Hyperbrowser

Welcome to Hyperbrowser

Welcome to Hyperbrowser, the Internet for AI. Hyperbrowser is the next-generation platform for effortless, scalable browser automation. Instead of wrestling with local infrastructure and performance bottlenecks, you can focus on building better solutions. Whether you’re collecting data, testing applications, or enabling AI-driven interactions, Hyperbrowser lets you launch and manage browser sessions with ease—no complicated setup required. Hyperbrowser also provides you easy to use solutions for all your webscraping needs, whether you need to scrape a single page or crawl an entire site.

Why Hyperbrowser?

- **Instant Scalability -** Spin up hundreds of browser sessions in seconds without infrastructure headaches
- **Simple Integration** - Works seamlessly with popular tools like Puppeteer and Playwright
- **Powerful APIs** - Easy to use APIs for managing your sessions, scraping/crawling any site, and much more
- **Production Ready** - Enterprise-grade reliability and security built-in
- **Bypass Anti-Bot Measures** - Built-in stealth mode, ad blocking, automatic CAPTCHA solving, and rotating proxies

Quick Example

Start automating in just a few lines of code

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { connect } from "puppeteer-core";

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const session = await client.sessions.create();

// Use the session to automate browser actions
const browser = await connect({
  browserWSEndpoint: session.wsEndpoint,
  defaultViewport: null,
});

// Use the browser to automate browser actions
const page = await browser.newPage();
await page.goto("https://example.com");

await browser.close();

// Once done, you can stop the session
await client.sessions.stop(session.id);
```

Jump right in

📖

Scraping

Scrape a site and get its contents in markdown

🧑‍💻

Puppeteer

Connect to a browser session with Puppeteer

🕷️

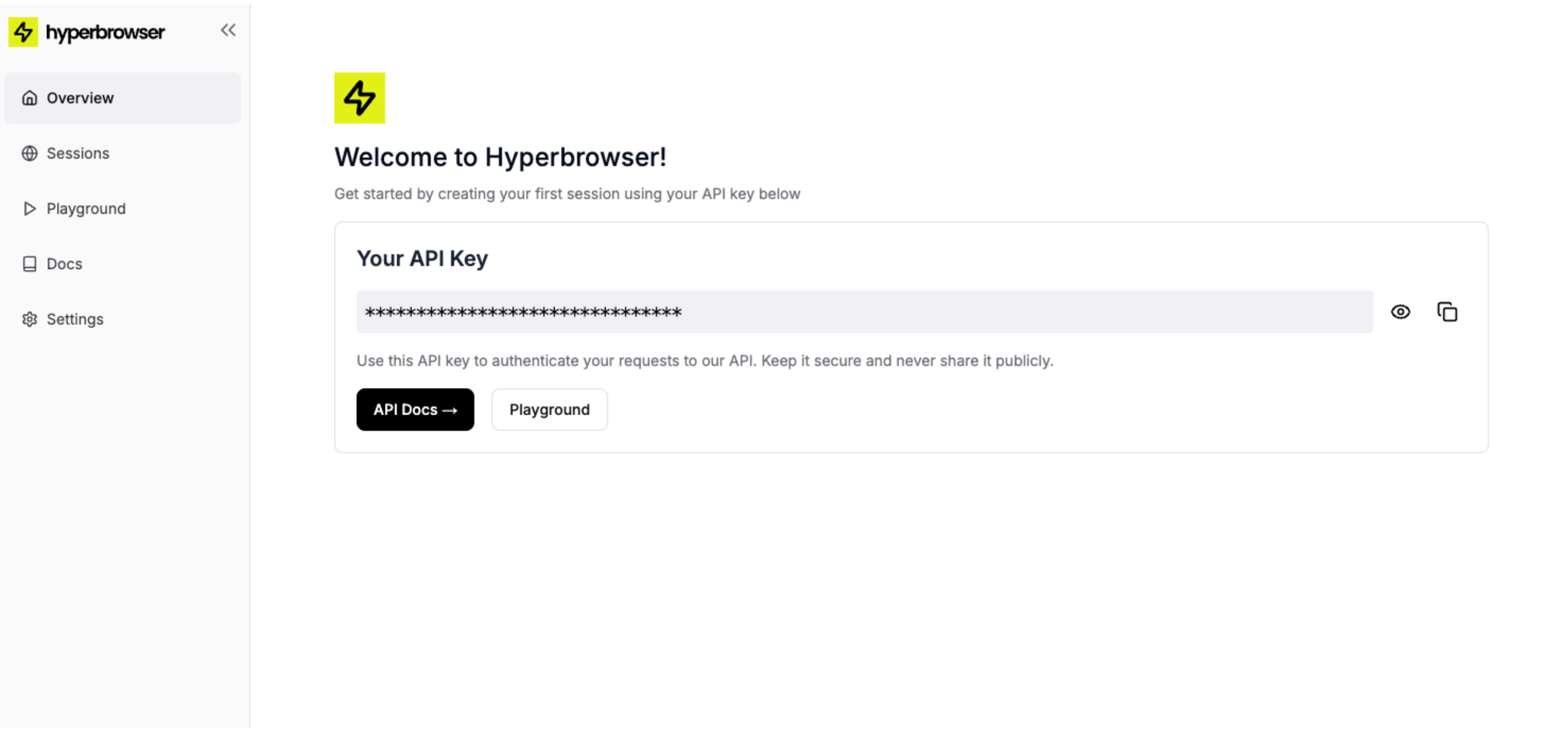
Crawling

Crawl an entire site and all its linked pages

Get Started

Quickstart

Get setup with Hyperbrowser in minutes.



Welcome to Hyperbrowser!

Get started by creating your first session using your API key below

Your API Key

****

Use this API key to authenticate your requests to our API. Keep it secure and never share it publicly.

To get started with Hyperbrowser, head on over to the dashboard and create an account. All you need now to start scraping, crawling, creating browser sessions, etc. is to use your API Key and one of our SDKs available in Node and Python or just any normal API request. Next, let's see the different types of things we can do within just a couple minutes.

Check out our in-depth SDK references for Node and Python.

Scraping

Scrape any site and get it's data.

1 Install Hyperbrowser

Node

```
npm install @hyperbrowser/sdk
```

or

```
yarn add @hyperbrowser/sdk
```

Python

```
pip install hyperbrowser
```

2 Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` .

3 Scrape a Site

Next, you can scrape any site by simply setting up the Hyperbrowser client and providing the site's url.

Node

```javascript
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const scrapeResult = await client.scrape.startAndWait({
    url: "https://example.com",
  });
  console.log("Scrape result:", scrapeResult);
};

main();
```

Python

```python
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser as Hyperbrowser
from hyperbrowser.models.scrape import StartScrapeJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Start scraping and wait for completion
    scrape_result = await client.scrape.start_and_wait(
        StartScrapeJobParams(url="https://example.com")
    )
    print("Scrape result:", scrape_result)


if __name__ == "__main__":
    import asyncio

    asyncio.run(main())
```

4 View Session in Dashboard

You can view all your sessions in the dashboard and see their recordings or other key metrics like logs.

Crawling

Crawl a site and all it's links.

1 Install Hyperbrowser

Node

```
npm install @hyperbrowser/sdk
```

or

```
yarn add @hyperbrowser/sdk
```

Python

```
pip install hyperbrowser
```

2 Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` .

3 Crawl a Site

Next, you can crawl any site by simply setting up the Hyperbrowser client and providing the site's url.

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const crawlResult = await client.crawl.startAndWait({
    url: "https://hyperbrowser.ai",
  });
  console.log("Crawl result:", crawlResult);
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser as Hyperbrowser
from hyperbrowser.models.crawl import StartCrawlJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Start crawling and wait for completion
    crawl_result = await client.crawl.start_and_wait(
        StartCrawlJobParams(url="https://hyperbrowser.ai")
    )
    print("Crawl result:")
    print(crawl_result.model_dump_json(indent=2))


if __name__ == "__main__":
    import asyncio

    asyncio.run(main())
```

4

View Session in Dashboard

You can view all your sessions in the dashboard and see their recordings or other key metrics like logs.

Puppeteer

Setup a browser session with Puppeteer.

1 Install Puppeteer and Hyperbrowser

Node

```
npm install puppeteer-core @hyperbrowser/sdk
```

or

```
yarn add puppeteer-core @hyperbrowser/sdk
```

Python

```
pip install pyppeteer hyperbrowser
```

2 Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` .

3 Setup Browser Session

Next, you can easily startup a browser session using puppeteer and your Hyperbrowser API Key.

Node

```
import { connect } from "puppeteer-core";
import { config } from "dotenv";

config();

const main = async () => {
  const browser = await connect({
    browserWSEndpoint: `wss://connect.hyperbrowser.ai?apiKey=${process.env.HYPERBI
  });

  const [page] = await browser.pages();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  await page.goto("https://example.com");
  console.log("Page 2:", await page.title());
  await page.evaluate(() => {
    console.log("Page 2:", document.title);
  });

  await page.goto("https://apple.com");
  console.log("Page 3:", await page.title());
  await page.evaluate(() => {
    console.log("Page 3:", document.title);
  });

  await page.goto("https://google.com");
  console.log("Page 4:", await page.title());
  await page.evaluate(() => {
    console.log("Page 4:", document.title);
  });

  // Clean up
  await browser.close();
};

main();
```

Python

```
import asyncio
from pyppeteer import connect
import os
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

async def main():
    # Connect to the browser using the API key from environment variables
    browser = await connect(
        browserWSEndpoint=f"wss://connect.hyperbrowser.ai?apiKey={os.getenv('HYPEI
    )

    pages = await browser.pages()
    page = pages[0]

    # Navigate to a website
    print("Navigating to Hacker News...")
    await page.goto("https://news.ycombinator.com/")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 1:', document.title); }")

    await page.goto("https://example.com")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 2:', document.title); }")

    await page.goto("https://apple.com")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 3:', document.title); }")

    await page.goto("https://google.com")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 4:', document.title); }")

    # Clean up
    await browser.close()

# Run the async main function
if __name__ == "__main__":
    asyncio.run(main())
```

4 View Session in Dashboard

You can view all your sessions in the dashboard and see their recordings or other key metrics like logs.

Playwright

Setup a browser session with Playwright.

1 Install Playwright and Hyperbrowser

Node

```
npm install playwright-core @hyperbrowser/sdk
```

or

```
yarn add playwright-core @hyperbrowser/sdk
```

Python

```
pip install playwright hyperbrowser
```

2 Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` .

3 Setup Browser Session

Next, you can easily startup a browser session using playwright and your Hyperbrowser API Key.

Node

```
import { chromium } from "playwright-core";
import { config } from "dotenv";

config();

const main = async () => {
  // Connect to browser using Playwright
  const browser = await chromium.connectOverCDP(
    `wss://connect.hyperbrowser.ai?apiKey=${process.env.HYPERBROWSER_API_KEY}`
  );

  // Create a new context and page
  const defaultContext = browser.contexts()[0];
  const page = defaultContext.pages()[0];

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  await page.goto("https://example.com");
  console.log("Page 2:", await page.title());
  await page.evaluate(() => {
    console.log("Page 2:", document.title);
  });

  await page.goto("https://apple.com");
  console.log("Page 3:", await page.title());
  await page.evaluate(() => {
    console.log("Page 3:", document.title);
  });

  await page.goto("https://google.com");
  console.log("Page 4:", await page.title());
  await page.evaluate(() => {
    console.log("Page 4:", document.title);
  });

  // Clean up
  await defaultContext.close();
  await browser.close();
};

main();
```

Python

```
import os
from playwright.sync_api import sync_playwright
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

def main():
    with sync_playwright() as p:
        # Connect to browser using Playwright
        browser = p.chromium.connect_over_cdp(
            f"wss://connect.hyperbrowser.ai?apiKey={os.getenv('HYPERBROWSER_API_K
        )

        # Get the default context and page
        default_context = browser.contexts[0]
        page = default_context.pages[0]

        # Navigate to various websites
        print("Navigating to Hacker News...")
        page.goto("https://news.ycombinator.com/")
        page_title = page.title()
        print("Page 1:", page_title)
        page.evaluate("() => { console.log('Page 1:', document.title); }")

        page.goto("https://example.com")
        print("Page 2:", page.title())
        page.evaluate("() => { console.log('Page 2:', document.title); }")

        page.goto("https://apple.com")
        print("Page 3:", page.title())
        page.evaluate("() => { console.log('Page 3:', document.title); }")

        page.goto("https://google.com")
        print("Page 4:", page.title())
        page.evaluate("() => { console.log('Page 4:', document.title); }")

        # Clean up
        default_context.close()
        browser.close()

if __name__ == "__main__":
    main()
```

4 View Session in Dashboard

You can view all your sessions in the dashboard and see their recordings or other key metrics like logs.

