BrowserContext

- extends: EventEmitter

BrowserContexts provide a way to operate multiple independent browser sessions.

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

Playwright allows creating "incognito" browser contexts with browser.new_context() method. "Incognito" browser contexts don't write any browsing data to disk.

Sync Async

```
# create a new incognito browser context
context = browser.new_context()
# create a new page inside context.
page = context.new_page()
page.goto("https://example.com")
# dispose context once it is no longer needed.
context.close()
```

Methods

add_cookies

Added in: v1.8

Adds cookies into this browser context. All pages within this context will have these cookies installed. Cookies can be obtained via browser_context.cookies().

Usage

Sync Async

```
browser_context.add_cookies([cookie_object1, cookie_object2])
```

Arguments

- `cookies` List[Dict]
  - `name` str
  - `value` str
  - `url` str *(optional)*

    either url or domain / path are required. Optional.
  - `domain` str *(optional)*

    either url or domain / path are required Optional.
  - `path` str *(optional)*

    either url or domain / path are required Optional.
  - `expires` float *(optional)*

    Unix time in seconds. Optional.
  - `httpOnly` bool *(optional)*

    Optional.
  - `secure` bool *(optional)*

    Optional.
  - `sameSite` "Strict"|"Lax"|"None" *(optional)*

    Optional.

  Adds cookies to the browser context.

  For the cookie to apply to all subdomains as well, prefix domain with a dot, like this: ".example.com".

add_init_script

Added in: v1.8

Adds a script which would be evaluated in one of the following scenarios:

- Whenever a page is created in the browser context or is navigated.
- Whenever a child frame is attached or navigated in any page in the browser context. In this case, the script is evaluated in the context of the newly attached frame.

The script is evaluated 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`.

Usage

An example of overriding `Math.random` before the page loads:

```
// preload.js
Math.random = () => 42;
```

Sync **Async**

```
# in your playwright script, assuming the preload.js file is in same directory.
browser_context.add_init_script(path="preload.js")
```

> **NOTE**
>
> The order of evaluation of multiple scripts installed via browser_context.add_init_script() and page.add_init_script() is not defined.

Arguments

- `path` Union[str, pathlib.Path] *(optional)*

  Path to the JavaScript file. If `path` is a relative path, then it is resolved relative to the current working directory. Optional.

- `script` str *(optional)*

  Script to be evaluated in all pages in the browser context. Optional.

clear_cookies

Added in: v1.8

Clears context cookies.

Usage

```
browser_context.clear_cookies()
```

clear_permissions

Added in: v1.8

Clears all permission overrides for the browser context.

Usage

Sync **Async**

```
context = browser.new_context()
context.grant_permissions(["clipboard-read"])
# do stuff ..
context.clear_permissions()
```

close

Added in: v1.8

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

> **NOTE**
>
> The default browser context cannot be closed.

Usage

```
browser_context.close()
```

cookies

Added in: v1.8

If no URLs are specified, this method returns all cookies. If URLs are specified, only cookies that affect those URLs are returned.

Usage

```
browser_context.cookies()
browser_context.cookies(**kwargs)
```

Arguments

- `urls` str|List[str] *(optional)*

  Optional list of URLs.

Returns

- List[Dict]
  - `name` str
  - `value` str
  - `domain` str
  - `path` str
  - `expires` float

    Unix time in seconds.
  - `httpOnly` bool
  - `secure` bool
  - `sameSite` "Strict"|"Lax"|"None"

expect_console_message

Added in: v1.34

Performs action and waits for a ConsoleMessage to be logged by in the pages in the context. If predicate is provided, it passes ConsoleMessage value into the `predicate` function and waits for `predicate(message)` to return a truthy value. Will throw an error if the page is closed before the browser_context.on("console") event is fired.

Usage

```
browser_context.expect_console_message()
browser_context.expect_console_message(**kwargs)
```

Arguments

- `predicate` Callable[ConsoleMessage]:bool *(optional)*

  Receives the ConsoleMessage object and resolves to truthy value when the waiting should resolve.

- `timeout` float *(optional)*

  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 browser_context.set_default_timeout().

Returns

- EventContextManager[ConsoleMessage]

expect_event

Added in: v1.8

Waits for event to fire and passes its value into the predicate function. Returns when the predicate returns truthy value. Will throw an error if the context closes before the event is fired. Returns the event data value.

Usage

Sync Async

```python
with context.expect_event("page") as event_info:
    page.get_by_role("button").click()
page = event_info.value
```

Arguments

- `event` str

  Event name, same one would pass into `browserContext.on(event)`.

