
scrapy-playwright: Playwright integration for Scrapy

python 3.8 | 3.9 | 3.10 | 3.11 | 3.12

python 3.8 | 3.9 | 3.10 | 3.11 | 3.12

 codecov 100%

 codecov 100%

A Scrapy Download Handler which performs requests using Playwright for Python. It can be used to handle pages that require JavaScript (among other things), while adhering to the regular Scrapy workflow (i.e. without interfering with request scheduling, item processing, etc).

Requirements

After the release of version 2.0, which includes coroutine syntax support and asyncio support, Scrapy allows to integrate [asyncio](#)-based projects such as [Playwright](#).

Minimum required versions

- Python ≥ 3.8
- Scrapy ≥ 2.0 ($\neq 2.4.0$)
- Playwright ≥ 1.15

Installation

`scrapy-playwright` is available on PyPI and can be installed with `pip`:

```
1 pip install scrapy-playwright
```

`playwright` is defined as a dependency so it gets installed automatically, however it might be necessary to install the specific browser(s) that will be used:

```
1 playwright install
```

It's also possible to install only a subset of the available browsers:

```
1 playwright install firefox chromium
```

Changelog

See the changelog document.

Activation

Replace the default `http` and/or `https` Download Handlers through `DOWNLOAD_HANDLERS`:

```
1 DOWNLOAD_HANDLERS = {
2     "http": "scrapy_playwright.handler.ScrapyPlaywrightDownloadHandler"
3     ,
4     "https": "scrapy_playwright.handler.ScrapyPlaywrightDownloadHandler"
5 }
```

Note that the `ScrapyPlaywrightDownloadHandler` class inherits from the default `http/https` handler. Unless explicitly marked (see Basic usage), requests will be processed by the regular Scrapy download handler.

Also, be sure to install the `asyncio`-based Twisted reactor:

```
1 TWISTED_REACTOR = "twisted.internet.asyncioreactor.
   AsyncioSelectorReactor"
```

Basic usage

Set the `playwright` Request.meta key to download a request using Playwright:

```
1 import scrapy
2
3 class AwesomeSpider(scrapy.Spider):
4     name = "awesome"
5
6     def start_requests(self):
7         # GET request
8         yield scrapy.Request("https://httpbin.org/get", meta={"playwright": True})
9         # POST request
10        yield scrapy.FormRequest(
11            url="https://httpbin.org/post",
12            formdata={"foo": "bar"},
13            meta={"playwright": True},
14        )
15
16    def parse(self, response, **kwargs):
17        # 'response' contains the page as seen by the browser
18        return {"url": response.url}
```

Notes about the User-Agent header

By default, outgoing requests include the `User-Agent` set by Scrapy (either with the `USER_AGENT` or `DEFAULT_REQUEST_HEADERS` settings or via the `Request.headers` attribute). This could cause some sites to react in unexpected ways, for instance if the user agent does not match the running Browser. If you prefer the `User-Agent` sent by default by the specific browser you're using, set the Scrapy user agent to `None`.

Supported settings

PLAYWRIGHT_BROWSER_TYPE

Type `str`, default `"chromium"`.

The browser type to be launched, e.g. `chromium`, `firefox`, `webkit`.

```
1 PLAYWRIGHT_BROWSER_TYPE = "firefox"
```

PLAYWRIGHT_LAUNCH_OPTIONS

Type `dict`, default `{}`

A dictionary with options to be passed as keyword arguments when launching the Browser. See the docs for `BrowserType.launch` for a list of supported keyword arguments.

```
1 PLAYWRIGHT_LAUNCH_OPTIONS = {
2     "headless": False,
3     "timeout": 20 * 1000, # 20 seconds
4 }
```

PLAYWRIGHT_CDP_URL

Type `Optional[str]`, default `None`

The endpoint of a remote Chromium browser to connect using the Chrome DevTools Protocol, via `BrowserType.connect_over_cdp`. If this setting is used: * all non-persistent contexts will be created on the connected remote browser * the `PLAYWRIGHT_LAUNCH_OPTIONS` setting is ignored * the `PLAYWRIGHT_BROWSER_TYPE` setting must not be set to a value different than "chromium"

```
1 PLAYWRIGHT_CDP_URL = "http://localhost:9222"
```