Selenium

Setup a browser session with Selenium.

1 Install Selenium and Hyperbrowser

Node

```
npm install selenium-webdriver @hyperbrowser/sdk
```

or

```
yarn add selenium-webdriver @hyperbrowser/sdk
```

Python

```
pip install selenium hyperbrowser
```

2 Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY`.

3 Setup Browser Session

Next, you can easily startup a browser session using selenium and your Hyperbrowser API Key.

Node

```
import dotenv from 'dotenv';
import https from 'https';
import { Builder, WebDriver } from 'selenium-webdriver';
import { Options } from 'selenium-webdriver/chrome';
import { Hyperbrowser } from '@hyperbrowser/sdk';

// Load environment variables from .env file
dotenv.config();

const client = new Hyperbrowser({ apiKey: process.env.HYPERBROWSER_API_KEY as str

async function main() {
    const session = await client.sessions.create();

    const customHttpsAgent = new https.Agent({});
    (customHttpsAgent as any).addRequest = (req: any, options: any) => {
        req.setHeader('x-hyperbrowser-token', session.token);
        (https.Agent.prototype as any).addRequest.call(customHttpsAgent, req, opt
    };

    const driver: WebDriver = await new Builder()
        .forBrowser('chrome')
        .usingHttpAgent(customHttpsAgent)
        .usingServer('https://connect.hyperbrowser.ai/webdriver')
        .setChromeOptions(new Options())
        .build();

    try {
        // Navigate to a URL
        await driver.get("https://www.google.com");
        console.log("Navigated to Google");

        // Search
        const searchBox = await driver.findElement({ name: "q" });
        await searchBox.sendKeys("Selenium WebDriver");
        await searchBox.submit();
        console.log("Performed search");

        // Screenshot
        await driver.takeScreenshot().then(data => {
            require('fs').writeFileSync('search_results.png', data, 'base64');
        });
        console.log("Screenshot saved");
    } finally {
        await driver.quit();
    }
}

if (require.main === module) {
    main().catch(console.error);
}
```

Python

```python
import os
from dotenv import load_dotenv

from selenium import webdriver
from selenium.webdriver.remote.remote_connection import RemoteConnection
from selenium.webdriver.chrome.options import Options
from hyperbrowser import Hyperbrowser

# Load environment variables from .env file
load_dotenv()

client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))

class CustomRC(RemoteConnection):
    _signing_key = None

    def __init__(self, server: str, token: str):
        super().__init__(server)
        self._token = token

    def get_remote_connection_headers(self, parsed_url, keep_alive=False):
        headers = super().get_remote_connection_headers(parsed_url, keep_alive)
        headers.update({'x-hyperbrowser-token': self._token})
        return headers

def main():
    session = client.sessions.create()
    custom_conn = CustomRC("https://connect.hyperbrowser.ai/webdriver", session.t
    driver = webdriver.Remote(custom_conn, options=Options())

    # Navigate to a URL
    driver.get("https://www.google.com")
    print("Navigated to Google")

    # Search
    search_box = driver.find_element("name", "q")
    search_box.send_keys("Selenium WebDriver")
    search_box.submit()
    print("Performed search")

    # Screenshot
    driver.save_screenshot("search_results.png")
    print("Screenshot saved")

if __name__ == "__main__":
    main()
```

4

View Session in Dashboard

You can view all your sessions in the dashboard and see their recordings or other key metrics like logs.

Sessions

Overview

Learn about Hyperbrowser Sessions

In Hyperbrowser, a Session is simply a dedicated, cloud-based browser instance that's yours to direct. It's like opening a fresh, private browser window—only this one lives in the cloud and is fully controllable through code. Each Session keeps its own cookies, storage, and browsing context, so you can run your tasks without mixing them up.

With our Sessions API, you can spin up these browser environments whenever you need them, configure them with optional parameters, and close them down when you're done. Higher level endpoints like scrape and crawl also utilize sessions internally to handle their tasks.

Connect with your favorite browser automation libraries

Puppeteer

Connect with Puppeteer to automate browser actions via a websocket connection

Playwright

Connect with Playwright to automate browser actions via a websocket connection

Check out our API Reference to see the Sessions API in more detail

Usage

Starting a session is pretty simple, you can either startup a session directly by connecting to our websocket endpoint with your preferred automation tool like Playwright or Puppeteer, or you can create a session via the Sessions API.

Simple Connect

With this approach, you can startup a session with all default configurations set by just changing one line.

Node

Puppeteer

```
const browser = await puppeteer.connect({
    browserWSEndpoint: `wss://connect.hyperbrowser.ai?apiKey=${process.env.HYPERBROWS
});
```

Playwright

```
const browser = await chromium.connectOverCDP(
    `wss://connect.hyperbrowser.ai?apiKey=${process.env.HYPERBROWSER_API_KEY}`
);
```

Python

Puppeteer

```
browser = await pyppeteer.connect(
    browserWSEndpoint=f"wss://connect.hyperbrowser.ai?apiKey={os.getenv('HYPERBROWSER
)
```

Playwright

```
with sync_playwright() as p:
        browser = p.chromium.connect_over_cdp(
            f"wss://connect.hyperbrowser.ai?apiKey={os.getenv('HYPERBROWSER_API_KEY')}
        )
```

Connect with Sessions API

With this approach, you have more control over your sessions with additional features or updated configurations.

Node

Puppeteer

```
import { connect } from "puppeteer-core";
import { Hyperbrowser } from "@hyperbrowser/sdk"
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const session = await client.sessions.create({
    solveCaptchas: true,
    useStealth: true,
    useProxy: true,
  });
  const browser = await connect({
    browserWSEndpoint: session.wsEndpoint,
  });

  const [page] = await browser.pages();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  // Clean up
  await page.close();
  await browser.close();
  await client.sessions.stop(session.id);
};

main();
```

Playwright

```
import { chromium } from "playwright-core";
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const session = await client.sessions.create({
    solveCaptchas: true,
    useStealth: true,
    useProxy: true,
  });

  const browser = await chromium.connectOverCDP(session.wsEndpoint);

  const context = await browser.newContext();
  const page = await context.newPage();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  // Clean up
  await page.close();
  await context.close();
  await browser.close();
  await client.sessions.stop(session.id);
};

main();
```

Python

Puppeteer

```python
import asyncio
from pyppeteer import connect
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser
from hyperbrowser.models.session import CreateSessionParams

# Load environment variables from .env file
load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Create a session and connect to it using Pyppeteer
    session = await client.sessions.create(
        params=CreateSessionParams(
            solve_captchas=True, use_stealth=True, use_proxy=True
        )
    )
    browser = await connect(browserWSEndpoint=session.ws_endpoint)

    pages = await browser.pages()
    page = pages[0]

    # Navigate to a website
    print("Navigating to Hacker News...")
    await page.goto("https://news.ycombinator.com/")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 1:', document.title); }")

    # Clean up
    await browser.close()
    await client.sessions.stop(session.id)

# Run the async main function
if __name__ == "__main__":
    asyncio.run(main())
```

Playwright

```python
import asyncio
from playwright.async_api import async_playwright
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser
# You can also use the sync `Hyperbrowser` to work with sync_playwright
from hyperbrowser.models.session import CreateSessionParams

# Load environment variables from .env file
load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Create a session and connect to it using Playwright
    session = await client.sessions.create(
        params=CreateSessionParams(
            solve_captchas=True, use_stealth=True, use_proxy=True
        )
    )

    # Initialize Playwright and connect to the browser
    async with async_playwright() as p:
        browser = await p.chromium.connect_over_cdp(session.ws_endpoint)
        default_context = browser.contexts[0]
        page = default_context.pages[0]

        # Navigate to a website
        print("Navigating to Hacker News...")
        await page.goto("https://news.ycombinator.com/")
        page_title = await page.title()
        print("Page title:", page_title)
        await page.evaluate("() => { console.log('Page 1:', document.title); }")

        # Clean up
        await browser.close()
        await client.sessions.stop(session.id)


# Run the async main function
if __name__ == "__main__":
    asyncio.run(main())
```

Hyperbrowser's CAPTCHA solving and Proxy features require being on a `PAID` plan.

Advanced Privacy & Anti-Detection

Hyperbrowser Anti-Bot Detection

Sometimes you need your automated browser sessions to fly under the radar. With Hyperbrowser's privacy and anti-detection features, you can tweak things like browser fingerprints, pick the right proxies, and decide exactly how requests get routed. This helps your automated visits look and feel more like regular browsing—something that's especially handy if you're dealing with strict anti-bot measures or running sensitive operations.

Whether you need to appear as if you're browsing from a specific region, or you want to vary details like your device type and OS, it's all possible. You can set things up so your workflows feel less like scripted tasks and more like genuine user behavior. Add in built-in captcha-solving capabilities, and you've got a setup that keeps you moving forward, even if the sites you're visiting throw a few hurdles your way.

Stealth Mode

Stealth mode helps you avoid detection by anti-bot systems. It randomizes browser fingerprints and can be configured when creating a new session via the API. Options include:

- **Devices** - Specify mobile or desktop device profiles
- **Locales** - Set browser locale (e.g. en-US, fr-FR)
- **Operating Systems** - Simulate different OSes like Android, iOS, Windows, macOS, Linux
- **Screen Size** - Specify viewport dimensions to emulate different devices
- **User Agents** - Rotate user agent strings

To enable stealth mode and other stealth configurations, you can set the desired options in the session creation params when creating a session.

Node

```
import { connect } from "puppeteer-core";
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const session = await client.sessions.create({
    useStealth: true,
    operatingSystems: ["macos"],
    device: ["desktop"],
    locales: ["en"],
    screen: {
      width: 1920,
      height: 1080,
    },
  });
  const browser = await connect({
    browserWSEndpoint: session.wsEndpoint,
  });

  const [page] = await browser.pages();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  // Clean up
  await page.close();
  await browser.close();
  await client.sessions.stop(session.id);
};

main();
```

Python

```python
import asyncio
from pyppeteer import connect
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser
from hyperbrowser.models.session import CreateSessionParams, ScreenConfig

# Load environment variables from .env file
load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Create a session and connect to it using Pyppeteer
    session = await client.sessions.create(
        params=CreateSessionParams(
            use_stealth=True,
            operating_systems=["macos"],
            device=["desktop"],
            locales=["en"],
            screen=ScreenConfig(width=1920, height=1080),
        )
    )
    browser = connect(browserWSEndpoint=session.ws_endpoint)

    pages = await browser.pages()
    page = pages[0]

    # Navigate to a website
    print("Navigating to Hacker News...")
    await page.goto("https://news.ycombinator.com/")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 1:', document.title); }")

    # Clean up
    await page.close()
    await browser.close()
    await client.sessions.stop(session.id)

# Run the async main function
if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

To see all the available options, check out the Create Session API Reference

Proxies

Route browser traffic through proxies to change IP addresses. You can:

- Use Hyperbrowser's proxy pool
- Bring your own proxies

To enable these proxy configurations, you can set them in the session creation params.

Node

```
import { connect } from "puppeteer-core";
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const session = await client.sessions.create({
    useProxy: true,
    proxyCountry: "US",
    // use own proxy
    proxyServer: "...",
    proxyServerUsername: "...",
    proxyServerPassword: "...",
  });
  const browser = await connect({
    browserWSEndpoint: session.wsEndpoint,
  });

  const [page] = await browser.pages();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  // Clean up
  await page.close();
  await browser.close();
  await client.sessions.stop(session.id);
};

main();
```

Python

```python
import asyncio
from pyppeteer import connect
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser
from hyperbrowser.models.session import CreateSessionParams

# Load environment variables from .env file
load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Create a session and connect to it using Pyppeteer
    session = await client.sessions.create(
        params=CreateSessionParams(
            use_proxy=True,
            proxy_country="US",
            # use own proxy
            proxy_server="...",
            proxy_server_username="...",
            proxy_server_password="...",
        )
    )
    browser = connect(browserWSEndpoint=session.ws_endpoint)

    pages = await browser.pages()
    page = pages[0]

    # Navigate to a website
    print("Navigating to Hacker News...")
    await page.goto("https://news.ycombinator.com/")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 1:', document.title); }")

    # Clean up
    await page.close()
    await browser.close()
    await client.sessions.stop(session.id)

# Run the async main function
if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

Using proxies can introduce latency to any page navigations, so make sure to properly await these navigations with timeouts.

Proxy usage is only available on the `PAID` plans.

CAPTCHA Solving

Hyperbrowser automatically solves CAPTCHAs when the `solveCaptchas` parameter is set to true for session creation.

