Pydoll - Python Chrome DevTools Protocol Library

Repo: thalissonvs, This doc: kgruiz

Source repo: `https://github.com/thalissonvs/pydoll`

Abstract

Pydoll is a Python library designed to simplify interaction with the Chrome DevTools Protocol (CDP). It provides an object-oriented interface for automating and controlling Chromium-based browsers, enabling tasks such as web scraping, automated testing, and performance analysis. Pydoll is built to be asynchronous, leveraging `asyncio` for efficient and non-blocking operations.

Contents

1 Conceptual Overview

Pydoll is structured around the core concept of interacting with a browser instance through the Chrome DevTools Protocol. Here's a breakdown of the key components:

- **Browser Class:** The central class (`Browser`) is responsible for managing the browser process lifecycle. It handles starting, stopping, and configuring the browser. It also provides methods for global browser-level operations like setting download paths and managing cookies.
- **Page Class:** The `Page` class represents a single tab or page within the browser. It offers methods for interacting with the content of a page, such as navigation, taking screenshots, executing JavaScript, and finding elements in the DOM.
- **WebElement Class:** Instances of `WebElement` represent DOM elements on a web page. They provide methods for interacting with elements, such as clicking, sending keys, getting attributes, and retrieving text content.
- **Commands Modules:** Organized into submodules (`browser`, `dom`, `fetch`, `input`, `network`, `page`, `runtime`, `storage`, `target`), these modules contain classes (e.g., `BrowserCommands`, `DomCommands`) that encapsulate the raw CDP commands. These are lower-level building blocks used by the `Browser` and `Page` classes.
- **Events Modules:** Similar to commands, event modules (`browser`, `dom`, `fetch`, `network`, `page`) define classes (e.g., `BrowserEvents`, `DomEvents`) that list the available CDP events. Pydoll provides mechanisms to register callbacks for these events, allowing your application to react to browser activities in real-time.
- **Connection Handling:** The `ConnectionHandler` class manages the WebSocket connection to the browser, sending commands and receiving event messages.
- **Managers:** Manager classes (`BrowserOptionsManager`, `BrowserProcessManager`, `ProxyManager`, `TempDirectoryManager`) handle specific aspects of browser setup and management, such as options processing, process control, proxy configuration, and temporary directory management.

Pydoll aims to abstract away the complexity of the raw CDP, providing a more Pythonic and user-friendly API for browser automation.

2 Basic Usage

2.1 Installation

```
pip install pydoll
```

2.2 Importing the main class

```
from pydoll.browser.chrome import Chrome
from pydoll.browser.options import Options
```

2.3 Starting and stopping a browser

```
import asyncio

async def main():
    async with Chrome() as browser:
```

```
        await browser.start()
        page = await browser.get_page()
        await page.go_to("https://www.example.com")
        print(await page.current_url) # >>> https://www.example.com/
        await browser.stop()

if __name__ == "__main__":
    asyncio.run(main())
```

2.4 Customizing browser options

```
import asyncio
from pydoll.browser.chrome import Chrome
from pydoll.browser.options import Options

async def main():
    options = Options()
    options.add_argument("—headless") # Run in headless mode
    options.binary_location = "/path/to/your/chrome/binary" # Specify Chrome binary path

    async with Chrome(options=options) as browser:
        await browser.start()
        page = await browser.get_page()
        await page.go_to("https://www.example.com")
        print(await page.current_url)
        await browser.stop()

if __name__ == "__main__":
    asyncio.run(main())
```

3 API Reference

`pydoll.browser` Module

`class pydoll.browser.base.Browser`

`class Browser(ABC)`
A class to manage a browser instance for automated interactions.
This class allows users to start and stop a browser, take screenshots, and register event callbacks.

```
def __init__(self, options: Options | None = None, connection_port: int = None
    ):
```

Arguments

- `options` (`Options | None`): Configuration options.
- `connection_port` (`int`): The port to connect to the browser. If `None`, a random port between 9223 and 9322 will be used.

Raises

- `TypeError`: If any of the arguments are of an incorrect type.

Asynchronous Context Manager Methods

```
async def __aenter__(self):
```

Enters the asynchronous context. Returns the `Browser` instance.

```
async def __aexit__(self, exc_type, exc_val, exc_tb):
```

Exits the asynchronous context. Automatically stops the browser and closes the connection.

Methods

```
async def start(self) -> None:
```

Starts the browser process.

Raises

- `BrowserNotRunning`: If the browser fails to start.

```
async def stop(self) -> None:
```

Stops the running browser process.

Raises

- `ValueError`: If the browser is not currently running.
- `BrowserNotRunning`: If the browser is not running.

```
async def set_download_path(self, path: str):
```

Sets the download path for the browser.

Arguments

- `path` (`str`): The path to the download directory.

```
async def get_page_by_id(self, page_id: str) -> Page:
```

Retrieves a `Page` instance by its ID.

Arguments

- `page_id` (`str`): The ID of the page to retrieve.

Returns

- `Page`: The `Page` instance corresponding to the specified ID.

```
async def get_page(self) -> Page:
```

Retrieves a `Page` instance for an existing page in the browser. If no pages are open, a new page will be created.

Returns

- `Page`: The `Page` instance.

```
async def delete_all_cookies(self):
```

Deletes all cookies from the browser.

```
async def set_cookies(self, cookies: list[dict]):
```

Sets cookies in the browser.

Arguments

- `cookies` (`list[dict]`): A list of dictionaries containing the cookie data.

```
async def get_cookies(self) -> list[dict]:
```

Retrieves all cookies from the browser.

Returns

- `list[dict]`: A list of dictionaries containing the cookie data.

```
async def on(self, event_name: str, callback: callable, temporary: bool =
    False) -> int:
```

Registers an event callback for a specific event. This method has a global scope and can be used to listen for events across all pages in the browser. Each `Page` instance also has an `on` method that allows for listening to events on a specific page.

Arguments

- `event_name` (`str`): Name of the event to listen for (e.g., from `PageEvents`, `NetworkEvents`, etc.).
- `callback` (`Callable`): Function to be called when the event occurs.
- `temporary` (`bool`, optional): If `True`, the callback will be automatically removed after it is triggered once. Defaults to `False`.

Returns

- `int`: The ID of the registered callback.

```
async def new_page(self, url: str = '') -> str:
```

Opens a new page in the browser.

Arguments

- `url` (`str`, optional): URL to navigate to in the new page. Defaults to an empty string (new tab page).

Returns

- `str`: The `targetId` of the new page.

```
async def get_targets(self) -> list[dict]:
```

Retrieves the list of open pages in the browser.

Returns

- `list[dict]`: The list of open pages in the browser, each represented as a dictionary with target information.

```
async def get_window_id(self) -> str:
```

Retrieves the ID of the current browser window.

Returns

- `str`: The ID of the current browser window.

```
async def set_window_bounds(self, bounds: dict):
```

Sets the bounds of the specified window.