PLAYWRIGHT_CDP_KWARGS

Type `dict[str, Any]`, default `{}`

Additional keyword arguments to be passed to `BrowserType.connect_over_cdp` when using `PLAYWRIGHT_CDP_URL`. The `endpoint_url` key is always ignored, `PLAYWRIGHT_CDP_URL` is used instead.

```
1 PLAYWRIGHT_CDP_KWARGS = {
2     "slow_mo": 1000,
3     "timeout": 10 * 1000
4 }
```

PLAYWRIGHT_CONTEXTS

Type `dict[str, dict]`, default `{}`

A dictionary which defines Browser contexts to be created on startup. It should be a mapping of (name, keyword arguments).

```
1 PLAYWRIGHT_CONTEXTS = {
2     "foobar": {
3         "context_arg1": "value",
4         "context_arg2": "value",
5     },
6     "default": {
7         "context_arg1": "value",
8         "context_arg2": "value",
9     },
10    "persistent": {
11        "user_data_dir": "/path/to/dir", # will be a persistent
            context
12        "context_arg1": "value",
13    },
14 }
```

See the section on browser contexts for more information. See also the docs for `Browser.new_context`.

PLAYWRIGHT_MAX_CONTEXTS

Type `Optional[int]`, default `None`

Maximum amount of allowed concurrent Playwright contexts. If unset or `None`, no limit is enforced. See the Maximum concurrent context count section for more information.

```
1 PLAYWRIGHT_MAX_CONTEXTS = 8
```

PLAYWRIGHT_DEFAULT_NAVIGATION_TIMEOUT

Type `Optional[float]`, default `None`

Timeout to be used when requesting pages by Playwright, in milliseconds. If `None` or unset, the default value will be used (30000 ms at the time of writing). See the docs for `BrowserContext.set_default_navigation_timeout`.

```
1 PLAYWRIGHT_DEFAULT_NAVIGATION_TIMEOUT = 10 * 1000 # 10 seconds
```

PLAYWRIGHT_PROCESS_REQUEST_HEADERS

Type `Optional[Union[Callable, str]]`, default `scrapy_playwright.headers.use_scrapy_headers`

A function (or the path to a function) that processes headers for a given request and returns a dictionary with the headers to be used (note that, depending on the browser, additional default headers could be sent as well). Coroutine functions (`async def`) are supported.

This will be called at least once for each Scrapy request (receiving said request and the corresponding Playwright request), but it could be called additional times if the given resource generates more requests (e.g. to retrieve assets like images or scripts).

The function must return a `dict` object, and receives the following positional arguments:

```
1 - browser_type: str
2 - playwright_request: playwright.async_api.Request
3 - scrapy_headers: scrapy.http.headers.Headers
```

The default function (`scrapy_playwright.headers.use_scrapy_headers`) tries to emulate Scrapy's behaviour for navigation requests, i.e. overriding headers with their values from the Scrapy request. For non-navigation requests (e.g. images, stylesheets, scripts, etc), only the `User-Agent` header is overridden, for consistency.

Setting `PLAYWRIGHT_PROCESS_REQUEST_HEADERS=None` will give complete control to Playwright, i.e. headers from Scrapy requests will be ignored and only headers set by Playwright will be sent. Keep in mind that in this case, headers passed via the `Request.headers` attribute or set by Scrapy components are ignored (including cookies set via the `Request.cookies` attribute).

```
1 def custom_headers(
```

```
2     browser_type: str,
3     playwright_request: playwright.async_api.Request,
4     scrapy_headers: scrapy.http.headers.Headers,
5 ) -> dict:
6     if browser_type == "firefox":
7         return {"User-Agent": "foo"}
8     return {"User-Agent": "bar"}
9
10 PLAYWRIGHT_PROCESS_REQUEST_HEADERS = custom_headers
```

PLAYWRIGHT_MAX_PAGES_PER_CONTEXT

Type **int**, defaults to the value of Scrapy's `CONCURRENT_REQUESTS` setting

Maximum amount of allowed concurrent Playwright pages for each context. See the notes about leaving unclosed pages.

```
1 PLAYWRIGHT_MAX_PAGES_PER_CONTEXT = 4
```

PLAYWRIGHT_ABORT_REQUEST

Type `Optional[Union[Callable, str]]`, default `None`

A predicate function (or the path to a function) that receives a `playwright.async_api.Request` object and must return `True` if the request should be aborted, `False` otherwise. Coroutine functions (`async def`) are supported.

