Great! I’ll create a concise yet comprehensive guide to Playwright’s sync_api that focuses on the features and arguments most useful for your web scraping node—especially for scraping images, videos, and metadata like titles, captions, alt text, and credits.

I'll organize it around topics like browser and page control, DOM interaction, attribute extraction, lazy loading handling, authentication, and navigating subpages. This will serve as a reference you can pass to an LLM.

I’ll get started and let you know when it’s ready.

Python Playwright sync_api Reference Guide for Scraping Images & Metadata

Browser Control (Launching Browser & Contexts)

- **Launching a Browser:** Use the sync_playwright() context manager to initialize Playwright and launch a browser. For example, launch Chromium in headless mode (no GUI) or headful mode (for debugging):
- from playwright.sync_api import sync_playwright
-
- with sync_playwright() as pw:
-   browser = pw.chromium.launch(headless=True) # headless False to see the browser
-   page = browser.new_page()
-   # ... your code ...
-   browser.close()

Key options: headless (bool, default True), slow_mo (ms delay for each action, useful for debugging), devtools (open devtools when headful).

- **Browser Contexts:** A Browser **Context** is like an incognito browser session. Multiple contexts can run in one Browser instance, each with isolated cookies/storage (BrowserContext | Playwright Python). Use browser.new_context(**options) to create one. If you call browser.new_page(), Playwright will create a new context under the hood for that page (isolating it from others).
  - **Context Options:** viewport={"width": 1280, "height": 720} (control resolution), user_agent="Custom UA" (masquerade as a specific device or browser), locale="en-US", timezone_id="UTC", etc.

  - **Persistent Context:** If you need to preserve login/session across runs, use pw.chromium.launch_persistent_context(user_data_dir="path/to/profile") instead of browser.new_context(). This uses a profile directory on disk to save cookies, localStorage, etc.
  - **Pages:** Once you have a context, open pages with context.new_page(). All pages in a context share that session's cookies and storage. For example:
  - context = browser.new_context()
  - page1 = context.new_page()
  - page2 = context.new_page() # shares cookies with page1
- **Closing:** Call browser.close() to close the browser (this closes all contexts and pages). You can also close individual contexts with context.close() when done (BrowserContext | Playwright Python).

Page Navigation (Loading Websites and Handling Navigation)

- **Navigating to a URL:** Use page.goto(url, timeout=ms, wait_until="load"). This loads the page and waits for the load event by default. Example:
- response = page.goto("https://example.com", timeout=30000, wait_until="domcontentloaded")

Here wait_until could be "load", "domcontentloaded", or "networkidle":

  - "load" waits for the full load (HTML, scripts, CSS, images) (Page | Playwright Python).
  - "domcontentloaded" waits for initial HTML DOM to be ready (faster) (Page | Playwright Python).
  - "networkidle" waits for network to be idle (no requests for ~500 ms) (Page | Playwright Python) – useful for SPAs but *discouraged for testing* (Page | Playwright Python).

- **Controlling Timeouts:** By default, navigation and actions timeout after 30s. You can override per call (timeout=5000) or globally:
- page.set_default_timeout(10000)   # 10s for actions (like click, selectors)
- page.set_default_navigation_timeout(20000)   # 20s for page.goto or wait_for_navigation

- **After Navigation:** It's good practice to wait for critical elements to load. For example, wait for a specific element that indicates the page is ready:
- page.goto("https://reddit.com/r/Art")
- page.wait_for_selector("div#siteTable", state="visible") # sample selector

This ensures the main content container is visible before proceeding.

- **Page URL and Title:** Access page.url for current URL (which may change after navigation or redirects) and use page.title() to get the <title> text of the page.
- **Going Back/Forward:** Use page.go_back() and page.go_forward() to navigate in history. These return a Response or None if no page in that direction.
- **Reloading:** page.reload() will refresh the page, similar to goto(page.url).

Selectors and DOM Access (Finding Elements)

Playwright offers **selectors** to find elements in the DOM. You can use:

- **CSS Selectors:** Default way to locate elements (e.g. "img.thumbnail" or "div.content > a").
- **Text Selectors:** Use the text= prefix to find by text content (e.g. page.locator("text=Sign In") finds a node with visible text "Sign In").
- **XPath:** Prefix with xpath= for XPath queries (e.g. page.locator("xpath=//div[@class='content']")).
- **Role Selectors:** Use page.get_by_role("button", name="Submit") to find elements by ARIA role and name (accessible name). This is robust for interactive elements.
- **Special Locators:** Playwright has built-in locator helpers:
  - page.get_by_text("exact text") – finds element by text (supports substring or regex).
  - page.get_by_alt_text("alt text") – finds an element (usually <img>) by its alt attribute (Locators | Playwright Python).
  - page.get_by_title("tooltip") – finds by the title attribute (often for <a title="..."> or icons).
  - page.get_by_label("Name") – finds input by associated <label> text.
  - These can simplify targeting elements like images by alt text or links by title.