Arguments

- `bounds` (`dict`): The bounds to set for the window. This should be a dictionary containing keys like `'width'`, `'height'`, `'x'`, `'y'`, and `'windowState'`.

```
async def set_window_maximized(self):
```

Maximizes the specified window.

```
async def set_window_minimized(self):
```

Minimizes the specified window.

```
async def enable_page_events(self):
```

Enables listening for page-related events over the websocket connection globally for the browser.

```
async def enable_network_events(self):
```

Activates listening for network events through the websocket connection globally for the browser.

```
async def enable_fetch_events(self, handle_auth_requests: bool = False,
    resource_type: str = ""):
```

Enables the Fetch domain for intercepting network requests before they are sent globally for the browser.

Arguments

- `handle_auth_requests` (`bool`, optional): Whether to handle authentication requests that require user credentials. Defaults to `False`.
- `resource_type` (`str`, optional): The type of resource to intercept (e.g., `'XHR'`, `'Script'`). If not specified, all requests will be intercepted. Defaults to `''`.

```
async def enable_dom_events(self):
```

Enables DOM-related events for the websocket connection globally for the browser.

```
async def disable_fetch_events(self):
```

Deactivates the Fetch domain, stopping the interception of network requests for the websocket connection globally for the browser.

class pydoll.browser.chrome.Chrome

class Chrome(Browser)
A subclass of `Browser` specifically for managing Google Chrome instances.

```
def __init__(self, options: Options | None = None, connection_port: int | None
    = None):
```

Initializes a `Chrome` browser instance. Inherits from `Browser.__init__`.

Arguments

- `options` (`Optional[Options]`, optional): Browser options. Defaults to `None`.
- `connection_port` (`Optional[int]`, optional): Connection port. Defaults to `None`.

class pydoll.browser.managers.BrowserOptionsManager

Manages browser options initialization and validation.

Static Methods

```
@staticmethod
def initialize_options(options: Options | None) -> Options:
```

Initializes browser options.

Arguments

- `options` (`Options  None|`): Options instance or `None` for default options.

Returns

- `Options`: Initialized options instance.

Raises

- `ValueError`: If `options` is not an instance of `Options`.

```
@staticmethod
def add_default_arguments(options: Options):
```

Adds default arguments to the provided options. Includes `--no-first-run` and `--no-default-browser-check`.

Arguments

- `options` (`Options`): The `Options` instance to modify.

```
@staticmethod
def validate_browser_path(path: str) -> str:
```

Validates the provided browser executable path.

Arguments

- `path` (`str`): Path to the browser executable.

Returns

- `str`: Validated browser path.

Raises

- `ValueError`: If the path does not exist.

class pydoll.browser.managers.BrowserProcessManager

Manages the browser process lifecycle.

```
def __init__(self, process_creator=None):
```

Initializes `BrowserProcessManager`.

Arguments

- `process_creator` (callable, optional): A custom function to create subprocesses, for testing purposes.

Methods

```
def start_browser_process(self, binary_location: str, port: int, arguments:
    list) -> subprocess.Popen:
```

Starts the browser subprocess.

Arguments

- `binary_location` (`str`): Path to the browser executable.
- `port` (`int`): Port for remote debugging.
- `arguments` (`list`): List of browser arguments.

Returns

- `subprocess.Popen`: The process object.

```
def stop_process(self):
```

Terminates the browser process if it is running.

class pydoll.browser.managers.ProxyManager

Manages proxy configurations for the browser.

```
def __init__(self, options):
```

Initializes `ProxyManager`.

Arguments

- `options` (`Options`): Browser options.

Methods

```
def get_proxy_credentials(self) -> tuple[bool, tuple[str, str]]:
```

Configures proxy settings and extracts credentials if present in the `--proxy-server` argument.

Returns

- `tuple[bool, tuple[str, str]]`: A tuple containing:
  - `bool`: `True` if a private proxy is configured (credentials found), `False` otherwise.
  - `tuple[str, str]`: A tuple containing the username and password for the proxy, or `(None, None)` if no credentials were found.

class pydoll.browser.managers.TempDirectoryManager

Manages temporary directories for browser data.

```
def __init__(self, temp_dir_factory=TemporaryDirectory):
```

Initializes `TempDirectoryManager`.

Arguments

- `temp_dir_factory` (callable, optional): A custom factory for creating temporary directories, for testing purposes.

Methods

```
def create_temp_dir(self) -> TemporaryDirectory:
```

Creates a temporary directory for the browser instance.

Returns

- `TemporaryDirectory`: The temporary directory object.

```
def cleanup(self):
```

Deletes all created temporary directories.

class pydoll.browser.options.Options

`class Options()`
A class to manage command-line options for a browser instance.
This class allows the user to specify command-line arguments and the binary location of the browser executable.

```
def __init__(self):
```

Initializes the `Options` instance. Sets up an empty list for command-line arguments and a string for the binary location of the browser.

Properties

```
arguments: list
```

Gets the list of command-line arguments.

Returns

- `list`: A list of command-line arguments added to the options.

`binary_location: str`

Gets the location of the browser binary.

Returns

- `str`: The file path to the browser executable.

Methods

```
def add_argument(self, argument: str):
```

Adds a command-line argument to the options.

Arguments

- `argument` (`str`): The command-line argument to be added.

Raises

- `ValueError`: If the argument is already in the list of arguments.

`pydoll.commands` Module

`pydoll.commands.browser` Module

`class pydoll.commands.browser.BrowserCommands`

BrowserCommands class provides a set of commands to interact with the browser's main functionality based on CDP. These commands allow for managing browser windows, such as closing windows, retrieving window IDs, and adjusting window bounds (size and state).

Class Attributes

- `CLOSE` (`dict`): Command template to close the browser.
- `GET_WINDOW_ID` (`dict`): Command template to get the current window ID.
- `SET_WINDOW_BOUNDS_TEMPLATE` (`dict`): Template for setting window bounds.
- `SET_DOWNLOAD_BEHAVIOR` (`dict`): Template for setting download behavior.

Class Methods

```
@classmethod
def set_download_path(cls, path: str) -> dict:
```

Generates the command to set the download path for the browser.

Arguments

- `path` (`str`): The path to set for downloads.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def close(cls) -> dict:
```

Generates the command to close the browser.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_window_id(cls) -> dict:
```

Generates the command to get the ID of the current window.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_window_bounds(cls, window_id: int, bounds: dict) -> dict:
```

Generates the command to set the bounds of a window.

Arguments

- `window_id` (`int`): The ID of the window to set the bounds for.
- `bounds` (`dict`): The bounds to set for the window, which should include width, height, and optionally x and y coordinates.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_window_maximized(cls, window_id: int) -> dict:
```

Generates the command to maximize a window.