Note that all requests will appear in the `DEBUG` level logs, however there will be no corresponding response log lines for aborted requests. Aborted requests are counted in the `playwright/request_count/aborted` job stats item.

```
1 def should_abort_request(request):
2     return (
3         request.resource_type == "image"
4         or ".jpg" in request.url
5     )
6
7 PLAYWRIGHT_ABORT_REQUEST = should_abort_request
```

General note about settings

For settings that accept object paths as strings, passing callable objects is only supported when using Scrapy>=2.4. With prior versions, only strings are supported.

Supported Request.meta keys

playwright

Type `bool`, default `False`

If set to a value that evaluates to `True` the request will be processed by Playwright.

```
1 return scrapy.Request("https://example.org", meta={"playwright": True})
```

playwright_context

Type `str`, default `"default"`

Name of the context to be used to download the request. See the section on browser contexts for more information.

```
1 return scrapy.Request(  
2     url="https://example.org",  
3     meta={  
4         "playwright": True,  
5         "playwright_context": "awesome_context",  
6     },  
7 )
```

playwright_context_kwargs

Type `dict`, default `{}`

A dictionary with keyword arguments to be used when creating a new context, if a context with the name specified in the `playwright_context` meta key does not exist already. See the section on browser contexts for more information.

```
1 return scrapy.Request(  
2     url="https://example.org",  
3     meta={  
4         "playwright": True,  
5         "playwright_context": "awesome_context",  
6         "playwright_context_kwargs": {  
7             "ignore_https_errors": True,  
8         },  
9     },  
10 )
```

playwright_include_page

Type `bool`, default `False`

If `True`, the Playwright page that was used to download the request will be available in the callback at `response.meta['playwright_page']`. If `False` (or unset) the page will be closed immediately after processing the request.

Important!

This meta key is entirely optional, it's NOT necessary for the page to load or for any asynchronous operation to be performed (specifically, it's NOT necessary for `PageMethod` objects to be applied). Use it only if you need access to the Page object in the callback that handles the response.

For more information and important notes see Receiving Page objects in callbacks.

```
1 return scrapy.Request(  
2     url="https://example.org",  
3     meta={"playwright": True, "playwright_include_page": True},  
4 )
```

playwright_page_event_handlers

Type `Dict[Str, Callable]`, default `{}`

A dictionary of handlers to be attached to page events. See Handling page events.

playwright_page_init_callback

Type `Optional[Union[Callable, str]]`, default `None`

A coroutine function (`async def`) to be invoked for newly created pages. Called after attaching page event handlers & setting up internal route handling, before making any request. It receives the Playwright page and the Scrapy request as positional arguments. Useful for initialization code. Ignored if the page for the request already exists (e.g. by passing `playwright_page`).

```
1 async def init_page(page, request):  
2     await page.add_init_script(path="./custom_script.js")  
3  
4 class AwesomeSpider(scrapy.Spider):  
5     def start_requests(self):  
6         yield scrapy.Request(  
7             url="https://httpbin.org/headers",  
8             meta={  
9                 "playwright": True,
```

```
10         "playwright_page_init_callback": init_page,
11     },
12 )
```

Important!

`scrapy-playwright` uses `Page.route` & `Page.unroute` internally, avoid using these methods unless you know exactly what you're doing.

`playwright_page_methods`

Type `Iterable[PageMethod]`, default `()`

An iterable of `scrapy_playwright.page.PageMethod` objects to indicate actions to be performed on the page before returning the final response. See Executing actions on pages.