- **page.query_selector vs page.locator:**
    - page.query_selector("selector") returns an **ElementHandle** for the first match (or None). It represents a specific element in the DOM at that moment.
    - page.locator("selector") returns a **Locator** which is a query that can find an element (or multiple). Locators **auto-wait** for elements to appear and can be reused for multiple actions. They evaluate to the element fresh each time, making them robust for dynamic pages (Playwright queryselector vs locator | Restackio).
    - **Best Practice:** Use locators for most scraping interactions because of their auto-wait and ease of chaining. Use query_selector_all or locator strategies to get multiple elements if needed.
- **Multiple Elements:** To get many elements:
    - handles = page.query_selector_all("div.post") gives a list of ElementHandles.
    - With locators: posts = page.locator("div.post"), then you can do count = posts.count(), and iterate for i in range(count): posts.nth(i).text_content().
    - You can also retrieve all at once: post_elements = posts.element_handles(). This gives a snapshot list of ElementHandles matching the locator at that time.
- **Chaining and Filtering:** Locators allow filtering:
- post = page.locator("div.post").filter(has_text="Hello") # post with "Hello" in it
- image = page.locator("figure").locator("img") # find img inside a figure
- first_item = page.locator("ul li").first # or .nth(0) – first <li>
- last_item = page.locator("ul li").last   # last <li>

This is useful for narrowing down to the right element (e.g., find an image inside a specific container).

- **Frames (Subframes):** If content is inside an <iframe>, use page.frame(name="iframe_name") or page.frame(url=re.compile(r"some_url")) to get a Frame object. Then use frame.locator/selector methods on that frame. Alternatively, iframe_element = page.query_selector("iframe.selector") then frame =

iframe_element.content_frame to get the frame context. All the same element-finding methods apply on Frame or Page.

Extracting Attributes and Text

Once you have an element (via Locator or ElementHandle), you can retrieve its properties:

- **Text Content:**
  - element_handle.text_content() returns the full text of the element (including hidden text, and text of child elements) as in the DOM.
  - element_handle.inner_text() returns the rendered innerText (visible text with whitespace collapsing, excluding text of hidden elements) (Page | Playwright Python). In most cases for scraping, text_content() is fine if you want all text; inner_text() if you specifically want visible text as seen by users.
  - *Shortcut:* If using locators, you can do locator.text_content() directly and it will fetch text from the first match.
  - Example:
  - title_elem = page.locator("h1.page-title")
  - title = title_elem.inner_text() # visible title text
  - description = page.locator("div.desc").text_content()
  - print(title, description)
- **Element Attributes:** Use element_handle.get_attribute("attr") to get any attribute (like src, href, alt, title, data-* attributes). This returns a string or None (Page | Playwright Python) (Page | Playwright Python). For example:
- img = page.query_selector("img.cover")
- src_url = img.get_attribute("src")
- alt_text = img.get_attribute("alt")
  - *Locator shortcut:* locator.get_attribute("attr") will wait for the element and return the attribute. For instance, page.locator("img.cover").get_attribute("src").
  - If the attribute is not present, result is None.

- **Page Content & HTML:** page.content() returns the full HTML source of the page. Similarly, element_handle.inner_html() gives the HTML markup inside an element (between its tags). Use these if you plan to parse HTML with another library (like BeautifulSoup or lxml).
- **Media Elements:** For images and videos, relevant attributes include:
  - Images: src, alt, title, possibly srcset (for responsive images – a string of URLs and descriptors). If srcset is present, you may need to parse it or choose the highest resolution URL.
  - Videos: If <video> has no direct src, check for <source> children. For example:
  - video = page.query_selector("video")
  - video_src = video.get_attribute("src")
  - if not video_src:
  -     # perhaps multiple sources
  -     source_elems = video.query_selector_all("source")
  -     sources = [s.get_attribute("src") for s in source_elems]
  - Anchor links: href attribute (and link text via .inner_text()).
- **Associated Metadata:** Often the metadata like author, caption, date, etc., are in sibling or parent elements:
  - You can navigate the DOM using locators or by evaluating JavaScript:
    - Example: After finding an image element, get its parent node's text:
    - img = page.locator("img.photo").first
    - caption = img.evaluate("img => img.parentElement.innerText")

This would yield the text content of the image's parent element (if the caption/credits are there).