Arguments

- `window_id` (`int`): The ID of the window to maximize.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_window_minimized(cls, window_id: int) -> dict:
```

Generates the command to minimize a window.

Arguments

- `window_id` (`int`): The ID of the window to minimize.

Returns

- `dict`: The command to be sent to the browser.

`pydoll.commands.dom` Module

`class pydoll.commands.dom.DomCommands`

A class to define commands for interacting with the Document Object Model (DOM) using the Chrome DevTools Protocol (CDP). The commands allow for various operations on DOM nodes, such as enabling the DOM domain, retrieving the DOM document, describing nodes, and querying elements.

Class Attributes

- `SelectorType` (`Literal`): A type definition for supported selector types (`By.CSS_SELECTOR`, `By.XPATH`, `By.CLASS_NAME`, `By.ID`, `By.TAG_NAME`).
- `ENABLE` (`dict`): Command template to enable DOM domain events.
- `DOM_DOCUMENT` (`dict`): Command template to get the DOM document.
- `DESCRIBE_NODE_TEMPLATE` (`dict`): Template for describing a DOM node.
- `FIND_ELEMENT_TEMPLATE` (`dict`): Template for finding a single DOM element.
- `FIND_ALL_ELEMENTS_TEMPLATE` (`dict`): Template for finding multiple DOM elements.
- `BOX_MODEL_TEMPLATE` (`dict`): Template for getting the box model of a DOM node.
- `RESOLVE_NODE_TEMPLATE` (`dict`): Template for resolving a DOM node.
- `REQUEST_NODE_TEMPLATE` (`dict`): Template for requesting a DOM node.
- `GET_OUTER_HTML` (`dict`): Template for getting the outer HTML of a DOM node.
- `SCROLL_INTO_VIEW_IF_NEEDED` (`dict`): Template for scrolling a DOM node into view if needed.

Class Methods

```
@classmethod
def scroll_into_view(cls, object_id: str) -> dict:
```

Generates the command to scroll a specific DOM node into view.

Arguments

- `object_id` (`str`): The object ID of the DOM node.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_outer_html(cls, object_id: int) -> dict:
```

Generates the command to get the outer HTML of a DOM node.

Arguments

- `object_id` (`int`): The object ID of the DOM node.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def dom_document(cls) -> dict:
```

Generates the command to get the root DOM node of the current page.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def request_node(cls, object_id: str) -> dict:
```

Generates the command to request a specific DOM node by its object ID.

Arguments

- `object_id` (`str`): The object ID of the DOM node.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def describe_node(cls, object_id: str) -> dict:
```

Generates the command to describe a specific DOM node.

Arguments

- `object_id` (`str`): The object ID of the DOM node.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def box_model(cls, object_id: str) -> dict:
```

Generates the command to get the box model of a specific DOM node.

Arguments

- `object_id` (`str`): The object ID of the DOM node.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def enable_dom_events(cls) -> dict:
```

Generates the command to enable the DOM domain events.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_current_url(cls) -> dict:
```

Generates the command to get the current URL of the page.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def find_element(cls, by: SelectorType, value: str, object_id: str = '') ->
    dict:
```

Generates a command to find a DOM element based on the specified criteria.

Arguments

- `by` (`SelectorType`): The type of selector to use (e.g., `By.CSS_SELECTOR`, `By.XPATH`).
- `value` (`str`): The selector value (e.g., CSS selector string, XPath expression).
- `object_id` (`str`, optional): The object ID of a parent element to search within. If empty, the search is performed on the entire document. Defaults to `''`.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def find_elements(cls, by: SelectorType, value: str, object_id: str = '') ->
    dict:
```

Generates a command to find multiple DOM elements based on the specified criteria.

Arguments

- `by` (`SelectorType`): The type of selector to use (e.g., `By.CSS_SELECTOR`, `By.XPATH`).
- `value` (`str`): The selector value (e.g., CSS selector string, XPath expression).
- `object_id` (`str`, optional): The object ID of a parent element to search within. If empty, the search is performed on the entire document. Defaults to `''`.

Returns

- `dict`: The command to be sent to the browser.

`pydoll.commands.fetch` Module

`class pydoll.commands.fetch.FetchCommands`

A collection of command templates for handling fetch events in the browser.
This class provides a structured way to create and manage commands related to fetch operations intercepted by the Fetch API. Each command corresponds to specific actions that can be performed on fetch requests, such as continuing a fetch request, fulfilling a fetch response, or handling authentication challenges.

Class Attributes

- `CONTINUE_REQUEST` (`dict`): Template for continuing an intercepted fetch request.
- `CONTINUE_REQUEST_WITH_AUTH` (`dict`): Template for continuing a fetch request that requires authentication.
- `DISABLE` (`dict`): Template for disabling fetch interception.
- `ENABLE` (`dict`): Template for enabling fetch interception.
- `FAIL_REQUEST` (`dict`): Template for simulating a failure in a fetch request.
- `FULFILL_REQUEST` (`dict`): Template for fulfilling a fetch request with custom responses.
- `GET_RESPONSE_BODY` (`dict`): Template for retrieving the response body of a fetch request.
- `CONTINUE_RESPONSE` (`dict`): Template for continuing a fetch response for an intercepted request.

Class Methods

```
@classmethod
def continue_request(cls, request_id: str, url: str = '', method: str = '',
    post_data: str = '', headers: dict = {}, intercept_response: bool = False)
    :
```

Creates a command to continue a paused fetch request.

Arguments

- `request_id` (`str`): The ID of the fetch request to continue.
- `url` (`str`, optional): The new URL for the fetch request. Defaults to `''`.
- `method` (`str`, optional): The HTTP method to use (e.g., `` `GET' ``, `` `POST' ``). Defaults to `''`.
- `post_data` (`str`, optional): The body data to send with the fetch request. Defaults to `''`.
- `headers` (`dict`, optional): A dictionary of HTTP headers to include in the fetch request. Defaults to `{}`.
- `intercept_response` (`bool`, optional): Indicates if the response should be intercepted. Defaults to `False`.

Returns

- `dict`: A command template for continuing the fetch request.

```
@classmethod
def continue_request_with_auth(cls, request_id: str, proxy_username: str,
    proxy_password: str):
```

Creates a command to continue a paused fetch request with authentication.

Arguments

- `request_id` (`str`): The ID of the fetch request to continue.
- `proxy_username` (`str`): The username for proxy authentication.
- `proxy_password` (`str`): The password for proxy authentication.

Returns

- `dict`: A command template for continuing the fetch request with authentication.

```
@classmethod
def disable_fetch_events(cls):
```

Creates a command to disable fetch interception.

Returns

- `dict`: A command template for disabling fetch interception.

```
@classmethod
def enable_fetch_events(cls, handle_auth_requests: bool, resource_type: str):
```

Creates a command to enable fetch interception.

Arguments

- `handle_auth_requests` (`bool`): Indicates if authentication requests should be handled.
- `resource_type` (`str`): The type of resource to intercept (e.g., `` `Document' ``, `` `Image' ``).

Returns

- `dict`: A command template for enabling fetch interception.

```
@classmethod
def fail_request(cls, request_id: str, error_reason: str):
```

Creates a command to simulate a failure in a fetch request.

Arguments

- `request_id` (`str`): The ID of the fetch request to fail.
- `error_reason` (`str`): A description of the failure reason.

Returns

- `dict`: A command template for failing the fetch request.

```
@classmethod
def fulfill_request(cls, request_id: str, response_code: int, response_headers
    : dict = {}, binary_response_headers: str = '', body: str = '',
    response_phrase: str = ''):
```

Creates a command to fulfill a fetch request with a custom response.