- `predicate` Callable *(optional)*

  Receives the event data and resolves to truthy value when the waiting should resolve.

- `timeout` float *(optional)*

  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 browser_context.set_default_timeout().

Returns

- EventContextManager

expect_page

Added in: v1.9

Performs action and waits for a new Page to be created in the context. If predicate is provided, it passes Page value into the `predicate` function and waits for `predicate(event)` to return a truthy value. Will throw an error if the context closes before new Page is created.

Usage

```python
browser_context.expect_page()
browser_context.expect_page(**kwargs)
```

Arguments

- `predicate` Callable[Page]:bool *(optional)*

  Receives the Page object and resolves to truthy value when the waiting should resolve.

- `timeout` float *(optional)*

  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 browser_context.set_default_timeout().

Returns

- EventContextManager[Page]

expose_binding

Added in: v1.8

The method adds a function called `name` on the `window` object of every frame in every page in the context. When called, the function executes `callback` and returns a Promise which resolves to the return value of `callback`. If the `callback` returns a Promise, it will be awaited.

The first argument of the `callback` function contains information about the caller: `{ browserContext: BrowserContext, page: Page, frame: Frame }`.

See page.expose_binding() for page-only version.

Usage

An example of exposing page URL to all frames in all pages in the context:

Sync **Async**

```
from playwright.sync_api import sync_playwright, Playwright

def run(playwright: Playwright):
    webkit = playwright.webkit
    browser = webkit.launch(headless=false)
    context = browser.new_context()
    context.expose_binding("pageURL", lambda source: source["page"].url)
    page = context.new_page()
    page.set_content("""
    <script>
      async function onClick() {
        document.querySelector('div').textContent = await window.pageURL();
      }
    </script>
    <button onclick="onClick()">Click me</button>
    <div></div>
    """)
    page.get_by_role("button").click()

with sync_playwright() as playwright:
    run(playwright)
```

An example of passing an element handle:

Sync **Async**

```
def print(source, element):
    print(element.text_content())

context.expose_binding("clicked", print, handle=true)
page.set_content("""
  <script>
    document.addEventListener('click', event => window.clicked(event.target));
  </script>
  <div>Click me</div>
  <div>Or click me</div>
""")
```

Arguments

- `name` str

  Name of the function on the window object.

- `callback` Callable

  Callback function that will be called in the Playwright's context.

- `handle` bool *(optional)*

Whether to pass the argument as a handle, instead of passing by value. When passing a handle, only one argument is supported. When passing by value, multiple arguments are supported.

expose_function

Added in: v1.8

The method adds a function called `name` on the `window` object of every frame in every page in the context. When called, the function executes `callback` and returns a Promise which resolves to the return value of `callback`.

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

See page.expose_function() for page-only version.

Usage

An example of adding a `sha256` function to all pages in the context:

Sync Async

```
import hashlib
from playwright.sync_api import sync_playwright

def sha256(text: str) -> str:
    m = hashlib.sha256()
    m.update(bytes(text, "utf8"))
    return m.hexdigest()


def run(playwright: Playwright):
    webkit = playwright.webkit
    browser = webkit.launch(headless=False)
    context = browser.new_context()
    context.expose_function("sha256", sha256)
    page = context.new_page()
    page.set_content("""
        <script>
          async function onClick() {
            document.querySelector('div').textContent = await
window.sha256('PLAYWRIGHT');
          }
        </script>
        <button onclick="onClick()">Click me</button>
        <div></div>
    """)
    page.get_by_role("button").click()

with sync_playwright() as playwright:
    run(playwright)
```

Arguments

- `name` str

  Name of the function on the window object.

- `callback` Callable

  Callback function that will be called in the Playwright's context.

grant_permissions

Added in: v1.8

Grants specified permissions to the browser context. Only grants corresponding permissions to the given origin if specified.

Usage

```
browser_context.grant_permissions(permissions)
browser_context.grant_permissions(permissions, **kwargs)
```

Arguments

- `permissions` List[str]

  A permission or an array of permissions to grant. Permissions can be one of the following values:

  - `'geolocation'`
  - `'midi'`
  - `'midi-sysex'` (system-exclusive midi)
  - `'notifications'`

  - `'camera'`
  - `'microphone'`
  - `'background-sync'`
  - `'ambient-light-sensor'`
  - `'accelerometer'`
  - `'gyroscope'`
  - `'magnetometer'`
  - `'accessibility-events'`
  - `'clipboard-read'`
  - `'clipboard-write'`
  - `'payment-handler'`
- `origin` str *(optional)*

  The origin to grant permissions to, e.g. "https://example.com".

new_cdp_session

Added in: v1.11

NOTE

CDP sessions are only supported on Chromium-based browsers.