`playwright_page`

Type `Optional[playwright.async_api.Page]`, default `None`

A Playwright page to be used to download the request. If unspecified, a new page is created for each request. This key could be used in conjunction with `playwright_include_page` to make a chain of requests using the same page. For instance:

```
1 def start_requests(self):
2     yield scrapy.Request(
3         url="https://httpbin.org/get",
4         meta={"playwright": True, "playwright_include_page": True},
5     )
6
7 def parse(self, response, **kwargs):
8     page = response.meta["playwright_page"]
9     yield scrapy.Request(
10        url="https://httpbin.org/headers",
11        callback=self.parse_headers,
12        meta={"playwright": True, "playwright_page": page},
13    )
```

`playwright_page_goto_kwargs`

Type `dict`, default `{}`

A dictionary with keyword arguments to be passed to the page's `goto` method when navigating to an URL. The `url` key is ignored if present, the request URL is used instead.

```
1 return scrapy.Request(
2     url="https://example.org",
3     meta={
4         "playwright": True,
5         "playwright_page_goto_kwargs": {
6             "wait_until": "networkidle",
7         },
8     },
9 )
```

playwright_security_details

Type `Optional[dict]`, read only

A dictionary with security information about the give response. Only available for HTTPS requests. Could be accessed in the callback via `response.meta['playwright_security_details']`

```
1 def parse(self, response, **kwargs):
2     print(response.meta["playwright_security_details"])
3     # {'issuer': 'DigiCert TLS RSA SHA256 2020 CA1', 'protocol': 'TLS
4       1.3', 'subjectName': 'www.example.org', 'validFrom': 1647216000,
5       'validTo': 1678838399}
```

Receiving Page objects in callbacks

Specifying a value that evaluates to `True` in the `playwright_include_page` meta key for a request will result in the corresponding `playwright.async_api.Page` object being available in the `playwright_page` meta key in the request callback. In order to be able to `await` coroutines on the provided `Page` object, the callback needs to be defined as a coroutine function (`async def`).

Caution

Use this carefully, and only if you really need to do things with the `Page` object in the callback. If pages are not properly closed after they are no longer necessary the spider job could get stuck because of the limit set by the `PLAYWRIGHT_MAX_PAGES_PER_CONTEXT` setting.

```
1 import scrapy
2
3 class AwesomeSpiderWithPage(scrapy.Spider):
4     name = "page_spider"
5
6     def start_requests(self):
```

```

7         yield scrapy.Request(
8             url="https://example.org",
9             callback=self.parse_first,
10            meta={"playwright": True, "playwright_include_page": True},
11            errback=self.errback_close_page,
12        )
13
14    def parse_first(self, response):
15        page = response.meta["playwright_page"]
16        return scrapy.Request(
17            url="https://example.com",
18            callback=self.parse_second,
19            meta={"playwright": True, "playwright_include_page": True,
20                "playwright_page": page},
21            errback=self.errback_close_page,
22        )
23
24    async def parse_second(self, response):
25        page = response.meta["playwright_page"]
26        title = await page.title() # "Example Domain"
27        await page.close()
28        return {"title": title}
29
30    async def errback_close_page(self, failure):
31        page = failure.request.meta["playwright_page"]
32        await page.close()
```

Notes:

- When passing `playwright_include_page=True`, make sure pages are always closed when they are no longer used. It's recommended to set a Request errback to make sure pages are closed even if a request fails (if `playwright_include_page=False` pages are automatically closed upon encountering an exception). This is important, as open pages count towards the limit set by `PLAYWRIGHT_MAX_PAGES_PER_CONTEXT` and crawls could freeze if the limit is reached and pages remain open indefinitely.
- Defining callbacks as `async def` is only necessary if you need to `await` things, it's NOT necessary if you just need to pass over the Page object from one callback to another (see the example above).
- Any network operations resulting from awaiting a coroutine on a Page object (`goto`, `go_back`, etc) will be executed directly by Playwright, bypassing the Scrapy request workflow (Scheduler, Middlewares, etc).

Browser contexts

Multiple browser contexts to be launched at startup can be defined via the `PLAYWRIGHT_CONTEXTS` setting.

Choosing a specific context for a request

Pass the name of the desired context in the `playwright_context` meta key:

```
1 yield scrapy.Request(  
2     url="https://example.org",  
3     meta={"playwright": True, "playwright_context": "first"},  
4 )
```

Default context

If a request does not explicitly indicate a context via the `playwright_context` meta key, it falls back to using a general context called **default**. This **default** context can also be customized on startup via the `PLAYWRIGHT_CONTEXTS` setting.