Arguments

- `request_id` (`str`): The ID of the fetch request to fulfill.
- `response_code` (`int`): The HTTP status code to return.
- `response_headers` (`dict`, optional): A dictionary of response headers. Defaults to `{}`.
- `binary_response_headers` (`str`, optional): Binary response headers. Defaults to `''`.
- `body` (`str`, optional): The body content of the response. Defaults to `''`.
- `response_phrase` (`str`, optional): The response phrase (e.g., `` `OK' ``, `` `Not Found' ``). Defaults to `''`.

Returns

- `dict`: A command template for fulfilling the fetch request.

```
@classmethod
def get_response_body(cls, request_id: str):
```

Creates a command to retrieve the response body of a fetch request.

Arguments

- `request_id` (`str`): The ID of the fetch request to retrieve the body from.

Returns

- `dict`: A command template for getting the response body.

```
@classmethod
def continue_response(cls, request_id: str, response_code: int = '',
    response_headers: dict = {}, binary_response_headers: str = '',
    response_phrase: str = ''):
```

Creates a command to continue a fetch response for an intercepted request.

Arguments

- `request_id` (`str`): The ID of the fetch request to continue the response for.
- `response_code` (`int`, optional): The HTTP status code to send. Defaults to `''`.
- `response_headers` (`dict`, optional): A dictionary of response headers. Defaults to `{}`.
- `binary_response_headers` (`str`, optional): Binary response headers. Defaults to `''`.
- `response_phrase` (`str`, optional): The response phrase (e.g., `` `OK' ``). Defaults to `''`.

Returns

- `dict`: A command template for continuing the fetch response.

`pydoll.commands.input` Module

`class pydoll.commands.input.InputCommands`

A class to define input commands for simulating user interactions with the browser using the Chrome DevTools Protocol (CDP). The commands allow for simulating mouse clicks and keyboard presses.

Class Attributes

- `CLICK_ELEMENT_TEMPLATE` (`dict`): Template for dispatching a mouse event (click).
- `KEY_PRESS_TEMPLATE` (`dict`): Template for dispatching a key event (press).
- `INSERT_TEXT_TEMPLATE` (`dict`): Template for inserting text into an input field.

Class Methods

```
@classmethod
def mouse_press(cls, x: int, y: int) -> dict:
```

Generates the command to simulate pressing the mouse button on a specific location.

Arguments

- `x` (`int`): The x-coordinate of the mouse press.
- `y` (`int`): The y-coordinate of the mouse press.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def mouse_release(cls, x: int, y: int) -> dict:
```

Generates the command to simulate releasing the mouse button.

Arguments

- `x` (`int`): The x-coordinate of the mouse release.
- `y` (`int`): The y-coordinate of the mouse release.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def key_press(cls, char: str) -> dict:
```

Generates the command to simulate pressing a key on the keyboard.

Arguments

- `char` (`str`): The character to be pressed.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def insert_text(cls, text: str) -> dict:
```

Generates the command to insert text into an input field.

Arguments

- `text` (`str`): The text to be inserted.

Returns

- `dict`: The command to be sent to the browser.

`pydoll.commands.network` Module

`class pydoll.commands.network.NetworkCommands`

This class encapsulates the network commands of the Chrome DevTools Protocol (CDP). CDP allows developers to interact with the Chrome browser's internal mechanisms to inspect, manipulate, and monitor network operations, which can be invaluable for debugging web applications, testing network behaviors, and optimizing performance.

Class Attributes

- `CLEAR_BROWSER_CACHE` (`dict`): Command template to clear browser cache.
- `CLEAR_BROWSER_COOKIES` (`dict`): Command template to clear browser cookies.
- `DELETE_COOKIES_TEMPLATE` (`dict`): Template for deleting specific cookies.
- `DISABLE` (`dict`): Command template to disable network events.
- `ENABLE` (`dict`): Command template to enable network events.
- `GET_COOKIES_TEMPLATE` (`dict`): Template for getting cookies.
- `GET_REQUEST_POST_DATA_TEMPLATE` (`dict`): Template for getting request POST data.
- `GET_RESPONSE_BODY_TEMPLATE` (`dict`): Template for getting response body.
- `SET_CACHE_DISABLED_TEMPLATE` (`dict`): Template for setting cache disabled state.
- `SET_COOKIE_TEMPLATE` (`dict`): Template for setting a cookie.
- `SET_COOKIES_TEMPLATE` (`dict`): Template for setting multiple cookies.
- `SET_EXTRA_HTTP_HEADERS_TEMPLATE` (`dict`): Template for setting extra HTTP headers.
- `SET_USERAGENT_OVERRIDE_TEMPLATE` (`dict`): Template for setting user agent override.
- `GET_ALL_COOKIES` (`dict`): Command template to get all cookies.
- `SEARCH_IN_RESPONSE_TEMPLATE` (`dict`): Template for searching in response body.
- `SET_BLOCKED_URLS` (`dict`): Template for setting blocked URLs.

Class Methods

```
@classmethod
def clear_browser_cache(cls):
```

Command to clear the browser's cache.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def clear_browser_cookies(cls):
```

Command to clear all cookies stored in the browser.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def delete_cookies(cls, name: str, url: str = ''):
```

Creates a command to delete a specific cookie by name.

Arguments

- `name` (`str`): The name of the cookie to delete.
- `url` (`str`, optional): The URL associated with the cookie. If specified, only the cookie matching both the name and URL will be deleted.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def disable_network_events(cls):
```

Command to disable network event notifications.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def enable_network_events(cls):
```

Command to enable network event notifications.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_cookies(cls, urls: list[str] = []):
```

Creates a command to retrieve cookies from specified URLs.

Arguments

- `urls` (`list[str]`, optional): A list of URLs for which to retrieve cookies. If not provided, cookies from all URLs will be fetched.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_request_post_data(cls, request_id: str):
```

Creates a command to retrieve POST data associated with a specific request.

Arguments

- `request_id` (`str`): The unique identifier of the network request whose POST data is to be retrieved.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_response_body(cls, request_id: str):
```

Creates a command to retrieve the body of a response for a specific request.

Arguments

- `request_id` (`str`): The unique identifier of the request for which the response body is to be fetched.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_cache_disabled(cls, cache_disabled: bool):
```

Creates a command to enable or disable the browser cache.

Arguments

- `cache_disabled` (`bool`): Set to `True` to disable caching, or `False` to enable it.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_cookie(cls, name: str, value: str, url: str = ''):
```

Creates a command to set a specific cookie.

Arguments

- `name` (`str`): The name of the cookie.
- `value` (`str`): The value of the cookie.
- `url` (`str`, optional): The URL associated with the cookie. If provided, the cookie will be valid for this URL only.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_cookies(cls, cookies: list[dict]):
```

Creates a command to set multiple cookies at once.

Arguments

- `cookies` (`list[dict]`): A list of dictionaries, each representing a cookie with its properties (name, value, url, etc.).

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_extra_http_headers(cls, headers: dict):
```

Creates a command to set additional HTTP headers for subsequent network requests.

