresponding Chromium commit and version number, search for the revision prefixed by an `r` in OmahaProxy's "Find Releases" section.

Q: Which Firefox version does Puppeteer use?

Since Firefox support is experimental, Puppeteer downloads the latest Firefox Nightly when the `PUPPETEER_PRODUCT` environment variable is set to `firefox`. That's also why the value of `firefox` in revisions.ts is `latest` -- Puppeteer isn't tied to a particular Firefox version.

To fetch Firefox Nightly as part of Puppeteer installation:

```
PUPPETEER_PRODUCT=firefox npm i puppeteer
# or "yarn add puppeteer"
```

Q: What's considered a "Navigation"?

From Puppeteer's standpoint, **"navigation" is anything that changes a page's URL**. Aside from regular navigation where the browser hits the network to fetch a new document from the web server, this includes anchor navigations and History API usage.

With this definition of "navigation," **Puppeteer works seamlessly with single-page applications.**

Q: What's the difference between a "trusted" and "untrusted" input event?

In browsers, input events could be divided into two big groups: trusted vs. untrusted.

- **Trusted events**: events generated by users interacting with the page, e.g. using a mouse or keyboard.
- **Untrusted event**: events generated by Web APIs, e.g. `document.createEvent` or `element.click()` methods.

Websites can distinguish between these two groups:

- using an `Event.isTrusted` event flag
- sniffing for accompanying events. For example, every trusted `'click'` event is preceded by `'mousedown'` and `'mouseup'` events.

For automation purposes it's important to generate trusted events. **All input events generated with Puppeteer are trusted and fire proper accompanying events.** If, for some reason, one needs an untrusted event, it's always possible to hop into a page context with `page.evaluate` and generate a fake event:

```
await page.evaluate(() => {
  document.querySelector('button[type=submit]').click();
});
```

Q: What features does Puppeteer not support?

You may find that Puppeteer does not behave as expected when controlling pages that incorporate audio and video. (For example, video playback/screenshots is likely to fail.) There are two reasons for this:

- Puppeteer is bundled with Chromium — not Chrome — and so by default, it inherits all of Chromium's media-related limitations. This means that Puppeteer does not support licensed formats such as AAC or H.264. (However, it is possible to force Puppeteer to use a separately-installed version Chrome instead of

Chromium via the `executablePath` option to `puppeteer.launch`. You should only use this configuration if you need an official release of Chrome that supports these media formats.)

- Since Puppeteer (in all configurations) controls a desktop version of Chromium/Chrome, features that are only supported by the mobile version of Chrome are not supported. This means that Puppeteer does not support HTTP Live Streaming (HLS).

Q: I am having trouble installing / running Puppeteer in my test environment. Where should I look for help?

We have a troubleshooting guide for various operating systems that lists the required dependencies.

Q: How do I try/test a prerelease version of Puppeteer?

You can check out this repo or install the latest prerelease from npm:

```
npm i --save puppeteer@next
```

Please note that prerelease may be unstable and contain bugs.

Q: I have more questions! Where do I ask?

There are many ways to get help on Puppeteer:

- bugtracker
- Stack Overflow

Make sure to search these channels before posting your question.