Persistent contexts

Pass a value for the `user_data_dir` keyword argument to launch a context as persistent. See also `BrowserType.launch_persistent_context`.

Note that persistent contexts are launched independently from the main browser instance, hence keyword arguments passed in the `PLAYWRIGHT_LAUNCH_OPTIONS` setting do not apply.

Creating contexts while crawling

If the context specified in the `playwright_context` meta key does not exist, it will be created. You can specify keyword arguments to be passed to `Browser.new_context` in the `playwright_context_kwargs` meta key:

```
1 yield scrapy.Request(  
2     url="https://example.org",  
3     meta={  
4         "playwright": True,  
5         "playwright_context": "new",  
6         "playwright_context_kwargs": {  
7             "java_script_enabled": False,
```

```

8         "ignore_https_errors": True,
9         "proxy": {
10             "server": "http://myproxy.com:3128",
11             "username": "user",
12             "password": "pass",
13         },
14     },
15 },
16 )

```

Please note that if a context with the specified name already exists, that context is used and `playwright_context_kwargs` are ignored.

Closing contexts while crawling

After receiving the Page object in your callback, you can access a context through the corresponding `Page.context` attribute, and await `close` on it.

```

1  def parse(self, response, **kwargs):
2      yield scrapy.Request(
3          url="https://example.org",
4          callback=self.parse_in_new_context,
5          errback=self.close_context_on_error,
6          meta={
7              "playwright": True,
8              "playwright_context": "awesome_context",
9              "playwright_include_page": True,
10         },
11     )
12
13  async def parse_in_new_context(self, response):
14      page = response.meta["playwright_page"]
15      title = await page.title()
16      await page.close()
17      await page.context.close()
18      return {"title": title}
19
20  async def close_context_on_error(self, failure):
21      page = failure.request.meta["playwright_page"]
22      await page.close()
23      await page.context.close()

```

Avoid race conditions & memory leaks when closing contexts

Make sure to close the page before closing the context. See this comment in #191 for more information.

Maximum concurrent context count

Specify a value for the `PLAYWRIGHT_MAX_CONTEXTS` setting to limit the amount of concurrent contexts. Use with caution: it's possible to block the whole crawl if contexts are not closed after they are no longer used (refer to this section to dynamically close contexts). Make sure to define an errback to still close contexts even if there are errors.

Proxy support

Proxies are supported at the Browser level by specifying the `proxy` key in the `PLAYWRIGHT_LAUNCH_OPTIONS` setting:

```
1 from scrapy import Spider, Request
2
3 class ProxySpider(Spider):
4     name = "proxy"
5     custom_settings = {
6         "PLAYWRIGHT_LAUNCH_OPTIONS": {
7             "proxy": {
8                 "server": "http://myproxy.com:3128",
9                 "username": "user",
10                "password": "pass",
11            },
12        }
13    }
14
15    def start_requests(self):
16        yield Request("http://httpbin.org/get", meta={"playwright":
17            True})
18
19    def parse(self, response, **kwargs):
20        print(response.text)
```

Proxies can also be set at the context level with the `PLAYWRIGHT_CONTEXTS` setting:

```
1 PLAYWRIGHT_CONTEXTS = {
2     "default": {
3         "proxy": {
4             "server": "http://default-proxy.com:3128",
5             "username": "user1",
6             "password": "pass1",
7         },
8     },
9     "alternative": {
10        "proxy": {
11            "server": "http://alternative-proxy.com:3128",
12            "username": "user2",
```

```
13         "password": "pass2",
14     },
15 },
16 }
```

Or passing a `proxy` key when creating contexts while crawling.

See also: * `zyte-smartproxy-playwright`: seamless support for Zyte Smart Proxy Manager in the Node.js version of Playwright. * the upstream Playwright for Python section on HTTP Proxies.

Executing actions on pages

A sorted iterable (e.g. `list`, `tuple`, `dict`) of `PageMethod` objects could be passed in the `playwright_page_methods` Request.meta key to request methods to be invoked on the `Page` object before returning the final `Response` to the callback.

This is useful when you need to perform certain actions on a page (like scrolling down or clicking links) and you want to handle only the final result in your callback.