Arguments

- `headers` (`dict`): A dictionary of headers to include in all future requests.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_useragent_override(cls, user_agent: str):
```

Creates a command to override the user agent string used in network requests.

Arguments

- `user_agent` (`str`): The user agent string to set for future network requests.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_all_cookies(cls):
```

Command to retrieve all cookies stored in the browser.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def search_in_response(cls, request_id: str, query: str, case_sensitive: bool
    = False, is_regex: bool = False):
```

Creates a command to search for a specific query in the response body of a network request.

Arguments

- `request_id` (`str`): The unique identifier of the request to search within.
- `query` (`str`): The string to search for within the response body.
- `case_sensitive` (`bool`, optional): Whether the search should be case sensitive. Defaults to `False`.
- `is_regex` (`bool`, optional): Whether the query should be treated as a regular expression. Defaults to `False`.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_blocked_urls(cls, urls: list[str]):
```

Creates a command to block specific URLs from being requested by the browser.

Arguments

- `urls` (`list[str]`): A list of URL patterns to block. The browser will not make requests to any URLs matching these patterns.

Returns

- `dict`: The command to be sent to the browser.

`pydoll.commands.page` Module

`class pydoll.commands.page.PageCommands`

PageCommands class provides a set of commands to interact with the Page domain of the Chrome DevTools Protocol (CDP). These commands enable users to perform operations related to web pages, such as capturing screenshots, navigating to URLs, refreshing pages, printing to PDF, and enabling the Page domain.

Class Attributes

- `SCREENSHOT_TEMPLATE` (`dict`): Template for capturing a screenshot.
- `GO_TO_TEMPLATE` (`dict`): Template for navigating to a URL.
- `REFRESH_TEMPLATE` (`dict`): Template for refreshing the page.
- `PRINT_TO_PDF_TEMPLATE` (`dict`): Template for printing to PDF.
- `ENABLE_PAGE` (`dict`): Command template to enable Page domain events.
- `DISABLE_PAGE` (`dict`): Command template to disable Page domain events.
- `SET_DOWNLOAD_BEHAVIOR` (`dict`): Template for setting download behavior on page level.
- `HANDLE_DIALOG` (`dict`): Template for handling JavaScript dialogs.
- `CLOSE` (`dict`): Command template to close the page.

Class Methods

```
@classmethod
def handle_dialog(cls, accept: bool = True) -> dict:
```

Generates the command to handle a JavaScript dialog.

Arguments

- `accept` (`bool`, optional): Whether to accept the dialog. If `True`, the dialog will be accepted. If `False`, the dialog will be dismissed. Defaults to `True`.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_download_path(cls, path: str) -> dict:
```

Generates the command to set the download path for the browser at the page level.

Arguments

- `path` (`str`): The path where the downloaded files should be saved.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def screenshot(cls, fmt: str = 'jpeg', quality: int = 100, clip: dict = None)
    -> dict:
```

Generates the command to capture a screenshot of the current page.

Arguments

- `fmt` (`str`, optional): The format of the image to be captured. Can be `` `png' `` or `` `jpeg' ``. Defaults to `'jpeg'`.
- `quality` (`int`, optional): The quality of the image to be captured, applicable only if the format is `` `jpeg' ``. Value should be between 0 (lowest quality) and 100 (highest quality). Defaults to `100`.
- `clip` (`dict`, optional): A dictionary defining the clipping region for the screenshot. It should contain keys `` `x' ``, `` `y' ``, `` `width' ``, `` `height' ``, and `` `scale' ``. Defaults to `None`.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def go_to(cls, url: str) -> dict:
```

Generates the command to navigate to a specific URL.

Arguments

- `url` (`str`): The URL to navigate to. It should be a valid URL format.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def refresh(cls, ignore_cache: bool = False) -> dict:
```

Generates the command to refresh the current page.

Arguments

- `ignore_cache` (`bool`, optional): Whether to ignore the cache when refreshing. If `True`, the cached resources will not be used. Defaults to `False`.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def print_to_pdf(cls, scale: int = 1, paper_width: float = 8.5, paper_height:
    float = 11) -> dict:
```

Generates the command to print the current page to a PDF.

Arguments

- `scale` (`int`, optional): The scale of the page to print. Default is 1 (100%). Defaults to `1`.
- `paper_width` (`float`, optional): The width of the paper to print on, in inches. Default is 8.5 inches. Defaults to `8.5`.
- `paper_height` (`float`, optional): The height of the paper to print on, in inches. Default is 11 inches. Defaults to `11`.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def enable_page(cls) -> dict:
```

Generates the command to enable the Page domain.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def disable_page(cls) -> dict:
```

Generates the command to disable the Page domain.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def close(cls) -> dict:
```

Generates the command to close the current page.

Returns

- `dict`: The command to be sent to the browser.

pydoll.commands.runtime Module

class pydoll.commands.runtime.RuntimeCommands

Class Attributes

- `EVALUATE_TEMPLATE` (`dict`): Template for evaluating JavaScript code.
- `CALL_FUNCTION_ON_TEMPLATE` (`dict`): Template for calling a function on a specific object.
- `GET_PROPERTIES` (`dict`): Template for getting properties of an object.

Class Methods

```
@classmethod
def get_properties(cls, object_id: str) -> dict:
```

Generates the command to get the properties of a specific object.

Arguments

- `object_id` (`str`): The object ID.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def call_function_on(cls, object_id: str, function_declaration: str,
    return_by_value: bool = False) -> dict:
```

Generates the command to call a function on a specific object.

Arguments

- `object_id` (`str`): The object ID.
- `function_declaration` (`str`): The JavaScript function declaration as a string.
- `return_by_value` (`bool`, optional): Whether to return the result by value. Defaults to `False`.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def evaluate_script(cls, expression: str) -> dict:
```

Generates the command to evaluate JavaScript code.

Arguments

- `expression` (`str`): The JavaScript expression to evaluate.

Returns

- `dict`: The command to be sent to the browser.

`pydoll.commands.storage` Module

`class pydoll.commands.storage.StorageCommands`

Class Attributes

- `CLEAR_COOKIES` (`dict`): Command template to clear all cookies.
- `SET_COOKIES` (`dict`): Command template to set multiple cookies.
- `GET_COOKIES` (`dict`): Command template to get cookies.

Class Methods

```
@classmethod
def clear_cookies(cls) -> dict:
```

Generates the command to clear all cookies.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def set_cookies(cls, cookies: list) -> dict:
```

Generates the command to set multiple cookies.

Arguments

- `cookies` (`list`): A list of cookie dictionaries.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_cookies(cls) -> dict:
```

Generates the command to get cookies.

Returns

- `dict`: The command to be sent to the browser.

`pydoll.commands.target` Module

`class pydoll.commands.target.TargetCommands`

Class Attributes

- `ACTIVATE_TARGET` (`dict`): Command template to activate a target.
- `ATTACH_TO_TARGET` (`dict`): Command template to attach to a target.
- `CLOSE_TARGET` (`dict`): Command template to close a target.
- `CREATE_TARGET` (`dict`): Command template to create a new target (page/tab).
- `GET_TARGETS` (`dict`): Command template to get all targets.
- `GET_TARGET_INFO` (`dict`): Command template to get target information.