Node

```
import { connect } from "puppeteer-core";
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const session = await client.sessions.create({
    solveCaptchas: true,
  });
  const browser = await connect({
    browserWSEndpoint: session.wsEndpoint,
  });

  const [page] = await browser.pages();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  // Clean up
  await page.close();
  await browser.close();
  await client.sessions.stop(session.id);
};

main();
```

Python

```python
import asyncio
from pyppeteer import connect
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser
from hyperbrowser.models.session import CreateSessionParams

# Load environment variables from .env file
load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Create a session and connect to it using Pyppeteer
    session = await client.sessions.create(
        params=CreateSessionParams(
           solve_captchas=True,
        )
    )
    browser = connect(browserWSEndpoint=session.ws_endpoint)

    pages = await browser.pages()
    page = pages[0]

    # Navigate to a website
    print("Navigating to Hacker News...")
    await page.goto("https://news.ycombinator.com/")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 1:', document.title); }")

    # Clean up
    await page.close()
    await browser.close()
    await client.sessions.stop(session.id)

# Run the async main function
if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

Captcha solving can take a bit of time, so make sure to implement proper waiting strategies when navigating to pages that might contain CAPTCHAs:

```
await page.goto("https://news.ycombinator.com/",
    { waitUntil: "networkidle0" }
);

// OR
const sleep = (ms) => new Promise((res) => setTimeout(res, ms));

await page.waitForNavigation({ waitUntil: "networkidle0"});
await sleep(15000);
```

CAPTCHA solving is only available on `PAID` plans.

Ad Blocking

Hyperbrowser's browser instances can automatically block ads and trackers. This improves page load times and further reduces detection risk. In addition to ads, Hyperbrowser allows you to block trackers and other annoyances including cookie notices.

To enable ad blocking, set the proper configurations in the session create parameters.

Node

```
import { connect } from "puppeteer-core";
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const session = await client.sessions.create({
    adblock: true,
    trackers: true,
    annoyances: true,
    // You must have trackers set to true to enable blocking annoyances and
    // adblock set to true to enable blocking trackers.
  });
  const browser = await connect({
    browserWSEndpoint: session.wsEndpoint,
  });

  const [page] = await browser.pages();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page 1:", pageTitle);
  await page.evaluate(() => {
    console.log("Page 1:", document.title);
  });

  // Clean up
  await page.close();
  await browser.close();
  await client.sessions.stop(session.id);
};

main();
```

Python

```python
import asyncio
from pyppeteer import connect
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser
from hyperbrowser.models.session import CreateSessionParams

# Load environment variables from .env file
load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    # Create a session and connect to it using Pyppeteer
    session = await client.sessions.create(
        params=CreateSessionParams(
           adblock=True,
           trackers=True,
           annoyances=True,
           # You must have trackers set to true to enable blocking annoyances and
           # adblock set to true to enable blocking trackers.
        )
    )
    browser = connect(browserWSEndpoint=session.ws_endpoint)

    pages = await browser.pages()
    page = pages[0]

    # Navigate to a website
    print("Navigating to Hacker News...")
    await page.goto("https://news.ycombinator.com/")
    page_title = await page.title()
    print("Page title:", page_title)
    await page.evaluate("() => { console.log('Page 1:', document.title); }")

    # Clean up
    await page.close()
    await browser.close()
    await client.sessions.stop(session.id)

# Run the async main function
if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

Profiles

Hyperbrowser Browser Profiles

Hyperbrowser profiles allow you to reuse browser state like cookies, local storage, and network cache across multiple sessions. This can improve performance and enable seamless workflows requiring persistent data.

How Profiles Work

A profile is essentially a saved snapshot of a browser's user data directory. By default, each Hyperbrowser session uses a fresh user data directory to ensure isolation.

When you create a profile and then use and persist it in a new session, Hyperbrowser saves that session's user data directory. You can then attach the profile to future sessions to pick up where you left off.

Common use cases for profiles include:

- Reusing authenticated sessions
- Improving page load times with a primed cache
- Preserving application state across sessions

Using Profiles

1. Create a Profile

First, create a new profile using the Profiles API. Note the returned `id` - you'll need this to attach the profile to sessions.

Node

```
const profile = await client.profiles.create();
console.log("profile", profile.id);
```

Python

```
profile = client.profiles.create()
print("profile", profile.id)
```

cURL

```
curl -X POST 'https://app.hyperbrowser.ai/api/profile' \
-H 'Content-Type: application/json' \
-H 'x-api-key: YOUR_API_KEY' \
-d '{}'
```

2. Attach Profile to Session

When creating a new session with the Sessions API, include the `id` in the `profile` field:

Node

```
const session = await client.sessions.create({
  profile: {
    id: "<PROFILE_ID>",
    persistChanges: true, // set to true for the first session of a new profile
  },
});
```

Python

```
session = client.sessions.create(
    params=CreateSessionParams(
        profile=CreateSessionProfile(
            id="<PROFILE_ID>",
            # set to true for the first session of a new profile
            persist_changes=True,
        ),
    )
)
```

cURL

```
curl -X POST 'https://app.hyperbrowser.ai/api/session' \
-H 'Content-Type: application/json' \
-H 'x-api-key: YOUR_API_KEY' \
-d '{
    "profile": {
      "id": "<PROFILE_ID>"
    }
}'
```

Set `"persistChanges": true` in the `profile` object if you want the session to update the profile with any changes to cookies, cache, etc. You will need to do this for the first time you use a new profile in a session so it can be used in subsequent sessions.

3. Reuse Profile

Attach the same `id` to additional sessions to reuse browser state. Created profiles are reusable until you delete them.

Deleting Profiles

Delete unused profiles to free up resources. Once deleted, a profile is no longer attachable to sessions.

To delete a profile, send a DELETE request to the Profiles API with the target `id`:

Node

```
const response = await client.profiles.delete("<PROFILE_ID>");
console.log("response", response);
```

Python

```
response = client.profiles.delete("<PROFILE_ID>")
print("response", response)
```

cURL

```
curl -X DELETE 'https://app.hyperbrowser.ai/api/profile/<PROFILE_ID>' \
-H 'x-api-key: YOUR_API_KEY'
```

Deletion is irreversible, so be sure you no longer need a profile before deleting.

Recordings

Hyperbrowser Session Recordings

Hyperbrowser allows you to record and replay your browser sessions. It uses rrweb, an open-source web session replay library. Session recordings let you:

- Visually debug test failures and errors
- Analyze user behavior and interactions
- Share reproducible bug reports
- Save and archive session data

Enabling Session Recording

To record a session, set the `enableWebRecording` option to `true` when creating a new Hyperbrowser session:

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const session = await client.sessions.create({
    enableWebRecording: true,
  });
};

main();
```

Python

```python
import asyncio
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser
from hyperbrowser.models.session import CreateSessionParams

load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    session = await client.sessions.create(
        params=CreateSessionParams(
            enable_web_recording=True,
        )
    )


if __name__ == "__main__":
    import asyncio

    asyncio.run(main())
```

This will record all browser interactions, DOM changes, and network requests for the duration of the session.

Retrieving Recordings

Please contact us at info@hyperbrowser.ai to get access to this feature.

Recorded sessions are automatically saved when the Hyperbrowser session ends. To retrieve a session recording:

1. Note the `id` of the session you want to replay
2. Use the Session Recordings API to download the recording, or if you are using the SDKs, you can just call the `getRecording` function:

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const recordingData = await client.sessions.getRecording(
    "91e96d43-0dd2-4882-8d3a-613b12583ba2"
  );
};

main();
```

Python

```python
import asyncio
import os
from dotenv import load_dotenv
from hyperbrowser import AsyncHyperbrowser

load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


async def main():
    recording_data = await client.sessions.get_recording(
        "91e96d43-0dd2-4882-8d3a-613b12583ba2"
    )


if __name__ == "__main__":
    import asyncio

    asyncio.run(main())
```

cURL

```
curl -X GET 'https://app.hyperbrowser.ai/api/session/{sessionId}/recording' \
-H 'x-api-key: YOUR_API_KEY
```

The recording data will be returned in rrweb's JSON format.

Replaying Recordings

To replay a session recording, you can use rrweb's player UI or build your own playback interface.

Using rrweb's Player

Here's an example of using rrweb's player to replay a recording:

1. Include the rrweb player script on your page:

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/rrweb@latest/dist/rrweb.min.css"/>
<script src="https://cdn.jsdelivr.net/npm/rrweb@latest/dist/rrweb.min.js"></script>
```

2. Add a container element for the player:

```
<div id="player"></div>
```

3. Initialize the player with your recording data:

```
// if using rrweb npm package
import rrwebPlayer from "rrweb-player";
import "rrweb-player/dist/style.css";

const recordingData = YOUR_RECORDING_DATA

const replayer = new rrwebPlayer({
  target: document.getElementById('player'),
  props: {
    events: recordingData,
    showController: true,
    autoPlay: true,
  },
});
```

This will launch an interactive player UI that allows you to play, pause, rewind, and inspect the recorded session.

Building a Custom Player

You can also use rrweb's APIs to build your own playback UI. Refer to the rrweb documentation for thorough details on how to customize the Replayer to your needs.

Storage and Retention

Session recordings are stored securely in Hyperbrowser's cloud infrastructure. Recordings are retained according to your plan's data retention policy.

Limitations

- Session recordings capture only the visual state of the page. They do not include server-side logs, database changes, or other non-DOM modifications.
- Recordings may not perfectly reproduce complex WebGL or canvas-based animations.

Live View

Hyperbrowser Live View

Hyperbrowser's Live View feature allows you to observe and interact with your browser sessions in real-time. You can use Live View to:

- Debug and troubleshoot your scripts
- Monitor the progress of long-running automations
- Provide a way for end-users to interact with the browser

How it Works

Live View works by providing a secure, authenticated URL that embeds a live stream of the browser session. The URL can be accessed in any modern web browser.

Whenever you create a new session or get the details of an existing session, Hyperbrowser returns a `liveUrl` field with a unique URL tied to that specific session. The URL remains valid as long as the session is active and the token in the URL hasn't expired (Securing Live View).

Node

```
const session = await hbClient.sessions.create();
const liveUrl = session.liveUrl;
```

Python

```
session = hb_client.sessions.create()
live_url = session.live_url
```

The returned `liveUrl` will look something like:

```
https://app.hyperbrowser.ai/live?token=<TOKEN>&keepAlive=true
```

You can now open this URL in a web browser to view the live session.

Embedding Live View

You can embed a Live View into your own web application using an iframe:

```
<iframe src="<LIVE_URL>"></iframe>
```

This is useful for providing a seamless experience to your end-users. For example, you could use an embedded Live View to:

- Show the progress of a web scraping job
- Allow users to complete a complex workflow that requires manual intervention
- Provide a preview of an automated process before committing it

Securing Live View

Live View URLs are secured with authentication and encryption. Only users with the correct URL can access the Live View.

However, anyone with the URL can view (and potentially interact with) the session. Be sure to protect Live View URLs as sensitive secrets, especially if you're embedding them in a public web page.

The token in the `liveUrl` will expire after 12 hours. In this case, you can simply call the GET request for the Sessions API to get the details for the given session id which will also return a new `liveUrl` with a refreshed token.

Web Scraping

Scrape

Scrape any page and get formatted data

The Scrape API allows you to get the data you want from web pages using with a single call. You can scrape page content and capture it's data in various formats.

For detailed usage, checkout the Scrape API Reference

Hyperbrowser exposes endpoints for starting a scrape request and for getting it's status and results. By default, scraping is handled in an asynchronous manner of first starting the job and then checking it's status until it is completed. However, with our SDKs, we provide a simple function that handles the whole flow and returns the data once the job is completed.

Installation

Node

```
npm install @hyperbrowser/sdk
```

or

```
yarn add @hyperbrowser/sdk
```

Python

```
pip install hyperbrowser
```

Usage

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  // Handles both starting and waiting for scrape job response
  const scrapeResult = await client.scrape.startAndWait({
    url: "https://example.com",
  });
  console.log("Scrape result:", scrapeResult);
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.scrape import StartScrapeJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


# Start scraping and wait for completion
scrape_result = client.scrape.start_and_wait(
    StartScrapeJobParams(url="https://example.com")
)
print("Scrape result:", scrape_result)
```

cURL

Start Scrape Job

```
curl -X POST https://app.hyperbrowser.ai/api/scrape \
    -H 'Content-Type: application/json' \
    -H 'x-api-key: <YOUR_API_KEY>' \
    -d '{
        "url": "https://example.com"
    }'
```

Get Scrape Job Status and Data

```
curl https://app.hyperbrowser.ai/api/scrape/{jobId} \
    -H 'x-api-key: <YOUR_API_KEY>'
```

Response

The Start Scrape Job `POST /scrape` endpoint will return a `jobId` in the response which can be used to get information about the job in subsequent requests.

```
{
    "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c"
}
```

The Get Scrape Job `GET /scrape/{jobId}` will return the following data:

```
{
  "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c",
  "status": "completed",
  "data": {
    "metadata": {
      "title": "Example Page",
      "description": "A sample webpage"
    },
    "markdown": "# Example Page\nThis is content...",
  }
}
```

The status of a scrape job can be one of `pending`, `running`, `completed`, `failed`. There can also be other optional fields like `error` with an error message if an error was encountered, and `html` and `links` in the data object depending on which formats are requested for the request.