Returns the newly created session.

Usage

```
browser_context.new_cdp_session(page)
```

Arguments

- `page` Page|Frame

  Target to create new session for. For backwards-compatibility, this parameter is named `page`, but it can be a `Page` or `Frame` type.

Returns

- CDPSession

new_page

Added in: v1.8

Creates a new page in the browser context.

Usage

```
browser_context.new_page()
```

Returns

- Page

route

Added in: v1.8

Routing provides the capability to modify network requests that are made by any page in the browser context. Once route is enabled, every request matching the url pattern will stall unless it's continued, fulfilled or aborted.

NOTE

browser_context.route() will not intercept requests intercepted by Service Worker. See this issue. We recommend disabling Service Workers when using request interception by setting `browser.new_context.service_workers` to `'block'`.

Usage

An example of a naive handler that aborts all image requests:

Sync Async

```
context = browser.new_context()
page = context.new_page()
context.route("**/*.{png,jpg,jpeg}", lambda route: route.abort())
```

```
page.goto("https://example.com")
browser.close()
```

or the same snippet using a regex pattern instead:

Sync **Async**

```
context = browser.new_context()
page = context.new_page()
context.route(re.compile(r"(\.png$)|(\.jpg$)"), lambda route: route.abort())
page = await context.new_page()
page = context.new_page()
page.goto("https://example.com")
browser.close()
```

It is possible to examine the request to decide the route action. For example, mocking all requests that contain some post data, and leaving all other requests as is:

Sync **Async**

```
def handle_route(route):
  if ("my-string" in route.request.post_data):
    route.fulfill(body="mocked-data")
  else:
    route.continue_()
context.route("/api/**", handle_route)
```

Page routes (set up with page.route()) take precedence over browser context routes when request matches both handlers.

To remove a route with its handler you can use browser_context.unroute().

> **NOTE**
>
> Enabling routing disables http cache.

Arguments

- `url` str|Pattern|Callable[URL]:bool

A glob pattern, regex pattern or predicate receiving URL to match while routing. When a `base_url` via the context options was provided and the passed URL is a path, it gets merged via the `new URL()` constructor.

- `handler` Callable[Route, Request]:Promise[Any]|Any

  handler function to route the request.

- `times` int *(optional)* Added in: v1.15

  How often a route should be used. By default it will be used every time.

route_from_har

Added in: v1.23

If specified the network requests that are made in the context will be served from the HAR file. Read more about Replaying from HAR.

Playwright will not serve requests intercepted by Service Worker from the HAR file. See this issue. We recommend disabling Service Workers when using request interception by setting `browser.new_context.service_workers` to `'block'`.

Usage

```
browser_context.route_from_har(har)
browser_context.route_from_har(har, **kwargs)
```

Arguments

- `har` Union[str, pathlib.Path]

  Path to a HAR file with prerecorded network data. If `path` is a relative path, then it is resolved relative to the current working directory.

- `not_found` "abort"|"fallback" *(optional)*

  - If set to 'abort' any request not found in the HAR file will be aborted.
  - If set to 'fallback' falls through to the next route handler in the handler chain.

  Defaults to abort.

- `update` bool *(optional)*

  If specified, updates the given HAR with the actual network information instead of serving from file. The file is written to disk when browser_context.close() is called.

- `update_content` "embed"|"attach" *(optional)* Added in: v1.32

  Optional setting to control resource content management. If `attach` is specified, resources are persisted as separate files or entries in the ZIP archive. If `embed` is specified, content is stored inline the HAR file.

- `update_mode` "full"|"minimal" *(optional)* Added in: v1.32

  When set to `minimal`, only record information necessary for routing from HAR. This omits sizes, timing, page, cookies, security and other types of HAR information that are not used when replaying from HAR. Defaults to `minimal`.

- `url` str|Pattern *(optional)*

  A glob pattern, regular expression or predicate to match the request URL. Only requests with URL matching the pattern will be served from the HAR file. If not specified, all requests are served from the HAR file.

set_default_navigation_timeout

Added in: v1.8

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

- page.go_back()
- page.go_forward()
- page.goto()
- page.reload()
- page.set_content()
- page.expect_navigation()

NOTE

page.set_default_navigation_timeout() and page.set_default_timeout() take priority over browser_context.set_default_navigation_timeout().