Class Methods

```
@classmethod
def activate_target(cls, target_id: str) -> dict:
```

Generates the command to activate a target.

Arguments

- `target_id` (`str`): The target ID.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def attach_to_target(cls, target_id: str) -> dict:
```

Generates the command to attach to a target.

Arguments

- `target_id` (`str`): The target ID.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def close_target(cls, target_id: str) -> dict:
```

Generates the command to close a target.

Arguments

- `target_id` (`str`): The target ID.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def create_target(cls, url: str) -> dict:
```

Generates the command to create a new target (page/tab).

Arguments

- `url` (`str`): The URL to open in the new target.

Returns

- `dict`: The command to be sent to the browser.

```
@classmethod
def get_targets(cls) -> dict:
```

Generates the command to get all targets.

Returns

- `dict`: The command to be sent to the browser.

`pydoll.connection` Module

`class pydoll.connection.connection.ConnectionHandler`

`class ConnectionHandler()`
A class to handle WebSocket connections for browser automation.
This class manages the connection to the browser and the associated page, providing methods to execute commands and register event callbacks.

```
def __init__(self, connection_port: int, page_id: str = 'browser',
    ws_address_resolver: Callable[[int], str] = get_browser_ws_address,
    ws_connector: Callable = websockets.connect):
```

Initializes the ConnectionHandler instance.

Arguments

- `connection_port` (`int`): The port to connect to the browser.
- `page_id` (`str`, optional): The page ID. Defaults to `'browser'` for browser-level connection.
- `ws_address_resolver` (`Callable[[int], str]`, optional): A function to resolve the WebSocket address. Defaults to `get_browser_ws_address`.
- `ws_connector` (`Callable`, optional): A function to establish the WebSocket connection. Defaults to `websockets.connect`.

Properties

```
network_logs: list
```

Returns the list of network logs received through this connection.

```
dialog: dict
```

Returns the current dialog information, if a JavaScript dialog is open.

Methods

```
async def ping(self) -> bool:
```

Sends a ping message to the browser to check the connection.

Returns

- `bool`: `True` if the ping was successful, `False` otherwise.

```
async def execute_command(self, command: dict, timeout: int = 10) -> dict:
```

Sends a command to the browser and awaits its response.

Arguments

- `command` (`dict`): The command to send, structured as a dictionary.
- `timeout` (`int`, optional): Time in seconds to wait for a response. Defaults to `10`.

Returns

- `dict`: The response from the browser.

Raises

- `InvalidCommand`: If the command is not a dictionary.
- `TimeoutError`: If the command execution exceeds the timeout.
- `websockets.ConnectionClosed`: If the WebSocket connection is closed unexpectedly.

```
async def register_callback(self, event_name: str, callback: Callable,
    temporary: bool = False) -> int:
```

Registers a callback function to be executed when a specific event occurs.

Arguments

- `event_name` (`str`): The name of the event to listen for (e.g., `` `Page.loadEventFired' ``).
- `callback` (`Callable`): The asynchronous or synchronous function to call when the event is received.
- `temporary` (`bool`, optional): If `True`, the callback will be removed after it is triggered once. Defaults to `False`.

Returns

- `int`: The ID of the registered callback.

```
async def remove_callback(self, callback_id: int) -> bool:
```

Removes a registered callback by its ID.

Arguments

- `callback_id` (`int`): The ID of the callback to remove.

Returns

- `bool`: `True` if the callback was successfully removed, `False` if the ID was not found.

```
async def clear_callbacks(self):
```

Clears all registered event callbacks.

```
async def close(self):
```

Closes the WebSocket connection and clears all event callbacks.

Asynchronous Context Manager Methods

```
async def __aenter__(self):
```

Enters the asynchronous context. Returns the `ConnectionHandler` instance.

```
async def __aexit__(self, exc_type, exc_val, exc_tb):
```

Exits the asynchronous context. Automatically closes the connection.

class pydoll.connection.managers.CommandManager

Manages pending commands and their futures.

Methods

```
def create_command_future(self, command: dict) -> asyncio.Future:
```

Creates a future for a command and stores it, associating it with a command ID.

Arguments

- `command` (`dict`): The command dictionary (will be assigned an `id' in this method).

Returns

- `asyncio.Future`: The future object that will be resolved when the response for this command is received.

```
def resolve_command(self, response_id: int, result: str):
```

Resolves a pending command's future with the received result, based on the response ID.

Arguments

- `response_id` (`int`): The ID of the command response.
- `result` (`str`): The JSON string response from the browser.

```
def remove_pending_command(self, command_id: int):
```

Removes a pending command without resolving it (useful for timeouts).

Arguments

- `command_id` (`int`): ID of the command to be removed.

class pydoll.connection.managers.EventsHandler

Manages event callbacks, processes events, and maintains network logs.

Properties

```
network_logs: list
```

Returns the list of network logs.

`dialog: dict`

Returns the dialog information.

Methods

```
def register_callback(self, event_name: str, callback: Callable, temporary:
    bool = False) -> int:
```

Registers a callback for a specific event type.

Arguments

- `event_name` (`str`): The name of the event to listen for.
- `callback` (`Callable`): The callback function to register.
- `temporary` (`bool`, optional): Whether the callback is temporary (removed after first trigger). Defaults to `False`.

Returns

- `int`: ID of the registered callback.

Raises

- `InvalidCallback`: If the callback is not callable.

```
def remove_callback(self, callback_id: int) -> bool:
```

Removes a callback by its ID.

Arguments

- `callback_id` (`int`): ID of the callback to remove.

Returns

- `bool`: `True` if callback was removed, `False` if ID not found.

```
def clear_callbacks(self):
```

Resets all registered callbacks.

```
async def process_event(self, event_data: dict):
```

Processes a received event and triggers corresponding callbacks.

Arguments

- `event_data` (`dict`): Event data in dictionary format.

---

pydoll.constants Module

class pydoll.constants.By(str, Enum)

class By(str, Enum)
An enumeration of element selector strategies.

Enum Members

- CSS_SELECTOR ('css')
- XPATH ('xpath')
- CLASS_NAME ('class_name')
- ID ('id')
- TAG_NAME ('tag_name')

class pydoll.constants.Scripts

class Scripts()
A class containing JavaScript code snippets (scripts) used for DOM interaction.

Class Attributes

- ELEMENT_VISIBLE (**str**): JavaScript to check if an element is visible.
- ELEMENT_ON_TOP (**str**): JavaScript to check if an element is on top.
- CLICK (**str**): JavaScript to simulate a click on an element.
- CLICK_OPTION_TAG (**str**): JavaScript for clicking <option> tags within <select>.
- BOUNDS (**str**): JavaScript to get element bounding rectangle as JSON.
- FIND_RELATIVE_XPATH_ELEMENT (**str**): JavaScript to find an element by XPath relative to another element.
- FIND_XPATH_ELEMENT (**str**): JavaScript to find an element by XPath in the document.
- FIND_RELATIVE_XPATH_ELEMENTS (**str**): JavaScript to find elements by XPath relative to another element (multiple results).
- FIND_XPATH_ELEMENTS (**str**): JavaScript to find elements by XPath in the document (multiple results).
- QUERY_SELECTOR (**str**): JavaScript to query a single element using CSS selector.
- RELATIVE_QUERY_SELECTOR (**str**): JavaScript to query a single element using CSS selector relative to another element.
- QUERY_SELECTOR_ALL (**str**): JavaScript to query multiple elements using CSS selector.
- RELATIVE_QUERY_SELECTOR_ALL (**str**): JavaScript to query multiple elements using CSS selector relative to another element.

---

pydoll.element Module

class pydoll.element.WebElement

class WebElement(FindElementsMixin)
Represents a DOM element on a web page.