To see the full schema, checkout the API Reference.

Session Configurations

You can also provide configurations for the session that will be used to execute the scrape job just as you would when creating a new session itself. These could include using a proxy or solving CAPTCHAs.

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const scrapeResult = await client.scrape.startAndWait({
    url: "https://example.com",
    sessionOptions: {
      useProxy: true,
      solveCaptchas: true,
      proxyCountry: "US",
      locales: ["en"],
    },
  });
  console.log("Scrape result:", scrapeResult);
};

main();
```

Python

```python
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.scrape import StartScrapeJobParams
from hyperbrowser.models.session import CreateSessionParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


# Start scraping and wait for completion
scrape_result = client.scrape.start_and_wait(
    StartScrapeJobParams(
        url="https://example.com",
        session_options=CreateSessionParams(use_proxy=True, solve_captchas=True),
    )
)
print("Scrape result:", scrape_result)
```

Using proxy and solving CAPTCHAs will slow down the scrape so use it if necessary.

Scrape Configurations

You can also provide optional parameters for the scrape job itself such as the formats to return, only returning the main content of the page, setting the maximum timeout for navigating to a page, etc.

Node

```javascript
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const scrapeResult = await client.scrape.startAndWait({
    url: "https://example.com",
    scrapeOptions: {
      formats: ["markdown", "html", "links"],
      onlyMainContent: false,
      timeout: 15000,
    },
  });
  console.log("Scrape result:", scrapeResult);
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.scrape import ScrapeOptions, StartScrapeJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


# Start scraping and wait for completion
scrape_result = client.scrape.start_and_wait(
    StartScrapeJobParams(
        url="https://example.com",
        scrape_options=ScrapeOptions(
            formats=["html", "links", "markdown"], only_main_content=False, timeout=5000
        ),
    )
)
print("Scrape result:", scrape_result)
```

For a full reference on the scrape endpoint, checkout the API Reference, or read the Advanced Scraping Guide to see more advanced options for scraping.

Batch Scrape

Batch Scrape works the same as regular scrape, except instead of a single URL, you can provide a list of up to 1,000 URLs to scrape at once.

Batch Scrape is currently only available on the `Ultra` plan.

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const scrapeResult = await client.scrape.batch.startAndWait({
    urls: ["https://example.com", "https://hyperbrowser.ai"],
    scrapeOptions: {
      formats: ["markdown", "html", "links"],
    },
  });
  console.log("Scrape result:", scrapeResult);
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.scrape import ScrapeOptions, StartBatchScrapeJobParams

load_dotenv()

client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


scrape_result = client.scrape.batch.start_and_wait(
    StartBatchScrapeJobParams(
        urls=["https://example.com", "https://hyperbrowser.ai"],
        scrape_options=ScrapeOptions(
            formats=["html", "links", "markdown"]
        ),
    )
)
print("Scrape result:", scrape_result)
```

Response

The Start Batch Scrape Job `POST /scrape/batch` endpoint will return a `jobId` in the response which can be used to get information about the job in subsequent requests.

```
{
    "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c"
}
```

The Get Batch Scrape Job `GET /scrape/batch/{jobId}` will return the following data:

```
{
    "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c",
    "status": "completed",
    "totalScrapedPages": 2,
    "totalPageBatches": 1,
    "currentPageBatch": 1,
    "batchSize": 20,
    "data": [
        {
            "markdown": "Hyperbrowser\n\n[Home](https://hyperbrowser.ai/)...",
            "metadata": {
                "url": "https://www.hyperbrowser.ai/",
                "title": "Hyperbrowser",
                "viewport": "width=device-width, initial-scale=1",
                "link:icon": "https://www.hyperbrowser.ai/favicon.ico",
                "sourceURL": "https://hyperbrowser.ai",
                "description": "Infinite Browsers"
            },
            "url": "hyperbrowser.ai",
            "status": "completed",
            "error": null
        },
        {
            "markdown": "Example Domain\n\n# Example Domain...",
            "metadata": {
                "url": "https://www.example.com/",
                "title": "Example Domain",
                "viewport": "width=device-width, initial-scale=1",
                "sourceURL": "https://example.com"
            },
            "url": "example.com",
            "status": "completed",
            "error": null
        }
    ]
}
```

The status of a batch scrape job can be one of `pending`, `running`, `completed`, `failed`. The results of all the scrapes will be an array in the `data` field of the response. Each scraped page will be returned in the order of the initial provided urls, and each one will have its own status and information.

To see the full schema, checkout the API Reference.

As with the single scrape, by default, batch scraping is handled in an asynchronous manner of first starting the job and then checking it's status until it is completed. However, with our SDKs, we provide a simple function (`client.scrape.batch.startAndWait`) that handles the whole flow and returns the data once the job is completed.

Crawl

Crawl a website and it's links to extract all it's data

The Crawl API allows you to crawl through an entire website and get all it's data with a single call.

For detailed usage, checkout the Crawl API Reference

Hyperbrowser exposes endpoints for starting a crawl request and for getting it's status and results. By default, crawling is handled in an asynchronous manner of first starting the job and then checking it's status until it is completed. However, with our SDKs, we provide a simple function that handles the whole flow and returns the data once the job is completed.

Installation

Node

```
npm install @hyperbrowser/sdk
```

or

```
yarn add @hyperbrowser/sdk
```

Python

```
pip install hyperbrowser
```

Usage

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  // Handles both starting and waiting for crawl job response
  const crawlResult = await client.crawl.startAndWait({
    url: "https://hyperbrowser.ai",
  });
  console.log("Crawl result:", crawlResult);
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.crawl import StartCrawlJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


# Start crawling and wait for completion
crawl_result = client.crawl.start_and_wait(
    StartCrawlJobParams(url="https://hyperbrowser.ai")
)
print("Crawl result:", crawl_result)
```

cURL

Start Crawl Job

```
curl -X POST https://app.hyperbrowser.ai/api/crawl \
    -H 'Content-Type: application/json' \
    -H 'x-api-key: <YOUR_API_KEY>' \
    -d '{
        "url": "https://hyperbrowser.ai"
    }'
```

Get Crawl Job Status and Data

```
curl https://app.hyperbrowser.ai/api/crawl/{jobId} \
    -H 'x-api-key: <YOUR_API_KEY>'
```

Response

The Start Crawl Job `POST /crawl` endpoint will return a `jobId` in the response which can be used to get information about the job in subsequent requests.

```
{
    "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c"
}
```

The Get Crawl Job `GET /crawl/{jobId}` will return the following data:

```
{
  "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c",
  "status": "completed",
  "totalCrawledPages": 2,
  "totalPageBatches": 1,
  "currentPageBatch": 1,
  "batchSize": 20,
  "data": [
    {
      "url": "https://example.com",
      "status": "completed",
      "metadata": {
        "title": "Example Page",
        "description": "A sample webpage"
      },
      "markdown": "# Example Page\nThis is content...",
    },
    ...
  ]
}
```

The status of a crawl job can be one of `pending`, `running`, `completed`, `failed` . There can also be other optional fields like `error` with an error message if an error was encountered, and `html` and `links` in the data object depending on which formats are requested for the request.

Unlike the scrape endpoint, the crawl endpoint returns a list in the data field with the all the pages that were crawled in the current page batch. The SDKs also provide a function which will start the crawl job, wait until it's complete, and return all the crawled pages for the entire crawl.

Each crawled page has it's own status of `completed` or `failed` and can have it's own error field, so be cautious of that.

To see the full schema, checkout the API Reference.

Additional Crawl Configurations

The crawl endpoint provides additional parameters you can provide to tailor the crawl to your needs. You can narrow down the pages crawled by setting a limit to the maximum number of pages visited, only including paths that match a certain pattern, excluding paths that match another pattern, etc.

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  // Handles both starting and waiting for crawl job response
  const crawlResult = await client.crawl.startAndWait({
    url: "https://hyperbrowser.ai",
    maxPages: 5,
    includePatterns: ["/blogs/*"],
  });
  console.log("Crawl result:", crawlResult);
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.crawl import StartCrawlJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


# Start crawling and wait for completion
crawl_result = client.crawl.start_and_wait(
    StartCrawlJobParams(
        url="https://hyperbrowser.ai",
        max_pages=5,
        include_patterns: ["/blogs/*"],
    )
)
print("Crawl result:", crawl_result)
```

Session Configurations

You can also provide configurations for the session that will be used to execute the crawl job just as you would when creating a new session itself. These could include using a proxy or solving CAPTCHAs.

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const crawlResult = await client.crawl.startAndWait({
    url: "https://example.com",
    sessionOptions: {
      useProxy: true,
      solveCaptchas: true,
      proxyCountry: "US",
      locales: ["en"],
    },
  });
  console.log("Crawl result:", crawlResult);
};

main();
```

Python

```python
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.crawl import StartCrawlJobParams
from hyperbrowser.models.session import CreateSessionParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))



# Start crawling and wait for completion
crawl_result = client.crawl.start_and_wait(
    StartCrawlJobParams(
        url="https://example.com",
        session_options=CreateSessionParams(use_proxy=True, solve_captchas=True),
    )
)
print("Crawl result:", crawl_result)
```

Using proxy and solving CAPTCHAs will slow down the crawl so use it only if necessary.

Scrape Configurations

You can also provide optional scrape options for the crawl job such as the formats to return, only returning the main content of the page, setting the maximum timeout for navigating to a page, etc.

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const crawlResult = await client.crawl.startAndWait({
    url: "https://example.com",
    scrapeOptions: {
      formats: ["markdown", "html", "links"],
      onlyMainContent: false,
      timeout: 10000,
    },
  });
  console.log("Crawl result:", crawlResult);
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.scrape import ScrapeOptions
from hyperbrowser.models.crawl import StartCrawlJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


# Start crawling and wait for completion
crawl_result = client.crawl.start_and_wait(
    StartCrawlJobParams(
        url="https://example.com",
        scrape_options=ScrapeOptions(
            formats=["html", "links", "markdown"], only_main_content=False, timeout=1000
        ),
    )
)
print("Crawl result:", crawl_result)
```

For a full reference on the crawl endpoint, checkout the API Reference, or read the Advanced Scraping Guide to see more advanced options for scraping.

Extract

Extract data from pages using AI

The Extract API allows you to get data in a structured format for any provided URLs with a single call.

For detailed usage, checkout the Extract API Reference

Hyperbrowser exposes endpoints for starting an extract request and for getting it's status and results. By default, extracting is handled in an asynchronous manner of first starting the job and then checking it's status until it is completed. However, with our SDKs, we provide a simple function that handles the whole flow and returns the data once the job is completed.