    - Or use relative selectors: e.g., if captions are in a <figcaption> next to the <img>:
    - figure = page.locator("figure:has(img.photo)").first
    - caption_text = figure.locator("figcaption").inner_text()

        - author = figure.locator(".author-name").inner_text()
    - **Alt/Title Text:** Already covered via attributes. These often serve as descriptive metadata for images.
    - **Date:** If a date is present, find an element containing it (e.g., a time tag or a specific class) and get its text. Sometimes date might be in a datetime attribute of a <time> tag.
    - **Credits/Attribution:** Could be in a caption or a span with class like "credit". Use appropriate selector and get text.
- **Example – Scraping an Image with Metadata:**
  Suppose a page has posts with an image, title, author, and date:
- posts = page.locator("div.post")
- for i in range(posts.count()):
-   post = posts.nth(i)
-   img_url = post.locator("img").get_attribute("src")
-   alt = post.locator("img").get_attribute("alt")
-   title = post.locator("h2.title").inner_text()
-   author = post.locator(".author-name").inner_text()
-   date = post.locator("time").get_attribute("datetime") or post.locator("time").inner_text()
-   print(title, img_url, alt, author, date)

This pattern finds each post container and extracts the desired fields.

Scrolling and Interaction (For Infinite Scroll & Lazy Content)

Modern sites often load content dynamically as you scroll or require interaction (clicks) to reveal content:

- **Scrolling the Page:** The simplest way is executing JavaScript:
- page.evaluate("window.scrollTo(0, document.body.scrollHeight)") # scroll to bottom

or using keyboard events:

page.keyboard.press("End") # press End to scroll to bottom

However, infinite scroll may load content incrementally. You might need to scroll multiple times.

- **Infinite Scroll Loop:** One strategy: scroll in increments or to the last known element until no new content loads:
- prev_height = 0
- while True:
-     page.evaluate("window.scrollTo(0, document.body.scrollHeight)")
-     page.wait_for_timeout(2000) # wait 2s for new content to load
-     new_height = page.evaluate("() => document.body.scrollHeight")
-     if new_height == prev_height:
-         break # no more content
-     prev_height = new_height

This scrolls to the bottom, waits, and checks if page height increased. If not, it assumes loading is done.

- **Scrolling Specific Elements:** As an alternative, find a container or list of items and scroll the last item into view to trigger loading of the next batch (Web Scraping with Playwright and Python) (Web Scraping with Playwright and Python):
- items = page.locator(".item-class") # locator for content items
- previous_count = 0
- while True:
-     count = items.count()
-     if count == 0 or count == previous_count:
-         break
-     items.nth(count - 1).scroll_into_view_if_needed()
-     page.wait_for_timeout(1500) # small pause for new items
-     previous_count = count

This uses locator.scroll_into_view_if_needed() on the last item to load more (Web Scraping with Playwright and Python). Continue until the count stops increasing.

- **Lazy-loaded Images:** Some images load when visible (using loading="lazy" or similar). By scrolling, you trigger them to load. Ensure to wait a bit (wait_for_timeout) after scrolling so the images' src attributes get populated. If images still have placeholders, consider checking for a data-src attribute:
- imgs = page.query_selector_all("img[data-src]")
- for img in imgs:
-   src = img.get_attribute("src")
-   data_src = img.get_attribute("data-src")
-   if src is None or "placeholder" in src:
-     # image not loaded yet, use data-src or trigger loading
-     page.evaluate("(img) => { img.src = img.dataset.src; }", img)

The above forces any lazy image to load by setting its src. Usually, scrolling is sufficient in most cases.

- **Clicking Buttons/Links:** Use page.click("selector") or locator .click(). This will auto-wait for the element to be visible and enabled. For example, clicking a "Load more" button:
- load_more = page.locator("text=Load more")
- while load_more.is_visible():
-   load_more.click()
-   page.wait_for_timeout(1000) # wait for new content chunk
- **Filling Forms / Interactions:** If login or search is required:
    - page.fill("input[name=username]", "myuser") and page.fill("input[name=password]", "mypassword") to fill fields.
    - page.press("input[name=password]", "Enter") to submit by pressing Enter, or page.click("button:has-text('Login')").
    - These actions also auto-wait for the element to be ready.