Usage

```
browser_context.set_default_navigation_timeout(timeout)
```

Arguments

- `timeout` float

  Maximum navigation time in milliseconds

set_default_timeout

Added in: v1.8

This setting will change the default maximum time for all the methods accepting `timeout` option.

NOTE

page.set_default_navigation_timeout(), page.set_default_timeout() and browser_context.set_default_navigation_timeout() take priority over browser_context.set_default_timeout().

Usage

```
browser_context.set_default_timeout(timeout)
```

Arguments

- `timeout` float

  Maximum time in milliseconds

set_extra_http_headers

Added in: v1.8

The extra HTTP headers will be sent with every request initiated by any page in the context. These headers are merged with page-specific extra HTTP headers set with page.set_extra_http_headers(). If page overrides a particular header, page-specific header value will be used instead of the browser context header value.

> **NOTE**
>
> browser_context.set_extra_http_headers() does not guarantee the order of headers in the outgoing requests.

Usage

```
browser_context.set_extra_http_headers(headers)
```

Arguments

- `headers` Dict[str, str]

  An object containing additional HTTP headers to be sent with every request. All header values must be strings.

set_geolocation

Added in: v1.8

Sets the context's geolocation. Passing `null` or `undefined` emulates position unavailable.

Usage

Sync **Async**

```
browser_context.set_geolocation({"latitude": 59.95, "longitude": 30.31667})
```

> **NOTE**
>
> Consider using browser_context.grant_permissions() to grant permissions for the browser context pages to read its geolocation.

Arguments

- `geolocation` NoneType|Dict
  - `latitude` float

    Latitude between -90 and 90.
  - `longitude` float

    Longitude between -180 and 180.
  - `accuracy` float *(optional)*

    Non-negative accuracy value. Defaults to `0`.

set_offline

Added in: v1.8

Usage

```
browser_context.set_offline(offline)
```

Arguments

- `offline` bool

  Whether to emulate network being offline for the browser context.

storage_state

Added in: v1.8

Returns storage state for this browser context, contains current cookies and local storage snapshot.

Usage

```
browser_context.storage_state()
browser_context.storage_state(**kwargs)
```

Arguments

- `path` Union[str, pathlib.Path] *(optional)*

  The file path to save the storage state to. If `path` is a relative path, then it is resolved relative to current working directory. If no path is provided, storage state is still returned, but won't be saved to the disk.

Returns

- Dict
  - `cookies` List[Dict]
    - `name` str
    - `value` str
    - `domain` str
    - `path` str
    - `expires` float

      Unix time in seconds.
    - `httpOnly` bool
    - `secure` bool
    - `sameSite` "Strict"|"Lax"|"None"
  - `origins` List[Dict]
    - `origin` str
    - `localStorage` List[Dict]
      - `name` str
      - `value` str

unroute

Added in: v1.8

Removes a route created with browser_context.route(). When `handler` is not specified, removes all routes for the `url`.

Usage

```
browser_context.unroute(url)
browser_context.unroute(url, **kwargs)
```

Arguments

- `url` str|Pattern|Callable[URL]:bool

  A glob pattern, regex pattern or predicate receiving URL used to register a routing with browser_context.route().

- `handler` Callable[Route, Request]:Promise[Any]|Any *(optional)*

  Optional handler function used to register a routing with browser_context.route().

wait_for_event

Added in: v1.8

> **NOTE**
>
> In most cases, you should use browser_context.expect_event().

Waits for given `event` to fire. If predicate is provided, it passes event's value into the `predicate` function and waits for `predicate(event)` to return a truthy value. Will throw an error if the browser context is closed before the `event` is fired.

Usage

```
browser_context.wait_for_event(event)
browser_context.wait_for_event(event, **kwargs)
```

Arguments

- `event` str

  Event name, same one typically passed into `*.on(event)`.

- `predicate` Callable *(optional)*

Receives the event data and resolves to truthy value when the waiting should resolve.

- `timeout` float *(optional)*

  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 browser_context.set_default_timeout().

Returns

- Any

Properties

background_pages

Added in: v1.11

> **NOTE**
>
> Background pages are only supported on Chromium-based browsers.

All existing background pages in the context.

Usage

```
browser_context.background_pages
```

Returns

- List[Page]

browser

Added in: v1.8

Returns the browser instance of the context. If it was launched as a persistent context null gets returned.

Usage

```
browser_context.browser
```

Returns

- NoneType|Browser

pages

Added in: v1.8

Returns all open pages in the context.