```
def __init__(self, object_id: str, connection_handler: ConnectionHandler,
    method: str = None, selector: str = None, attributes_list: list = []):
```

Initializes the `WebElement` instance.

Arguments

- `object_id` (`str`): The object ID of the DOM node from CDP.
- `connection_handler` (`ConnectionHandler`): The connection handler instance.
- `method` (`str`, optional): The method used to search for the element (e.g., `By.CSS_SELECTOR`). Defaults to `None`.
- `selector` (`str`, optional): The selector value used to find the element. Defaults to `None`.
- `attributes_list` (`list`, optional): A list of attributes and values for the element, obtained from CDP. Defaults to `[]`.

Properties

`value: str`

Retrieves the `value` attribute of the element.

Returns

- `str`: The value of the element, or `None` if not found.

`class_name: str`

Retrieves the `class` attribute (mapped to `class_name` property) of the element.

Returns

- `str`: The class name of the element, or `None` if not found.

`id: str`

Retrieves the `id` attribute of the element.

Returns

- `str`: The ID of the element, or `None` if not found.

`is_enabled: bool`

Retrieves the enabled status of the element, based on the presence of the `` `disabled' `` attribute.

Returns

- `bool`: `True` if the element is enabled (not disabled), `False` otherwise.

`async bounds: list`

Asynchronously retrieves the bounding box of the element using CDP's DOM API.

Returns

- `list`: The bounding box coordinates of the element.

`async inner_html: str`

Asynchronously retrieves the inner HTML of the element.

Returns

- `str`: The inner HTML of the element.

Methods

```
async def get_bounds_using_js(self) -> list:
```

Retrieves the bounding box of the element using JavaScript execution.

Returns

- `list`: The bounding box coordinates of the element.

```
async def get_screenshot(self, path: str):
```

Takes a screenshot of the element and saves it to the specified path.

Arguments

- `path` (`str`): The file path to save the screenshot to.

```
async def get_element_text(self) -> str:
```

Retrieves the text content of the element.

Returns

- `str`: The text content of the element, stripped of HTML tags.

```
def get_attribute(self, name: str) -> str:
```

Retrieves the value of a specific attribute of the element.

Arguments

- `name` (`str`): The name of the attribute to retrieve.

Returns

- `str`: The value of the attribute, or `None` if the attribute is not present.

```
async def scroll_into_view(self):
```

Scrolls the element into view within the browser window.

```
async def click_using_js(self):
```

Clicks the element using JavaScript execution. This is useful for elements that are difficult to click directly due to overlapping elements.

Raises

- `ElementNotVisible`: If the element is not visible.
- `ElementNotInteractable`: If the element is not interactable.

```
async def click(self, x_offset: int = 0, y_offset: int = 0):
```

Clicks the element by simulating mouse events at the element's center, with optional offsets.

Arguments

- `x_offset` (`int`, optional): Horizontal offset from the element's center for the click position. Defaults to `0`.
- `y_offset` (`int`, optional): Vertical offset from the element's center for the click position. Defaults to `0`.

Raises

- `ElementNotVisible`: If the element is not visible.

```
async def click_option_tag(self):
```

Clicks an `<option>` tag element, specifically designed for `<select>` dropdowns.

```
async def send_keys(self, text: str):
```

Sends a sequence of keys to the element, simulating typing.

Arguments

- `text` (`str`): The text to send.