Installation

Node

```
npm install @hyperbrowser/sdk
```

or

```
yarn add @hyperbrowser/sdk
```

Python

```
pip install hyperbrowser
```

Usage

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";
import { z } from "zod";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const schema = z.object({
    productName: z.string(),
    productOverview: z.string(),
    keyFeatures: z.array(z.string()),
    pricing: z.array(
      z.object({
        plan: z.string(),
        price: z.string(),
        features: z.array(z.string()),
      })
    ),
  });

  // Handles both starting and waiting for extract job response
  const result = await client.extract.startAndWait({
    urls: ["https://hyperbrowser.ai"],
    prompt:
      "Extract the product name, an overview of the product, its key features, and a list
    schema: schema,
  });

  console.log("result", JSON.stringify(result, null, 2));
};

main();
```

Python

```
import os
import json
from typing import List
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.extract import StartExtractJobParams
from pydantic import BaseModel


# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


class PricingSchema(BaseModel):
    plan: str
    price: str
    features: List[str]


class ExtractSchema(BaseModel):
    product_name: str
    product_overview: str
    key_features: List[str]
    pricing: List[PricingSchema]


def main():
    result = client.extract.start_and_wait(
        params=StartExtractJobParams(
            urls=["https://hyperbrowser.ai"],
            prompt="Extract the product name, an overview of the product, its key feature
            schema=ExtractSchema,
        )
    )
    print("result:", json.dumps(result.data, indent=2))


main()
```

cURL

Start Extract Job

```
curl -X POST https://app.hyperbrowser.ai/api/extract \
    -H 'Content-Type: application/json' \
    -H 'x-api-key: <YOUR_API_KEY>' \
    -d '{
        "urls": ["https://hyperbrowser.ai"],
        "prompt": "Extract the product name, an overview of the product, its key features
        "schema": {
          "type": "object",
          "properties": {
            "productName": {
              "type": "string"
            },
            "productOverview": {
              "type": "string"
            },
            "keyFeatures": {
              "type": "array",
              "items": {
                "type": "string"
              }
            },
            "pricing": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "plan": {
                    "type": "string"
                  },
                  "price": {
                    "type": "string"
                  },
                  "features": {
                    "type": "array",
                    "items": {
                      "type": "string"
                    }
                  }
                },
                "required": [
                  "plan",
                  "price",
                  "features"
                ]
              }
            }
          },
          "required": [
            "productName",
            "productOverview",
            "keyFeatures",
            "pricing"
          ]
        }
    }'
```

Get Extract Job Status and Data

```
curl https://app.hyperbrowser.ai/api/extract/{jobId} \
    -H 'x-api-key: <YOUR_API_KEY>'
```

You can configure the extract request with the following parameters:

- `urls` - A required list of urls you want to use to extract data from. To allow crawling for any of the urls provided in the list, simply add `/*` to the end of the url (`https://hyperbrowser.ai/*`). This will crawl other pages on the site with the same origin and find relevant pages to use for the extraction context.
- `schema` - A strict json schema you want the returned data to be structured as. Gives the best results.
- `prompt` - A prompt describing how you want the data structured. Useful if you don't have a specific schema in mind.
- `maxLinks` - The maximum number of links to look for if performing a crawl for any given url.
- `waitFor` - A delay in milliseconds to wait after the page loads before initiating the scrape to get data for extraction from page. This can be useful for allowing dynamic content to fully render. This is also useful for waiting to detect CAPTCHAs on the page if you have `solveCaptchas` set to true in the `sessionOptions`.
- `sessionOptions` - Options for the session.

You must provide either a `schema` or a `prompt` in your request, and if both are provided the schema takes precedence.

For the Node SDK, you can simply pass in a zod schema for ease of use or an actual json schema. For the Python SDK, you can pass in a pydantic model or an actual json schema.

Ensure that the root level of the schema is `type: "object"`.

Response

The Start Extract Job `POST /extract` endpoint will return a `jobId` in the response which can be used to get information about the job in subsequent requests.

```
{
    "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c"
}
```

The Get Extract Job `GET /extract/{jobId}` will return the following data:

```
{
  "jobId": "962372c4-a140-400b-8c26-4ffe21d9fb9c",
  "status": "completed",
  "data": {
    "pricing": [
      {
        "plan": "Free",
        "price": "$0",
        "features": [
          "3,000 Credits Included",
          "5 Concurrent Browsers",
          "7 Days Data Retention",
          "Basic Stealth Mode"
        ]
      },
      {
        "plan": "Startup",
        "price": "$30 / Month",
        "features": [
          "18,000 Credits Included",
          "25 Concurrent Browsers",
          "30 Day Data Retention",
          "Auto Captcha Solving",
          "Basic Stealth Mode"
        ]
      },
      {
        "plan": "Scale",
        "price": "$100 / Month",
        "features": [
          "60,000 Credits Included",
          "100 Concurrent Browsers",
          "30 Day Data Retention",
          "Auto Captcha Solving",
          "Advanced Stealth Mode"
        ]
      },
      {
        "plan": "Enterprise",
        "price": "Custom",
        "features": [
          "Volume discounts available",
          "Premium Support",
          "HIPAA/SOC 2",
          "250+ Concurrent Browsers",
          "180+ Day Data Retention",
          "Auto Captcha Solving",
          "Advanced Stealth Mode"
        ]
      }
    ],
    "keyFeatures": [
      "Run headless browsers to automate tasks like web scraping, testing, and form filling.
      "Use browsers to scrape and structure web data at scale for analysis and insights.",
      "Integrate with AI agents to enable browsing, data collection, and interaction with we
      "Automatically solve captchas to streamline automation workflows.",
      "Operate browsers in stealth mode to bypass bot detection and stay undetected.",
      "Manage browser sessions with logging, debugging, and secure resource isolation."
    ],
    "productName": "Hyperbrowser",
    "productOverview": "Hyperbrowser is a platform for running and scaling headless browsers
  }
}
```

The status of an extract job can be one of `pending`, `running`, `completed`, `failed` . There can also be an optional `error` field with an error message if an error was encountered.

To see the full schema, checkout the API Reference.

Session Configurations

You can also provide configurations for the session that will be used to execute the extract job just as you would when creating a new session itself. These could include using a proxy or solving CAPTCHAs. To see the full list of session configurations, checkout the Session API Reference.

Node

```
import { config } from "dotenv";
import { z } from "zod";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const schema = z.object({
    productName: z.string(),
    productOverview: z.string(),
    keyFeatures: z.array(z.string()),
    pricing: z.array(
      z.object({
        plan: z.string(),
        price: z.string(),
        features: z.array(z.string()),
      })
    ),
  });

  const result = await client.extract.startAndWait({
    urls: ["https://hyperbrowser.ai"],
    prompt:
      "Extract the product name, an overview of the product, its key features, and its pr
    schema: schema,
    // include sessionOptions
    sessionOptions: {
      useProxy: true,
      solveCaptchas: true,
    },
  });

  console.log("result", JSON.stringify(result, null, 2));
};

main();
```

Python

```
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.extract import StartExtractJobParams
from pydantic import BaseModel


load_dotenv()


client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


class PricingSchema(BaseModel):
    plan: str
    price: str
    features: List[str]


class ExtractSchema(BaseModel):
    product_name: str
    product_overview: str
    key_features: List[str]
    pricing: List[PricingSchema]


def main():
    result = client.extract.start_and_wait(
        params=StartExtractJobParams(
            urls=["https://hyperbrowser.ai"],
            prompt="Extract the product name, an overview of the product, its key feature
            schema=ExtractSchema,
            # include session_options
            session_options=CreateSessionParams(use_proxy=True, solve_captchas=True),
        )
    )
    print("result:", json.dumps(result.data, indent=2))


main()
```

Hyperbrowser's CAPTCHA solving and proxy usage features require being on a `PAID` plan.

Using proxy and solving CAPTCHAs will slow down the page scraping in the extract job so use it only if necessary.

For a full reference on the extract endpoint, checkout the API Reference.

Billing

Credit usage for extract jobs are charged based on the total number of output tokens used for successful extract jobs. Each output token costs **0.015 credits** which comes out to $30 per million output tokens.

Guides

Scraping

Advanced Options for Hyperbrowser Scraping

Basic Usage

With supplying just a url, you can easily extract the contents of a page in markdown format with the `/scrape` endpoint.

Node

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  // Handles both starting and waiting for scrape job response
  const scrapeResult = await client.scrape.startAndWait({
    url: "https://example.com",
  });
  console.log("Scrape result:", scrapeResult);
};

main();
```

Python

```python
import os
from dotenv import load_dotenv
from hyperbrowser import Hyperbrowser
from hyperbrowser.models.scrape import StartScrapeJobParams

# Load environment variables from .env file
load_dotenv()

# Initialize Hyperbrowser client
client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


# Start scraping and wait for completion
scrape_result = client.scrape.start_and_wait(
    StartScrapeJobParams(url="https://example.com")
)
print("Scrape result:", scrape_result)
```

cURL

Start Scrape Job

```
curl -X POST https://app.hyperbrowser.ai/api/scrape \
    -H 'Content-Type: application/json' \
    -H 'x-api-key: <YOUR_API_KEY>' \
    -d '{
        "url": "https://example.com"
    }'
```

Get Scrape Job Status and Data

```
curl https://app.hyperbrowser.ai/api/scrape/{jobId} \
    -H 'x-api-key: <YOUR_API_KEY>'
```

Now, let's take an in depth look at all the provided options for scraping.

Scrape Options

`formats`

- **Type**: `array`
- **Items**: `string`
- **Enum**: `["html", "links", "markdown", "screenshot"]`
- **Description**: Choose the formats to include in the API response:

    - `html` - Returns the scraped content as HTML.
    - `links` - Includes a list of links found on the page.
    - `markdown` - Provides the content in Markdown format.
    - `screenshot` - Provides a screenshot of the page.
- **Default**: `["markdown"]`

`includeTags`

- **Type**: `array`
- **Items**: `string`
- **Description**: Provide an array of HTML tags, classes, or IDs to include in the scraped content. Only elements matching these selectors will be returned.
- **Default**: `undefined`

`excludeTags`

- **Type**: `array`
- **Items**: `string`
- **Description**: Provide an array of HTML tags, classes, or IDs to exclude from the scraped content. Elements matching these selectors will be omitted from the response.
- **Default**: `undefined`

`onlyMainContent`

- **Type**: `boolean`
- **Description**: When set to `true` (default), the API will attempt to return only the main content of the page, excluding common elements like headers, navigation menus, and footers. Set to `false` to return the full page content.
- **Default**: `true`

`waitFor`

- **Type**: `number`
- **Description**: Specify a delay in milliseconds to wait after the page loads before initiating the scrape. This can be useful for allowing dynamic content to fully render. This is also useful for waiting to detect CAPTCHAs on the page if you have `solveCaptchas` set to true in the `sessionOptions`.
- **Default**: `0`

`timeout`

- **Type**: `number`

- **Description**: Specify the maximum time in milliseconds to wait for the page to load before timing out. This would be like doing:

```
await page.goto("https://example.com", { waitUntil: "load", timeout: 30000 })
```

- **Default**: `30000` (30 seconds)

`waitUntil`

- **Type**: `string`
- **Enum**: `["load", "domcontentloaded", "networkidle"]`
- **Description**: Specify the condition to wait for the page to load:
  - `domcontentloaded`: Wait until the HTML is fully parsed and DOM is ready
  - `load` - Wait until DOM and all resources are completely loaded
  - `networkidle` - Wait until no more network requests occur for a certain period of time
- **Default**: `load`

`screenshotOptions`

- **Type**: `object`
- **Properties**:
  - **fullPage** - Take screenshot of the full page beyond the viewport
    - **Type**: `boolean`
    - **Default**: `false`
  - **format** - The image type of the screenshot
    - **Type**: `string`
    - **Enum**: `["webp", "jpeg", "png"]`
    - **Default**: `webp`
- **Description**: Configurations for the returned screenshot. Only applicable if `screenshot` is provided in the `formats` array.

Session Options

Enabling Stealth Mode with `useStealth`

- **Type**: `boolean`
- **Description**: When set to `true`, the session will be launched in stealth mode, which employs various techniques to make the browser harder to detect as an automated tool.

- **Default**: `false`

Using a Proxy with `useProxy`

- **Type**: `boolean`
- **Description**: When set to `true`, the session will be launched with a proxy server.
- **Default**: `false`

Specifying a Custom Proxy Server with `proxyServer`

- **Type**: `string`
- **Description**: The hostname or IP address of the proxy server to use for the session. This option is only used when `useProxy` is set to `true`.
- **Default**: `undefined`

Providing Proxy Server Authentication with `proxyServerUsername` and `proxyServerPassword`

- **Type**: `string`
- **Description**: The username and password to use for authenticating with the proxy server, if required. These options are only used when `useProxy` is set to `true` and the proxy server requires authentication.
- **Default**: `undefined`

Selecting a Proxy Location with `proxyCountry`

- **Type**: `string`
- **Enum**: `["US", "GB", "CA", ...]`
- **Description**: The country where the proxy server should be located.
- **Default**: `"US"`

Specifying Operating Systems with `operatingSystems`

- **Type**: `array`
- **Items**: `string`
- **Enum**: `["windows", "android", "macos", "linux", "ios"]`
- **Description**: An array of operating systems to use for fingerprinting.
- **Default**: `undefined`

Choosing Device Types with `device`

- **Type**: `array`

- **Items**: `string`
- **Enum**: `["desktop", "mobile"]`
- **Description**: An array of device types to use for fingerprinting.
- **Default**: `undefined`

Selecting Browser Platforms with `platform`

- **Type**: `array`
- **Items**: `string`
- **Enum**: `["chrome", "firefox", "safari", "edge"]`
- **Description**: An array of browser platforms to use for fingerprinting.
- **Default**: `undefined`

Setting Browser Locales with `locales`

- **Type**: `array`
- **Items**: `string`
- **Enum**: `["en", "es", "fr", ...]`
- **Description**: An array of browser locales to specify the language for the browser.
- **Default**: `["en"]`

Customizing Screen Resolution with `screen`

- **Type**: `object`
- **Properties**:
  - `width` (number, default 1280): The screen width in pixels.
  - `height` (number, default 720): The screen height in pixels.
- **Description**: An object specifying the screen resolution to emulate in the session.