PageMethod class

`scrapy_playwright.page.PageMethod(method: str, *args, **kwargs):`

Represents a method to be called (and awaited if necessary) on a `playwright.page.Page` object (e.g. “click”, “screenshot”, “evaluate”, etc). `method` is the name of the method, `*args` and `**kwargs` are passed when calling such method. The return value will be stored in the `PageMethod.result` attribute.

For instance:

```
1 def start_requests(self):
2     yield Request(
3         url="https://example.org",
4         meta={
5             "playwright": True,
6             "playwright_page_methods": [
7                 PageMethod("screenshot", path="example.png", full_page=
8                     True),
9             ],
10        },
11    )
12
13 def parse(self, response, **kwargs):
14     screenshot = response.meta["playwright_page_methods"][0]
15     # screenshot.result contains the image's bytes
```

produces the same effect as:

```
1 def start_requests(self):
2     yield Request(
3         url="https://example.org",
4         meta={"playwright": True, "playwright_include_page": True},
5     )
6
7 async def parse(self, response, **kwargs):
8     page = response.meta["playwright_page"]
9     screenshot = await page.screenshot(path="example.png", full_page=
10         True)
11     # screenshot contains the image's bytes
12     await page.close()
```

Supported methods

Refer to the upstream docs for the [Page](#) class to see available methods.

Impact on Response objects

Certain [Response](#) attributes (e.g. `url`, `ip_address`) reflect the state after the last action performed on a page. If you issue a [PageMethod](#) with an action that results in a navigation (e.g. a `click` on a link), the `Response.url` attribute will point to the new URL, which might be different from the request's URL.

Handling page events

A dictionary of Page event handlers can be specified in the `playwright_page_event_handlers` Request.meta key. Keys are the name of the event to be handled (e.g. `dialog`, `download`, etc). Values can be either callables or strings (in which case a spider method with the name will be looked up).

Example:

```
1 from playwright.async_api import Dialog
2
3 async def handle_dialog(dialog: Dialog) -> None:
4     logging.info(f"Handled dialog with message: {dialog.message}")
5     await dialog.dismiss()
6
7 class EventSpider(scrapy.Spider):
8     name = "event"
9
```

```

10     def start_requests(self):
11         yield scrapy.Request(
12             url="https://example.org",
13             meta={
14                 "playwright": True,
15                 "playwright_page_event_handlers": {
16                     "dialog": handle_dialog,
17                     "response": "handle_response",
18                 },
19             },
20         )
21
22     async def handle_response(self, response: PlaywrightResponse) ->
23         None:
24         logging.info(f"Received response with URL {response.url}")

```

See the upstream [Page](#) docs for a list of the accepted events and the arguments passed to their handlers.

Notes about page event handlers

- Event handlers will remain attached to the page and will be called for subsequent downloads using the same page unless they are removed later. This is usually not a problem, since by default requests are performed in single-use pages.
- Event handlers will process Playwright objects, not Scrapy ones. For example, for each Scrapy request/response there will be a matching Playwright request/response, but not the other way: background requests/responses to get images, scripts, stylesheets, etc are not seen by Scrapy.

Memory usage extension

The default Scrapy memory usage extension (`scrapy.extensions.memusage.MemoryUsage`) does not include the memory used by Playwright because the browser is launched as a separate process. The scrapy-playwright package provides a replacement extension which also considers the memory used by Playwright. This extension needs the `psutil` package to work.

Update the EXTENSIONS setting to disable the built-in Scrapy extension and replace it with the one from the scrapy-playwright package:

```

1 # settings.py
2 EXTENSIONS = {
3     "scrapy.extensions.memusage.MemoryUsage": None,
4     "scrapy_playwright.memusage.ScrapyPlaywrightMemoryUsageExtension":
5         0,
6 }

```

Refer to the upstream docs for more information about supported settings.