- **Hover and Other Actions:** If needed, page.hover(selector) or locator.hover() can simulate mouseover (useful if content appears on hover). You can also use the mouse or keyboard directly via page.mouse and page.keyboard for custom interactions (e.g., page.mouse.wheel(0, 300) to scroll).

Handling Multiple Pages and Frames

Your scraping workflow may involve multiple pages or pop-ups:

- **Multiple Pages/Tabs:** If clicking a link opens a new tab or window, Playwright can catch it:
- with page.expect_popup() as popup_info:
-   page.click("a[target=_blank]") # or whatever triggers new page
- new_page = popup_info.value
- new_page.wait_for_load_state()
- print("New page URL:", new_page.url)

Here we use page.expect_popup() context manager to wait for a new Page (Page | Playwright Python). Once opened, new_page is a Page object you can interact with (just like the original). Remember to close it when done if needed (new_page.close()).

- **Multiple Pages in one script:** You can also manually create pages:
- page1 = context.new_page()
- page2 = context.new_page()

This can be used to scrape two pages concurrently (e.g., navigate page1 and page2 to different URLs and operate on both). However, if doing so, be mindful of asynchronous behavior – in sync API you'd typically do one page at a time or use threads for true parallelism.

- **Iterating through links:** For scraping detail pages, one strategy:
    1. Use one page (or context) to load a listing (e.g., a gallery of images).
    2. Extract links for details.
    3. For each link, either reuse one page to navigate to the link (with page.goto(detail_url)), scrape data, then go back or navigate to the next. *Or*, open a new page for each link (especially if you want to keep the listing page open).

```
Example sequentially on one page:
links = [a.get_attribute("href") for a in page.query_selector_all("a.project-link")]
for link in links:
   page.goto(link)
   # scrape detail (image, author, etc.)
   data = {...}
   results.append(data)
```

This navigates the same page to each detail. Alternatively, use a separate context/new_page inside the loop to isolate each navigation if needed.

- **Frames and Embeds:** If the page includes iframes (e.g., embedded videos or ads), you might need to access them:
    - List frames: for frame in page.frames: print(frame.url, frame.name).
    - To get a specific frame by name or URL: frame = page.frame(name="iframeName") or page.frame(url=re.compile("iframeURLpart")).
    - Then use frame.locator(...) or frame.query_selector(...) to scrape inside that frame. Remember, if the iframe is from another domain, you can only access it if it's not cross-origin or if CORS allows. Otherwise, you may need to navigate that frame by itself via its frame.goto if allowed.

Cookies, Authentication, and Session Handling

When scraping sites that require login or maintain sessions, Playwright provides tools to manage cookies and local storage:

- **Setting Cookies:** Use context.add_cookies([...]) to add cookies to a context (BrowserContext | Playwright Python) (BrowserContext | Playwright Python). Each cookie is a dict with fields like name, value, domain, path, etc. You can grab cookies from a logged-in session or from your browser and inject them. Example:
- context = browser.new_context()
- context.add_cookies([{
-    "name": "sessionid", "value": "ABC123", "domain": ".example.com", "path": "/"

- }])
- page = context.new_page()
- page.goto("https://example.com/profile")

Ensure the domain/path match the site.

- **Retrieving Cookies:** context.cookies() returns a list of cookies (optionally filtered by URL). This is useful after logging in via UI:
- # after login:
- cookies = context.cookies()
- for cookie in cookies:
-     print(cookie["name"], cookie["value"])
- **Saving/Loading Session (Storage State):** Playwright can save the combined state (cookies + localStorage) of a context:
- context.storage_state(path="state.json")

This writes a JSON with cookies and local storage. Later, you can launch a context with that state:

```
context = browser.new_context(storage_state="state.json")

page = context.new_page()

page.goto("https://example.com")  # already logged in if state had login info
```

This is very handy for not redoing login each time.

- **Authentication Flows:** If you need to log in:
  - Automate the login form fill as described in **Scrolling and Interaction**.
  - Or use persistent context: e.g., launch persistent context pointed at a profile directory where you have logged in before (e.g., using Chrome user data). This is heavier, but keeps you logged in across runs.
  - After login, save storage_state as above for reuse.
- **Session Cookies in Headless:** Cookies set in headful (when you manually log in) can be reused in headless by using storage_state or explicitly setting them, since

headless browser is a separate instance. Always ensure you're using the same domain and maybe user agent if needed to avoid the site treating it as a new device.

- **Handling Cookie Banners/Popups:** Many modern sites show GDPR cookie consent dialogs. These can block interactions or content. You might need to detect and close them:
- consent = page.query_selector("button#accept-cookies")
- if consent:
-     consent.click()
-     page.wait_for_timeout(500) # wait a bit after clicking

You could also use page.locator(...).click() with a try/except if not present. Playwright even has page.add_init_script() to preemptively remove or stub certain overlays, but that's advanced usage.