Solving CAPTCHAs Automatically with `solveCaptchas`

- **Type**: `boolean`
- **Description**: When set to `true`, the session will attempt to automatically solve any CAPTCHAs encountered during the session.
- **Default**: `false`

Blocking Ads with `adblock`

- **Type**: `boolean`

- **Description**: When set to `true`, the session will attempt to block ads and other unwanted content during the session.
- **Default**: `false`

Blocking Trackers with `trackers`

- **Type**: `boolean`
- **Description**: When set to `true`, the session will attempt to block web trackers and other privacy-invasive technologies during the session.
- **Default**: `false`

Blocking Annoyances with `annoyances`

- **Type**: `boolean`
- **Description**: When set to `true`, the session will attempt to block common annoyances like pop-ups, overlays, and other disruptive elements during the session.
- **Default**: `false`

Example

By configuring these options when making a scrape request, you can control the format and content of the scraped data, as well as the behavior of the scraper itself.

For example, to scrape a page with the following:

- In stealth mode
- With CAPTCHA solving
- Return only the main content as HTML
- Exclude any `<span>` elements
- Wait 2 seconds after the page loads and before scraping

```
curl -X POST https://app.hyperbrowser.ai/api/scrape \
    -H 'Content-Type: application/json' \
    -H 'x-api-key: <YOUR_API_KEY>' \
    -d '{
            "url": "https://example.com",
            "sessionOptions": {
                    "useStealth": true,
                    "solveCaptchas": true
            },
            "scrapeOptions": {
                    "formats": ["html"],
                    "onlyMainContent": true,
                    "excludeTags": ["span"],
                    "waitFor": 2000
            }
    }'
```

Crawl a Site

Instead of just scraping a single page, you might want to get all the content across multiple pages on a site. The `/crawl` endpoint is perfect for such a task. You can use the same `sessionOptions` and `scrapeOptions` as before for this endpoint as well. The crawl endpoint does have some extra parameters that are used to tailor the crawl to your scraping needs.

Crawl Options

Limiting the Number of Pages to Crawl with `maxPages`

- **Type**: `integer`
- **Minimum**: 1
- **Description**: The maximum number of pages to crawl before stopping.

Following Links with `followLinks`

- **Type**: `boolean`
- **Default**: `true`
- **Description**: When set to `true`, the crawler will follow links found on the pages it visits, allowing it to discover new pages and expand the scope of the crawl. When set to `false`, the crawler will only visit the starting URL and any explicitly specified pages, without following any additional links.

Ignoring the Sitemap with `ignoreSitemap`

- **Type**: `boolean`

- **Default**: `false`
- **Description**: When set to `true`, the crawler will not pre-generate a list of urls from potential sitemaps it finds. The crawler will try to locate sitemaps beginning at the base URL of the URL provided in the `url` param.

Excluding Pages with `excludePatterns`

- **Type**: `array`
- **Items**: `string`
- **Description**: An array of regular expressions or wildcard patterns specifying which URLs should be excluded from the crawl. Any pages whose URLs' path match one of these patterns will be skipped.

Including Pages with `includePatterns`

- **Type**: `array`
- **Items**: `string`
- **Description**: An array of regular expressions or wildcard patterns specifying which URLs should be included in the crawl. Only pages whose URLs' path match one of these path patterns will be visited.

Example

By configuring these options when initiating a crawl, you can control the scope and behavior of the crawler to suit your specific needs.

For example, to crawl a site with the following:

- Maximum of 5 pages
- Only include `/blog` pages
- Return only the main content as HTML
- Exclude any `<span>` elements
- Wait 2 seconds after the page loads and before scraping

```
curl -X POST https://app.hyperbrowser.ai/api/crawl \
    -H 'Content-Type: application/json' \
    -H 'x-api-key: <YOUR_API_KEY>' \
    -d '{
            "url": "https://example.com",
            "maxPages": 5,
            "includePatterns": ["/blog/*"],
            "scrapeOptions": {
                    "formats": ["html"],
                    "onlyMainContent": true,
                    "excludeTags": ["span"],
                    "waitFor": 2000
            }
    }'
```

AI Function Calling

Using Hyperbrowser with OpenAI and Anthropic Function Tools

Hyperbrowser integrates seamlessly with OpenAI and Anthropic's function calling APIs, enabling you to enhance your AI applications with web scraping and crawling capabilities. This guide will walk you through setting up and using Hyperbrowser's scrape and crawl tools with OpenAI and Anthropic.

Setup

Installation

First, install the necessary dependencies to run our script.

Node

```
npm install @hyperbrowser/sdk dotenv openai
```

Python

```
pip install hyperbrowser openai python-dotenv
```

Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` . You will also need an `OPENAI_API_KEY`.

Code

Node

```
import OpenAI from "openai";
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { WebsiteCrawlTool, WebsiteScrapeTool } from "@hyperbrowser/sdk/tools";
import { config } from "dotenv";

config();

// Initialize clients
const hb = new Hyperbrowser({ apiKey: process.env.HYPERBROWSER_API_KEY });
const oai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
});

async function handleToolCall(tc) {
  console.log("Handling tool call");

  try {
    const args = JSON.parse(tc.function.arguments);
    console.log(`Tool call ID: ${tc.id}`);
    console.log(`Function name: ${tc.function.name}`);
    console.log(`Function args: ${JSON.stringify(args, null, 2)}`);
    console.log("-".repeat(50));

    if (
      tc.function.name === WebsiteCrawlTool.openaiToolDefinition.function.name
    ) {
      const response = await WebsiteCrawlTool.runnable(hb, args);
      return {
        tool_call_id: tc.id,
        content: response,
        role: "tool",
      };
    } else if (
      tc.function.name === WebsiteScrapeTool.openaiToolDefinition.function.name
    ) {
      const response = await WebsiteScrapeTool.runnable(hb, args);
      return {
        tool_call_id: tc.id,
        content: response,
```

Python

```python
import json
import os

from hyperbrowser import Hyperbrowser
from hyperbrowser.tools import WebsiteCrawlTool, WebsiteScrapeTool

from openai import OpenAI
from openai.types.chat import (
    ChatCompletionMessageToolCall,
    ChatCompletionMessageParam,
    ChatCompletionToolMessageParam,
)
from dotenv import load_dotenv

load_dotenv()

oai = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
hb = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))


def handle_tool_call(
    tc: ChatCompletionMessageToolCall,
) -> ChatCompletionToolMessageParam:
    print("Handling tool call")

    try:
        args = json.loads(tc.function.arguments)
        print(f"Tool call ID: {tc.id}")
        print(f"Function name: {tc.function.name}")
        print(f"Function args: {args}")
```

Hyperbrowser exposes `WebsiteCrawlTool` and `WebsiteScrapeTool` to be used for OpenAI and Anthropic function calling. Each class provides a tool definition for both OpenAI and Anthropic that defines the schema which can be passed into the tools argument. Then, in the `handleToolCall` function, we parse the tool call arguments and dispatch the appropriate tool class `runnable` function based on the function name which will return the result of the scrape or crawl in formatted markdown.

Extract Information with an LLM

Use Hyperbrowser to scrape a wikipedia page and extract information with an LLM

In this guide, we'll use Hyperbrowser's Node.js SDK to get formatted data from a Wikipedia page and then feed it into an LLM like ChatGPT to extract the information we want. Our goal is to get a list of the most populous cities.

Setup

First, lets create a new Node.js project.

```
mkdir wiki-scraper && cd wiki-scraper
npm init -y
```

Installation

Next, let's install the necessary dependencies to run our script.

```
npm install @hyperbrowser/sdk dotenv openai zod
```

Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` . You will also need an `OPENAI_API_KEY` to use ChatGPT to extract information from our scraped data.

Code

Next, create a new file `scraper.js` and add the following code:

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";
import { z } from "zod";
import OpenAI from "openai";
import { zodResponseFormat } from "openai/helpers/zod";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});
const openai = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });

const CitySchema = z.object({
  city: z.string(),
  country: z.string(),
  population: z.number(),
  rank: z.number(),
});

const ResponseSchema = z.object({ cities: z.array(CitySchema) });

const SYSTEM_PROMPT = `You are a helpful assistant that can extract information from markdow
Ensure the output adheres to the following:
- city: The name of the city
- country: The name of the country
- population: The population of the city
- rank: The rank of the city

Provide the extracted data as a JSON object. Parse the Markdown content carefully to identif
`;

const main = async () => {
  console.log("Started scraping");
  const scrapeResult = await client.scrape.startAndWait({
    url: "https://en.wikipedia.org/wiki/List_of_largest_cities",
    scrapeOptions: {
      // Only return the markdown for the scraped data
      formats: ["markdown"],
      // Only include the table element with class `wikitable` from the page
      includeTags: [".wikitable"],
      // Remove any img tags from the table
      excludeTags: ["img"],
    },
  });
  console.log("Finished scraping");
  if (scrapeResult.status === "failed") {
    console.error("Scrape failed:", scrapeResult.error);
    return;
  }
  if (!scrapeResult.data.markdown) {
    console.error("No markdown data found in the scrape result");
    return;
  }

  console.log("Extracting data from markdown");
```

```
    const completion = await openai.beta.chat.completions.parse({
      model: "gpt-4o-mini",
      messages: [
        {
          role: "system",
          content: SYSTEM_PROMPT,
        },
        { role: "user", content: scrapeResult.data.markdown },
      ],
      response_format: zodResponseFormat(ResponseSchema, "cities"),
    });
    console.log("Finished extracting data from markdown");

    const cities = completion.choices[0].message.parsed;

    const data = JSON.stringify(cities, null, 2);
    fs.writeFileSync("cities.json", data);
  };

  main();
```

With just a single call to the SDKs crawl `startAndWait` function, we can get back the exact information we need from the page in properly formatted markdown. To make sure we narrow down the data we get back to just the information we need, we make sure to only include the `wikiTable` class element and remove any unnecessary image tags.

Once we have the markdown text, we can simply just pass it into the request to the `parse` function from the openai library with the `response_format` we want and we will have our list of the most populous cities.

Run the Scraper

Once you have the code copied, you can run the script with:

```
node scraper.js
```

If everything completes successfully, you should see a `cities.json` file in your project directory with the data in this format:

```
{
  "cities": [
    {
      "city": "Tokyo",
      "country": "Japan",
      "population": 37468000,
      "rank": 1
    },
    {
      "city": "Delhi",
      "country": "India",
      "population": 28514000,
      "rank": 2
    },
    {
      "city": "Shanghai",
      "country": "China",
      "population": 25582000,
      "rank": 3
    },
    ...
  ]
}
```

Next Steps

This is a simple example, but you can adapt it to scrape more complex data from other sites, or crawl entire websites.

Using Hyperbrowser Session

Using Hyperbrowser's session

In this guide, we will see how to use Hyperbrowser and Puppeteer to create a new session, connect to it, and scrape current weather data.

Setup

First, lets create a new Node.js project.

```
mkdir weather-scraper && cd weather-scraper
npm init -y
```

Installation

Next, let's install the necessary dependencies to run our script.