Examples

Click on a link, save the resulting page as PDF

```
1 class ClickAndSavePdfSpider(scrapy.Spider):
2     name = "pdf"
3
4     def start_requests(self):
5         yield scrapy.Request(
6             url="https://example.org",
7             meta=dict(
8                 playwright=True,
9                 playwright_page_methods={
10                     "click": PageMethod("click", selector="a"),
11                     "pdf": PageMethod("pdf", path="/tmp/file.pdf"),
12                 },
13             ),
14         )
15
16     def parse(self, response, **kwargs):
17         pdf_bytes = response.meta["playwright_page_methods"]["pdf"].
            result
18         with open("iana.pdf", "wb") as fp:
19             fp.write(pdf_bytes)
20         yield {"url": response.url} # response.url is "https://www.
            iana.org/domains/reserved"
```

Scroll down on an infinite scroll page, take a screenshot of the full page

```
1 class ScrollSpider(scrapy.Spider):
2     name = "scroll"
3
4     def start_requests(self):
5         yield scrapy.Request(
6             url="http://quotes.toscrape.com/scroll",
7             meta=dict(
8                 playwright=True,
9                 playwright_include_page=True,
10                 playwright_page_methods=[
11                     PageMethod("wait_for_selector", "div.quote"),
12                     PageMethod("evaluate", "window.scrollTo(0, document
13                         .body.scrollHeight)",
14                     PageMethod("wait_for_selector", "div.quote:nth-
15                         child(11)", # 10 per page
16                 ],
17             ),
18         )
```

```
17
18     async def parse(self, response, **kwargs):
19         page = response.meta["playwright_page"]
20         await page.screenshot(path="quotes.png", full_page=True)
21         await page.close()
22         return {"quote_count": len(response.css("div.quote"))} #
            quotes from several pages
```

See the examples directory for more.

Known issues

Lack of native support for Windows

This package does not work natively on Windows. This is because:

- Playwright runs the driver in a subprocess. Source: Playwright repository.
- “On Windows, the default event loop `ProactorEventLoop` supports subprocesses, whereas `SelectorEventLoop` does not”. Source: Python docs.
- Twisted’s `asyncio` reactor requires the `SelectorEventLoop`. Source: Twisted repository.

Some users have reported having success running under WSL. See also #78 for information about working in headful mode under WSL.

No per-request proxy support

Specifying a proxy via the `proxy` Request meta key is not supported. Refer to the Proxy support section for more information.

Unsupported signals

The `headers_received` and `bytes_received` signals are not fired by the scrapy-playwright download handler.

Reporting issues

Before opening an issue please make sure the unexpected behavior can only be observed by using this package and not with standalone Playwright. To do this, translate your spider code to a reasonably close Playwright script: if the issue also occurs this way, you should instead report it upstream. For instance:

```

1 import scrapy
2
3 class ExampleSpider(scrapy.Spider):
4     name = "example"
5
6     def start_requests(self):
7         yield scrapy.Request(
8             url="https://example.org",
9             meta=dict(
10                 playwright=True,
11                 playwright_page_methods=[
12                     PageMethod("screenshot", path="example.png",
13                               full_page=True),
14                 ],
15             ),
16         )

```

translates roughly to:

```

1 import asyncio
2 from playwright.async_api import async_playwright
3
4 async def main():
5     async with async_playwright() as pw:
6         browser = await pw.chromium.launch()
7         page = await browser.new_page()
8         await page.goto("https://example.org")
9         await page.screenshot(path="example.png", full_page=True)
10        await browser.close()
11
12 asyncio.run(main())

```

Reproducible code example

When opening an issue please include a Minimal, Reproducible Example that shows the reported behavior. In addition, please make the code as self-contained as possible so an active Scrapy project is not required and the spider can be executed directly from a file with `scrapy runspider`. This usually means including the relevant settings in the spider's `custom_settings` attribute:

```

1 import scrapy
2
3 class ExampleSpider(scrapy.Spider):
4     name = "example"
5     custom_settings = {
6         "TWISTED_REACTOR": "twisted.internet.asyncioreactor.
7         AsyncioSelectorReactor",
8         "DOWNLOAD_HANDLERS": {

```

```
8         "https": "scrapy_playwright.handler.  
9             scrapy_playwright.handler.  
10            scrapy_playwright.handler.  
11            scrapy_playwright.handler.",  
12        },  
13    def start_requests(self):  
14        yield scrapy.Request(  
15            url="https://example.org",  
16            meta={"playwright": True},  
17        )
```

Frequently Asked Questions

See the FAQ document.

Deprecation policy

Deprecated features will be supported for at least six months following the release that deprecated them. After that, they may be removed at any time. See the changelog for more information about deprecations and removals.