- **Headers and User Agent:** For scraping, you might want to set a custom user agent or other headers (especially if the default Playwright UA triggers bot detection). Use context = browser.new_context(user_agent="Mozilla/5.0 ...") or context.set_extra_http_headers({"Accept-Language": "en"}) etc. These apply to all requests in that context.

Waiting and Timing Strategies

Getting the timing right is crucial when scraping dynamic sites:

- **Auto-Waiting:** Playwright automatically waits for elements during actions like click or fill (it waits for element to be present and enabled). However, when just extracting data, you often need to wait explicitly for content to load.
- **page.wait_for_selector(selector, **kwargs):** Pause until an element matching selector appears (or satisfies a condition). By default it waits for the element to be **visible** (Page | Playwright Python). You can specify state="attached" to wait just for presence in DOM, or state="visible" (default), or even state="hidden" to wait for disappearance.
  - Example: page.wait_for_selector("img.photo", timeout=10000) – wait up to 10s for an image with class "photo" to appear.
  - This returns an ElementHandle which you can use, or None if waiting for hidden and it goes hidden.

  - **Use case:** After navigating or scrolling, use wait_for_selector to ensure new content or a specific item loaded. *E.g.*:
  - page.goto("https://behance.net/...")
  - page.wait_for_selector("div.project-item") # wait for project items to load

Then proceed to scrape.

- **page.wait_for_load_state(state):** Wait for the page's load state to reach a certain point. Commonly used after actions that trigger navigation. For instance, after page.click() that triggers navigation:
- page.click("a.next-page")
- page.wait_for_load_state("domcontentloaded")

If you omit state, it waits for full load. This is generally not needed for page.goto (which already waits), but can be useful for waiting on newly opened pages or after expect_popup as shown in the new page example above (Page | Playwright Python) (Page | Playwright Python).

- **page.wait_for_timeout(ms):** Simply sleeps for the given milliseconds. Use this for crude waits (e.g., waiting a couple seconds after a scroll for content to load). Example: page.wait_for_timeout(2000) waits 2 seconds. This does not check any condition, it just delays.
- **page.wait_for_function(expression, **kwargs):** Repeatedly evaluates a JavaScript expression until it returns a truthy value. This is powerful for custom conditions. For example, wait until the number of loaded items increases:
- prev_count = page.locator(".item").count()
- page.evaluate("window.scrollTo(0, document.body.scrollHeight)")
- page.wait_for_function("(prev) => document.querySelectorAll('.item').length > prev", prev_count)

The above passes prev_count into the page context and waits until the length of .item elements is greater. You can also use this to wait for a specific JavaScript variable or any condition in the page.

- **Default Timeouts:** As mentioned, the default timeout for most waits and actions is 30s. If the site is very slow, consider increasing it or using try/except to handle TimeoutError.

- **Networking Considerations:** If waiting for network requests (like after clicking a load-more that fetches via XHR/Fetch), you might not have a DOM element to wait on. In those cases, consider:
    - page.wait_for_response(lambda response: response.url == "https://api/endpoint" and response.finished()) to wait for a specific network call.
    - Or use networkidle state in wait_for_load_state, though as noted, it can be unreliable if the page continuously polls.
- **Headless vs Headful timing:** Headless browser might load faster (no rendering overhead), but some sites might behave differently. If you notice content not loading in headless that does in headful, ensure that you wait for the content. Sometimes adding a slight delay or using scroll_into_view_if_needed() on elements can help trigger lazy-load in headless mode (Web Scraping with Playwright and Python). In rare cases, using headful (headless=False) might be necessary for complex animations or if the site detects headless (you can try to spoof detection, but that's beyond this guide).
- **Error Handling:** Wrap your scraping steps in try/except for TimeoutError to handle cases where an element never appears. For example:
- from playwright.sync_api import TimeoutError
- try:
-   page.wait_for_selector(".profile-picture", timeout=5000)
- except TimeoutError:
-   print("Profile picture not found, skipping...")

Putting it all together: Using these techniques, you can reliably automate a browser to scroll through Behance or Reddit, collect image and video URLs, and gather metadata like titles, authors, and captions. By controlling the browser context, you maintain session or login state; by using proper selectors and waits, you ensure you get all dynamic content loaded; and by extracting attributes and text carefully, you gather the rich data (alt text, titles, etc.) associated with each media element.