```
npm install @hyperbrowser/sdk puppeteer-core dotenv
```

Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY`.

Code

Next, create a new file `index.js` and add the following code:

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";
import { connect } from "puppeteer-core";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const main = async () => {
  const location = process.argv[2];
  if (!location) {
    console.error("Please provide a location as a command line argument");
    process.exit(1);
  }

  console.log("Starting session");
  const session = await client.sessions.create();
  console.log("Session created:", session.id);

  try {
    const browser = await connect({ browserWSEndpoint: session.wsEndpoint });

    const [page] = await browser.pages();

    await page.goto("https://openweathermap.org/city", {
      waitUntil: "load",
      timeout: 20_000,
    });
    await page.waitForSelector(".search-container", {
      visible: true,
      timeout: 10_000,
    });
    await page.type(".search-container input", location);
    await page.click(".search button");
    await page.waitForSelector(".search-dropdown-menu", {
      visible: true,
      timeout: 10_000,
    });

    const [response] = await Promise.all([
      page.waitForNavigation(),
      page.click(".search-dropdown-menu li:first-child"),
    ]);

    await page.waitForSelector(".current-container", {
      visible: true,
      timeout: 10_000,
    });
    const locationName = await page.$eval(
      ".current-container h2",
      (el) => el.textContent
    );
    const currentTemp = await page.$eval(
      ".current-container .current-temp",
      (el) => el.textContent
    );
    const description = await page.$eval(
      ".current-container .bold",
      (el) => el.textContent
    );

    const windInfo = await page.$eval(".weather-items .wind-line", (el) =>
      el.textContent.trim()
    );
    const pressureInfo = await page.$eval(
      ".weather-items li:nth-child(2)",
      (el) => el.textContent.trim()
    );
    const humidityInfo = await page.$eval(
      ".weather-items li:nth-child(3)",
      (el) => el.textContent.trim()?.split(":")[1]
    );
    const dewpoint = await page.$eval(
      ".weather-items li:nth-child(4)",
      (el) => el.textContent.trim()?.split(":")[1]
    );
    const visibility = await page.$eval(
      ".weather-items li:nth-child(5)",
      (el) => el.textContent.trim()?.split(":")[1]
    );

    console.log("\nWeather Information:");
    console.log("------------------");
    console.log(`Location: ${locationName}`);
    console.log(`Temperature: ${currentTemp}`);
    console.log(`Conditions: ${description}`);
    console.log(`Wind: ${windInfo}`);
    console.log(`Pressure: ${pressureInfo}`);
    console.log(`Humidity: ${humidityInfo}`);
    console.log(`Dew Point: ${dewpoint}`);
    console.log(`Visibility: ${visibility}`);
    console.log("------------------\n");

    await page.screenshot({ path: "screenshot.png" });
    await page.close();
    await browser.close();
  } catch (error) {
    console.error(`Encountered an error: ${error}`);
  } finally {
    await client.sessions.stop(session.id);
    console.log("Session stopped:", session.id);
  }
};

main().catch((error) => {
  console.error(`Encountered an error: ${error}`);
});
```

Run the Scraper

To run the weather scraper:

1. Open a terminal and navigate to your project directory

2. Run the script with a location argument:

```
node index.js "New York"
```

Replace `"New York"` with the location you want weather data for.

The script will:

1. Create a new Hyperbrowser session
2. Launch a Puppeteer browser and connect to the session
3. Navigate to the OpenWeatherMap city page
4. Search for the specified location and hit the Search button
5. Select the first option from a list in a dropdown menu and navigate to that page
6. Scrape the current weather data from the page
7. Print the weather information to the console
8. Save a screenshot of the page
9. Close the browser and stop the Hyperbrowser session

You should see output like:

```
Weather Information:
------------------
Location: New York City, US
Temperature: 9°C
Conditions: overcast clouds
Wind: Gentle breeze, 3.6 m/s, west-southwest
Pressure: 1013 hPa
Humidity: 81%
Dew Point: 6°C
Visibility: 10 km
------------------
```

And a `screenshot.png` file saved in your project directory.

How it Works

Let's break down the key steps:

1. We import the required libraries and load the environment variables
2. We create a new Hyperbrowser client with the API key
3. We start a new Hyperbrowser session with `client.sessions.create()`
4. We launch a Puppeteer browser and connect it to the Hyperbrowser session

5. We navigate to the OpenWeatherMap city page
6. We search for the location provided as a command line argument
7. We wait for the search results and click the first result
8. We scrape the weather data from the page using Puppeteer's `page.$eval` method
9. We print the scraped data, take a screenshot, and save it to disk
10. Finally, we close the browser and stop the Hyperbrowser session

Next Steps

This example demonstrates a basic weather scraping workflow using a Hyperbrowser session. You can expand on it to:

- Accept multiple locations and fetch weather data for each
- Get the 8-day forecast for the location
- Schedule the script to run periodically and save historical weather data

CAPTCHA Solving

Using Hyperbrowser's CAPTCHA Solving

Hyperbrowser's CAPTCHA solving feature requires being on a `PAID` plan.

In this guide, we will see how to use Hyperbrowser and its integrated CAPTCHA solver to scrape Today's Top Deals from Amazon without being blocked.

Setup

First, lets create a new Node.js project.

```
mkdir amazon-deals-scraper && cd amazon-deals-scraper
npm init -y
```

Installation

Next, let's install the necessary dependencies to run our script.

```
npm install @hyperbrowser/sdk puppeteer-core dotenv
```

Setup your Environment

To use Hyperbrowser with your code, you will need an API Key. You can get one easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY`.

Code

Next, create a new file `index.js` and add the following code:

```
import { Hyperbrowser } from "@hyperbrowser/sdk";
import { config } from "dotenv";
import { connect } from "puppeteer-core";

config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

const sleep = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

const main = async () => {
  console.log("Starting session");
  const session = await client.sessions.create({
    solveCaptchas: true,
    adblock: true,
    annoyances: true,
    trackers: true,
  });
  console.log("Session created:", session.id);

  try {
    const browser = await connect({ browserWSEndpoint: session.wsEndpoint });

    const [page] = await browser.pages();

    await page.goto("https://amazon.com/deals", {
      waitUntil: "load",
      timeout: 20_000,
    });

    const pageTitle = await page.title();
    console.log("Navigated to Page:", pageTitle);

    await sleep(10_000);

    const products = await page.evaluate(() => {
      const items = document.querySelectorAll(".dcl-carousel-element");
      return Array.from(items)
        .map((item) => {
          const nameElement = item.querySelector(".dcl-product-label");
          const dealPriceElement = item.querySelector(
            ".dcl-product-price-new .a-offscreen"
          );
          const originalPriceElement = item.querySelector(
            ".dcl-product-price-old .a-offscreen"
          );
          const percentOffElement = item.querySelector(
            ".dcl-badge .a-size-mini"
          );

          return {
            name: nameElement ? nameElement.textContent.trim() : null,
            dealPrice: dealPriceElement
              ? dealPriceElement.textContent.trim()
              : null,
            originalPrice: originalPriceElement
              ? originalPriceElement.textContent.trim()
              : null,
            percentOff: percentOffElement
              ? percentOffElement.textContent.trim()
              : null,
          };
        })
        .filter((product) => product.name && product.dealPrice);
    });

    console.log("Found products:", JSON.stringify(products, null, 2));

    await page.close();
    await browser.close();
  } catch (error) {
    console.error(`Encountered an error: ${error}`);
  } finally {
    await client.sessions.stop(session.id);
    console.log("Session stopped:", session.id);
  }
};

main().catch((error) => {
  console.error(`Encountered an error: ${error}`);
});
```

Run the Scraper

To run the Amazon deals scraper:

1. In your terminal, navigate to the project directory
2. Run the script with Node.js:

```
node index.js
```

The script will:

1. Create a new Hyperbrowser session with captcha solving, ad blocking, and anti-tracking enabled
2. Launch a Puppeteer browser and connect it to the session
3. Navigate to the Amazon deals page, solving any CAPTCHAs that are encountered
4. Wait 10 seconds for the page to load its content
5. Scrape the deal data using Puppeteer's `page.evaluate` method
6. Print the scraped products to the console
7. Close the browser and stop the Hyperbrowser session

You should see the scraped products printed in the console, like:

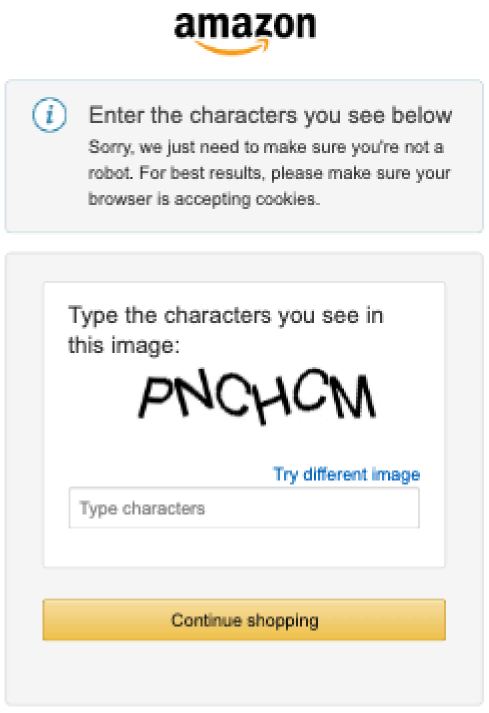
```
[
  {
    "name": "Apple AirPods Pro",
    "dealPrice": "$197.00",
    "originalPrice": "$249.99",
    "percentOff": "21% off"
  },
  {
    "name": "Echo Dot (4th Gen)",
    "dealPrice": "$27.99",
    "originalPrice": "$49.99",
    "percentOff": "44% off"
  }
]
```

How it Works

Let's break down the key parts:

1. We create a new Hyperbrowser session with `solveCaptchas`, `adblock`, `annoyances`, and `trackers` set to `true`. This enables the captcha solver and other anti-bot evasion features.
2. We launch a Puppeteer browser and connect it to the Hyperbrowser session.
3. We navigate to the Amazon deals page and wait for any CAPTCHAs to be solved automatically by Hyperbrowser.
4. We pause execution for 10 seconds with `sleep` to allow all content to be loaded.
5. We use `page.evaluate` to run JavaScript on the page to scrape the deal data.
6. In the evaluator function, we select the deal elements, extract the relevant data, and return an array of product objects.
7. We print the scraped data, close the browser, and stop the Hyperbrowser session.

Without the `solveCaptchas` enabled, we would encounter a screen like this when trying to navigate to the deals page:


Enter the characters you see below

Sorry, we just need to make sure you're not a robot. For best results, please make sure your browser is accepting cookies.

Type the characters you see in this image:

PNCHCM

Try different image

Type characters

Continue shopping

The captcha solver runs automatically in the background, so we don't need to handle captchas explicitly in our script. If a captcha appears, Hyperbrowser will solve it and continue loading the page. In this case, it would solve this CAPTCHA and continue on to the deals page.

reference

SDKs

Hyperbrowser provides SDKs in Node and Python.

Node

Learn about Hyperbrowser's Node SDK

View on Github

Installation

```
npm install @hyperbrowser/sdk
```

or

```
yarn add @hyperbrowser/sdk
```

Usage

```
import { connect } from "puppeteer-core";
import { Hyperbrowser } from "@hyperbrowser/sdk";
import dotenv from "dotenv";

dotenv.config();

const client = new Hyperbrowser({
  apiKey: process.env.HYPERBROWSER_API_KEY,
});

(async () => {
  const session = await client.sessions.create();

  const browser = await connect({
    browserWSEndpoint: session.wsEndpoint,
    defaultViewport: null,
  });

  // Create a new page
  const [page] = await browser.pages();

  // Navigate to a website
  console.log("Navigating to Hacker News...");
  await page.goto("https://news.ycombinator.com/");
  const pageTitle = await page.title();
  console.log("Page title:", pageTitle);

  await page.close();
  await browser.close();
  console.log("Session completed!");
  await client.sessions.stop(session.id);
})().catch((error) => console.error(error.message));
```

Sessions

Create Session

Creates a new browser session with optional configuration.

Method: `client.sessions.create(params?: CreateSessionParams): Promise<SessionDetail>`

Endpoint: `POST /api/session`

Parameters:

- `CreateSessionParams`:
  - `useStealth?: boolean` - Use stealth mode.
  - `useProxy?: boolean` - Use proxy.
  - `proxyServer?: string` - Proxy server URL to route the session through.
  - `proxyServerUsername?: string` - Username for proxy server authentication.
  - `proxyServerPassword?: string` - Password for proxy server authentication.
  - `proxyCountry?:` `Country` - Desired proxy country.
  - proxyState?: `State`- Desired State. Currently only US states are supported. States need to be in two letter codes
  - proxyCity?: string - Desired City. Some cities might not be supported, so before using a new city, we recommend trying it out.
  - `operatingSystems?:` `OperatingSystem` `[]` - Preferred operating systems for the session. Possible values are:
    - `OperatingSystem.WINDOWS`
    - `OperatingSystem.ANDROID`
    - `OperatingSystem.MACOS`
    - `OperatingSystem.LINUX`
    - `OperatingSystem.IOS`
  - `device?: ("desktop" | "mobile")[]` - Preferred device types. Possible values are:
    - `"desktop"`
    - `"mobile"`
  - `platform?:` `Platform` `[]` - Preferred browser platforms. Possible values are:
    - `Platform.CHROME`
    - `Platform.FIREFOX`
    - `Platform.SAFARI`

        - `Platform.EDGE`
    - `locales?:` `ISO639_1` `[]` - Preferred locales (languages) for the session. Use ISO 639-1 codes.
    - `screen?:` `ScreenConfig` - Screen configuration for the session.
        - `width: number` - Screen width.
        - `height: number` - Screen height.
    - `solveCaptchas?: boolean` - Solve captchas.
    - `adblock?: boolean` - Block ads.
    - `trackers?: boolean` - Block trackers.
    - `annoyances?: boolean` - Block annoyances.
    - `enableWebRecording?: boolean` - Default true
    - `extensionIds?: string[]` - Array of extension Ids
    - `acceptCookies?: boolean` - Automatically Accept Cookies on the page
    - `urlBlocklist?: string[]`
    - `browserArgs?: string[]`

Response: `SessionDetail`

Example:

```
const session = await client.sessions.create();
console.log(session.id);
```

Get Session Details

Retrieves details of a specific session.