Usage

```
browser_context.pages
```

Returns

- List[Page]

request

Added in: v1.16

API testing helper associated with this context. Requests made with this API will use context cookies.

Usage

```
browser_context.request
```

Type

- APIRequestContext

service_workers

Added in: v1.11

NOTE

Service workers are only supported on Chromium-based browsers.

All existing service workers in the context.

Usage

```
browser_context.service_workers
```

Returns

- List[Worker]

tracing

Added in: v1.12

Usage

```
browser_context.tracing
```

Type

- Tracing

Events

on("backgroundpage")

Added in: v1.11

NOTE

Only works with Chromium browser's persistent context.

Emitted when new background page is created in the context.

Sync Async

```
background_page = context.wait_for_event("backgroundpage")
```

Usage

```
browser_context.on("backgroundpage", handler)
```

Event data

- Page

on("close")

Added in: v1.8

Emitted when Browser context gets closed. This might happen because of one of the following:

- Browser context is closed.
- Browser application is closed or crashed.
- The browser.close() method was called.

Usage

```
browser_context.on("close", handler)
```

Event data

- BrowserContext

on("console")

Added in: v1.34

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` and the page are available on the ConsoleMessage event handler argument.

Usage

Sync   Async

```
def print_args(msg):
    for arg in msg.args:
        print(arg.json_value())

context.on("console", print_args)
page.evaluate("console.log('hello', 5, { foo: 'bar' })")
```

Event data

- ConsoleMessage

on("dialog")

Added in: v1.34

Emitted when a JavaScript dialog appears, such as `alert`, `prompt`, `confirm` or `beforeunload`. Listener **must** either dialog.accept() or dialog.dismiss() the dialog - otherwise the page will freeze waiting for the dialog, and actions like click will never finish.

Usage

```
context.on("dialog", lambda dialog: dialog.accept())
```

NOTE

When no page.on("dialog") or browser_context.on("dialog") listeners are present, all dialogs are automatically dismissed.

Event data

- Dialog

on("page")

Added in: v1.8

The event is emitted when a new Page is created in the BrowserContext. The page may still be loading. The event will also fire for popup pages. See also page.on("popup") to receive events about popups relevant to a specific page.

The earliest moment that page is available is when it has navigated to the initial url. For example, when opening a popup with `window.open('http://example.com')`, this event will fire when the network request to "http://example.com" is done and its response has started loading in the popup.

Sync **Async**

```
with context.expect_page() as page_info:
    page.get_by_text("open new page").click(),
page = page_info.value
print(page.evaluate("location.href"))
```

> **NOTE**
>
> Use page.wait_for_load_state() to wait until the page gets to a particular state (you should not need it in most cases).

Usage

```
browser_context.on("page", handler)
```

Event data

- Page

on("request")

Added in: v1.12

Emitted when a request is issued from any pages created through this context. The request object is read-only. To only listen for requests from a particular page, use page.on("request").

In order to intercept and mutate requests, see browser_context.route() or page.route().

Usage

```
browser_context.on("request", handler)
```

Event data

- Request

on("requestfailed")

Added in: v1.12

Emitted when a request fails, for example by timing out. To only listen for failed requests from a particular page, use page.on("requestfailed").

> **NOTE**
>
> HTTP Error responses, such as 404 or 503, are still successful responses from HTTP standpoint, so request will complete with browser_context.on("requestfinished") event and not with browser_context.on("requestfailed").

Usage

```
browser_context.on("requestfailed", handler)
```

Event data

- Request

on("requestfinished")

Added in: v1.12

Emitted when a request finishes successfully after downloading the response body. For a successful response, the sequence of events is `request`, `response` and `requestfinished`. To listen for successful requests from a particular page, use page.on("requestfinished").

Usage

```
browser_context.on("requestfinished", handler)
```

Event data

- Request

on("response")

Added in: v1.12

Emitted when response status and headers are received for a request. For a successful response, the sequence of events is `request`, `response` and `requestfinished`. To listen for response events from a particular page, use page.on("response").

Usage

```
browser_context.on("response", handler)
```

Event data

- Response

on("serviceworker")

Added in: v1.11

> **NOTE**
>
> Service workers are only supported on Chromium-based browsers.

Emitted when new service worker is created in the context.

Usage

```
browser_context.on("serviceworker", handler)
```

Event data

- Worker

on("weberror")

Added in: v1.38

Emitted when exception is unhandled in any of the pages in this context. To listen for errors from a particular page, use page.on("pageerror") instead.

Usage

```
browser_context.on("weberror", handler)
```

Event data

- WebError