```
async def type_keys(self, text: str):
```

Types text into the element in a more realistic manner, sending keys one by one with a small delay.

Arguments

- `text` (**str**): The text to type.

---

pydoll.events Module

pydoll.events.browser Module

class pydoll.events.browser.BrowserEvents

A class to define the browser events available through the Chrome DevTools Protocol (CDP). These events allow for monitoring specific actions and states within the browser, such as downloads.

Class Attributes

- `DOWNLOAD_PROGRESS` (**str**): Event triggered when download progress updates.
- `DOWNLOAD_WILL_BEGIN` (**str**): Event triggered when a download is about to start.

pydoll.events.dom Module

class pydoll.events.dom.DomEvents

A class to define the DOM events available through the Chrome DevTools Protocol (CDP). These events allow for monitoring changes and updates within the Document Object Model (DOM) of a web page, enabling developers to react to specific modifications and interactions with the DOM elements.

Class Attributes

- `ATTRIBUTE_MODIFIED` (**str**): Event triggered when an attribute of a DOM node is modified.
- `ATTRIBUTE_REMOVED` (**str**): Event triggered when an attribute of a DOM node is removed.
- `CHARACTER_DATA_MODIFIED` (**str**): Event triggered when the character data of a DOM node is modified.
- `CHILD_NODE_COUNT_UPDATED` (**str**): Event triggered when the number of child nodes of a DOM node is updated.
- `CHILD_NODE_INSERTED` (**str**): Event triggered when a new child node is inserted into a DOM node.
- `CHILD_NODE_REMOVED` (**str**): Event triggered when a child node is removed from a DOM node.
- `DOCUMENT_UPDATED` (**str**): Event triggered when the DOM document is updated.
- `SCROLLABLE_FLAG_UPDATED` (**str**): Event triggered when the scrollable flag of a DOM node is updated.
- `SHADOW_ROOT_POPPED` (**str**): Event triggered when a shadow root is popped from the stack.
- `SHADOW_ROOT_PUSHED` (**str**): Event triggered when a shadow root is pushed onto the stack.
- `TOP_LAYER_ELEMENTS_UPDATED` (**str**): Event triggered when the top layer elements in the DOM are updated.

pydoll.events.fetch Module

class pydoll.events.fetch.FetchEvents

A class to define the Fetch events available through the Chrome DevTools Protocol (CDP). These events are related to the management of network requests, allowing developers to intercept, modify, and monitor HTTP requests and responses made by the browser.

Class Attributes

- AUTH_REQUIRED (str): Event triggered when authentication is required for a network request.
- REQUEST_PAUSED (str): Event triggered when a network request is paused.

pydoll.events.network Module

class pydoll.events.network.NetworkEvents

A class that defines constants for various network-related events. These constants can be used to identify and handle network interactions in applications, particularly in event-driven architectures or APIs that monitor network activity.

Class Attributes

- DATA_RECEIVED (str): Event triggered when data is received over the network.
- EVENT_SOURCE_MESSAGE_RECEIVED (str): Event fired when a message is received from an EventSource.
- LOADING_FAILED (str): Event that indicates a failure in loading a network resource.
- LOADING_FINISHED (str): Event fired when a network loading operation is completed.
- REQUEST_SERVED_FROM_CACHE (str): Event indicating that a network request was fulfilled from the cache.
- REQUEST_WILL_BE_SENT (str): Event triggered just before a network request is sent.
- RESPONSE_RECEIVED (str): Event that indicates a response has been received from a network request.
- WEB_SOCKET_CLOSED (str): Event that occurs when a WebSocket connection has been closed.
- WEB_SOCKET_CREATED (str): Event fired when a new WebSocket connection is established.
- WEB_SOCKET_FRAME_ERROR (str): Event indicating that there was an error with a frame in a WebSocket communication.
- WEB_SOCKET_FRAME_RECEIVED (str): Event fired when a frame is received through a WebSocket.
- WEB_SOCKET_FRAME_SENT (str): Event representing a frame that has been sent through a WebSocket.
- WEB_TRANSPORT_CLOSED (str): Event indicating that a web transport connection has been closed.
- WEB_TRANSPORT_CONNECTION_ESTABLISHED (str): Event fired when a web transport connection is successfully established.
- WEB_TRANSPORT_CREATED (str): Event that signifies that a new web transport connection has been created.
- POLICY_UPDATED (str): Event that indicates that the network policy has been updated.
- REQUEST_INTERCEPTED (str): Event fired when a network request has been intercepted.

pydoll.events.page Module

class pydoll.events.page.PageEvents

A class that defines constants for various page-related events. These constants represent significant events in the lifecycle of a web page, particularly in the context of web automation, testing, or monitoring.

Class Attributes

- `PAGE_LOADED` (`str`): Event triggered when the page has fully loaded.
- `DOM_CONTENT_LOADED` (`str`): Event fired when the DOMContentLoaded event is fired.
- `FILE_CHOOSER_OPENED` (`str`): Event indicating that a file chooser dialog has been opened.
- `FRAME_ATTACHED` (`str`): Event that occurs when a frame is attached to the page.
- `FRAME_DETACHED` (`str`): Event triggered when a frame is detached from the page.
- `FRAME_NAVIGATED` (`str`): Event that indicates a frame has been navigated to a new URL.
- `JS_DIALOG_CLOSED` (`str`): Event fired when a JavaScript dialog (such as an alert or confirmation) is closed.
- `JS_DIALOG_OPENING` (`str`): Event triggered when a JavaScript dialog is about to open.
- `LIFECYCLE_EVENT` (`str`): Event representing a generic lifecycle event for the page.
- `WINDOW_OPENED` (`str`): Event that indicates a new window has been opened.
- `DOCUMENT_OPENED` (`str`): Event that signifies a new document has been opened in the page.
- `FRAME_STARTED_LOADING` (`str`): Event triggered when a frame starts loading content.
- `FRAME_STOPPED_LOADING` (`str`): Event that indicates a frame has stopped loading content.
- `DOWNLOAD_PROGRESS` (`str`): Event fired to indicate progress on a download operation.
- `DOWNLOAD_WILL_BEGIN` (`str`): Event that occurs when a download is about to start.
- `NAVIGATED_WITHIN_DOCUMENT` (`str`): Event that indicates navigation within the same document.

`pydoll.exceptions` Module

Exceptions

- `ConnectionFailed`: Exception raised when connection to the browser fails.
- `InvalidCommand`: Exception raised when an invalid command is provided.
- `InvalidCallback`: Exception raised when an invalid callback is provided.
- `NetworkError`: Exception raised when a network error occurs.
- `InvalidResponse`: Exception raised when an invalid response is received.
- `ReconnectionFailed`: Exception raised when reconnection to the browser fails.
- `ResendCommandFailed`: Exception raised when resending a command fails.
- `BrowserNotRunning`: Exception raised when the browser is not running.
- `ElementNotFound`: Exception raised when an element is not found.
- `ClickIntercepted`: Exception raised when a click is intercepted.
- `ElementNotVisible`: Exception raised when an element is not visible.
- `ElementNotInteractable`: Exception raised when an element is not interactable.
- `InvalidFileExtension`: Exception raised when an invalid file extension is provided for file operations.

---

pydoll.mixins Module

pydoll.mixins.find_elements Module

class pydoll.mixins.find_elements.FindElementsMixin

Mixin class providing methods for finding elements within a Page or WebElement.

Methods

```
async def wait_element(self, by: DomCommands.SelectorType, value: str, timeout
    : int = 10, raise_exc: bool = True) -> WebElement | None:
```

Waits for an element to be present in the DOM.

Arguments

- by (SelectorType): The type of selector to use (e.g., By.CSS_SELECTOR, By.XPATH).
- value (str): The selector value.
- timeout (int, optional): Time in seconds to wait for the element. Defaults to 10.
- raise_exc (bool, optional): Whether to raise TimeoutError if the element is not found within the timeout. Defaults to True.

Returns

- WebElement None|: The WebElement instance if found, None if not found and raise_exc is False.

Raises

- TimeoutError: If the element is not found within the timeout and raise_exc is True.

```
async def find_element(self, by: DomCommands.SelectorType, value: str,
    raise_exc: bool = True) -> WebElement | None:
```

Finds a single element on the current page or within the current element context using the specified selector.

Arguments

- by (SelectorType): The type of selector to use (e.g., By.CSS_SELECTOR, By.XPATH).
- value (str): The selector value.
- raise_exc (bool, optional): Whether to raise ElementNotFound exception if element is not found. Defaults to True.

Returns

- WebElement None|: The WebElement instance if found, None if not found and raise_exc is False.

Raises

- ElementNotFound: If the element is not found and raise_exc is True.

```
async def find_elements(self, by: DomCommands.SelectorType, value: str,
    raise_exc: bool = True) -> list[WebElement]:
```

Finds all elements on the current page or within the current element context using the specified selector.

Arguments

- `by` (`SelectorType`): The type of selector to use (e.g., `By.CSS_SELECTOR`, `By.XPATH`).
- `value` (`str`): The selector value.
- `raise_exc` (`bool`, optional): Whether to raise `ElementNotFound` exception if no elements are found. Defaults to `True`.

Returns

- `list[WebElement]`: A list of `WebElement` instances found, or an empty list if no elements are found and `raise_exc` is `False`.

Raises

- `ElementNotFound`: If no elements are found and `raise_exc` is `True`.

`pydoll.utils` Module

`pydoll.utils.decode_image_to_bytes(image: str) -> bytes`

Decodes a base64 image string to bytes.

Arguments

- `image` (`str`): The base64 image string to decode.

Returns

- `bytes`: The decoded image as bytes.

`pydoll.utils.get_browser_ws_address(port: int) -> str`

Fetches the WebSocket address for the browser instance.

Arguments

- `port` (`int`): The port number of the Chrome DevTools instance.

Returns

- `str`: The WebSocket address for the browser.

Raises

- `NetworkError`: If the address cannot be fetched due to network errors.
- `InvalidResponse`: If the response from the browser is invalid or does not contain the WebSocket address.