Method: `client.sessions.get(id: string): Promise<SessionDetail>`

Endpoint: `GET /api/session/{id}`

Parameters:

- `id: string` - Session ID

Response: `SessionDetail`

Example:

```
const session = await client.sessions.get("182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e");
console.log(session.id);
```

List Sessions

Retrieves a list of all sessions with optional filtering.

Method: `client.sessions.list(params?: SessionListParams): Promise<SessionListResponse>`

Endpoint: `GET /api/sessions`

Parameters:

- `SessionListParams`:
  - `status?: "active" | "closed" | "error"` - Filter sessions by status
  - `page?: number` - Page number for pagination

Response: `SessionListResponse`

Example:

```
const response = await client.sessions.list({
  status: "active",
  page: 1,
});
console.log(response.sessions);
```

Stop Session

Stops a running session.

Method: `client.sessions.stop(id: string): Promise<BasicResponse>`

Endpoint: `PUT /api/session/{id}/stop`

Parameters:

- `id: string` - Session ID

Response: `BasicResponse`

Example:

```
const response = await client.sessions.stop(
  "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
);
console.log(`Session stopped: ${response.success}`);
```

Get Session Recording

Get the recording of a session.

Method: `client.sessions.getRecording(id: string): Promise<SessionRecording[]>`

Endpoint: `GET /api/session/{id}/recording`

Parameters:

- `id: string` - Session ID

Response: `SessionRecording` `[]`

Example:

```
const recordingData = await client.sessions.getRecording(
    "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
);
console.log(recordingData);
```

Types

SessionStatus

```
type SessionStatus = "active" | "closed" | "error";
```

Country

```
type Country =
  | "AD"
  | "AE"
  | "AF"
  ...
```

State

Currently only US States are supported. Standard two letter state codes are used.

```
type State =
  | "AL"
  | "AK"
  | "AZ"
  ...
```

OperatingSystem

```
type OperatingSystem = "windows" | "android" | "macos" | "linux" | "ios";
```

Platform

```
type Platform = "chrome" | "firefox" | "safari" | "edge";
```

ISO639_1

```
type ISO639_1 =
  | "aa"
  | "ab"
  | "ae"
  ...
```

BasicResponse

```
interface BasicResponse {
  success: boolean;
}
```

Session

```
interface Session {
  id: string;
  teamId: string;
  status: SessionStatus;
  startTime?: number;
  endTime?: number;
  createdAt: string;
  updatedAt: string;
  sessionUrl: string;
}
```

SessionDetail

```
interface SessionDetail extends Session {
  wsEndpoint?: string;
  liveUrl?: string;
  token?: string;
}
```

SessionListResponse

```
interface SessionListResponse {
  sessions: Session[];
  totalCount: number;
  page: number;
  perPage: number;
}
```

ScreenConfig

```
interface ScreenConfig {
  width: number;
  height: number;
}
```

CreateSessionParams

```
interface CreateSessionParams {
  useStealth?: boolean;
  useProxy?: boolean;
  proxyServer?: string;
  proxyServerPassword?: string;
  proxyServerUsername?: string;
  proxyCountry?: Country;
  operatingSystems?: OperatingSystem[];
  device?: ("desktop" | "mobile")[];
  platform?: Platform[];
  locales?: ISO639_1[];
  screen?: ScreenConfig;
  solveCaptchas?: boolean;
  adblock?: boolean;
  trackers?: boolean;
  annoyances?: boolean;
  enableWebRecording?: boolean;
  extensionIds?: string[];
  acceptCookies?: boolean;
  urlBlocklist?: string[];
  browserArgs?: string[];
}
```

SessionRecording

```
interface SessionRecording {
  type: number;
  data: unknown;
  timestamp: number;
  delay?: number;
}
```

Profiles

Create Profile

Creates a new profile.

Method: `client.profiles.create(): Promise<CreateProfileResponse>`

Endpoint: `POST /api/profile`

Response: `CreateProfileResponse`

Example:

```
const profile = await client.profiles.create();
console.log(profile.id);
```

Get Profile

Get details of an existing profile.

Method: `client.profiles.get(id: string): Promise<ProfileResponse>`

Endpoint: `GET /api/profile/{id}`

Parameters:

- `id: string` - Profile ID

Response: `ProfileResponse`

Example:

```
const profile = await client.profiles.get("36946080-9b81-4288-81d5-b15f2191f222");
console.log(profile.id);
```

Delete Profile

Delete an existing profile.

Method: `client.profiles.delete(id: string): Promise<BasicResponse>`

Endpoint: `DELETE /api/profile/{id}`

Parameters:

- `id: string` - Profile ID

Response: `BasicResponse`

Example:

```
const response = await client.profiles.delete("36946080-9b81-4288-81d5-b15f2191f222");
console.log(response);
```

Types

CreateProfileResponse

```
interface CreateProfileResponse {
  id: string;
}
```

ProfileResponse

```
interface ProfileResponse {
  id: string;
  teamId: string;
  createdAt: string;
  updatedAt: string;
}
```

Scrape

Start Scrape Job

Starts a scrape job for a given URL.

Method: `client.scrape.start(params: StartScrapeJobParams): Promise<StartScrapeJobResponse>`

Endpoint: `POST /api/scrape`

Parameters:

- `StartScrapeJobParams`:
  - `url: string` - URL to scrape
  - `sessionOptions?:` `CreateSessionParams`
  - `scrapeOptions?:` `ScrapeOptions`

Response: `StartScrapeJobResponse`

Example:

```
const response = await client.scrape.start({
  url: "https://example.com",
});
console.log(response.jobId);
```

Get Scrape Job

Retrieves details of a specific scrape job.

Method: `client.scrape.get(id: string): Promise<ScrapeJobResponse>`

Endpoint: `GET /api/scrape/{id}`

Parameters:

- `id: string` - Scrape job ID

Response: `ScrapeJobResponse`

Example:

```
const response = await client.scrape.get(
  "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
);
console.log(response.status);
```

Start Scrape Job and Wait

Start a scrape job and wait for it to complete

Method: `client.scrape.startAndWait(params: StartScrapeJobParams): Promise<ScrapeJobResponse>`

Parameters:

- `StartScrapeJobParams`:
  - `url: string` - URL to scrape
  - `sessionOptions?:` `CreateSessionParams`
  - `scrapeOptions?:` `ScrapeOptions`

Response: `ScrapeJobResponse`

Example:

```
const response = await client.scrape.startAndWait({
  url: "https://example.com"
});
console.log(response.status);
```

Types

ScrapeFormat

```
type ScrapeFormat = "markdown" | "html" | "links" | "screenshot";
```

ScrapeJobStatus

```
type ScrapeJobStatus = "pending" | "running" | "completed" | "failed";
```

ScrapeOptions

```
interface ScrapeOptions {
  formats?: ScrapeFormat[];
  includeTags?: string[];
  excludeTags?: string[];
  onlyMainContent?: boolean;
  waitFor?: number;
  timeout?: number;
}
```

StartScrapeJobResponse

```
interface StartScrapeJobResponse {
  jobId: string;
}
```

ScrapeJobData

```
interface ScrapeJobData {
  metadata?: Record<string, string | string[]>;
  markdown?: string;
  html?: string;
  links?: string[];
}
```

ScrapeJobResponse

```
interface ScrapeJobResponse {
  jobId: string;
  status: ScrapeJobStatus;
  data?: ScrapeJobData;
  error?: string;
}
```

Crawl

Start Crawl Job

Starts a crawl job for a given URL.

Method: `client.crawl.start(params: StartCrawlJobParams): Promise<StartCrawlJobResponse>`

Endpoint: `POST /api/crawl`

Parameters:

- `StartCrawlJobParams`:
  - `url: string` - URL to scrape
  - `maxPages?: number` - Max number of pages to crawl
  - `followLinks?: boolean` - Follow links on the page
  - `ignoreSitemap?: boolean` - Ignore sitemap when finding links to crawl
  - `excludePatterns?: string[]` - Patterns for paths to exclude from crawl
  - `includePatterns?: string[]` - Patterns for paths to include in the crawl
  - `sessionOptions?:` `CreateSessionParams`
  - `scrapeOptions?:` `ScrapeOptions`

Response: `StartCrawlJobResponse`

Example:

```
const response = await client.crawl.start({
  url: "https://example.com",
});
console.log(response.jobId);
```

Get Crawl Job

Retrieves details of a specific crawl job.

Method: `client.crawl.get(id: string): Promise<CrawlJobResponse>`

Endpoint: `GET /api/crawl/{id}`

Parameters:

- `id: string` - Crawl job ID

Response: `CrawlJobResponse`

Example:

```
const response = await client.crawl.get(
  "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
);
console.log(response.status);
```

Start Crawl Job and Wait

Start a crawl job and wait for it to complete

Method: `client.crawl.startAndWait(params: StartCrawlJobParams, returnAllPages: boolean = true): Promise<CrawlJobResponse>`

Parameters:

- `StartCrawlJobParams`:
  - `url: string` - URL to scrape
  - `maxPages?: number` - Max number of pages to crawl
  - `followLinks?: boolean` - Follow links on the page
  - `ignoreSitemap?: boolean` - Ignore sitemap when finding links to crawl
  - `excludePatterns?: string[]` - Patterns for paths to exclude from crawl
  - `includePatterns?: string[]` - Patterns for paths to include in the crawl
  - `sessionOptions?:` `CreateSessionParams`
  - `scrapeOptions?:` `ScrapeOptions`
- `returnAllPages: boolean` - Return all pages in the crawl job response

Response: `CrawlJobResponse`

Example:

```
const response = await client.crawl.startAndWait({
  url: "https://example.com"
});
console.log(response.status);
```

Types

CrawlPageStatus

```
type CrawlPageStatus = "completed" | "failed";
```

CrawlJobStatus

```
type CrawlJobStatus = "pending" | "running" | "completed" | "failed";
```

StartCrawlJobResponse

```
interface StartCrawlJobResponse {
  jobId: string;
}
```

CrawledPage

```
interface CrawledPage {
  url: string;
  status: CrawlPageStatus;
  error?: string | null;
  metadata?: Record<string, string | string[]>;
  markdown?: string;
  html?: string;
  links?: string[];
}
```

CrawlJobResponse

```
interface CrawlJobResponse {
  jobId: string;
  status: CrawlJobStatus;
  data?: CrawledPage[];
  error?: string;
  totalCrawledPages: number;
  totalPageBatches: number;
  currentPageBatch: number;
  batchSize: number;
}
```

Extensions

Add an extension

Adds a new chrome **manifest V3** extension

Method: `client.extensions.create(params:` CreateExtensionParams `):` `Promise<` ExtensionResponse `>`

Endpoint: `POST /api/extensions/add`

Parameters:

- params: CreateExtensionParams
  - filePath: string - Path to the zip containing the manifest V3 compliant extension
  - name?: string - Optional name to give to the extension. Does not affect functionality.

Response: **ExtensionResponse**

Example:

```
const response = await client.extensions.create({
  filePath: "/Users/test-user/Downloads/extension.zip",
});
console.log(response);
```

List all available extension

List all available extensions

Method: `client.extensions.list(` `): Promise<` ` ListExtensionsResponse >`

Endpoint: `POST /api/extensions/list`

Parameters:

- N/A

Response: **ListExtensionResponse**

Example:

```
const extensionsList = await client.extensions.list();
for (let i=0;i<extensionsList.length;i++){
    const extension = extensionsList[i]
    console.log(`extension-${i} => `, JSON.stringify(extension))
}
```

Types

CreateExtensionParams

```
interface CreateExtensionParams {
  filePath: string;
  name?: string;
}
```

ExtensionResponse

```
interface ExtensionResponse {
  name: string;
  id: string;
  createdAt: string;
  updatedAt: string;
}
```

ListExtensionsResponse

```
type ListExtensionsResponse = Array<ExtensionResponse>
```

Python

Learn about Hyperbrowser's Python SDK

View on Github

Installation

```
pip install hyperbrowser
```

Usage

Hyperbrowser's Python SDK includes both a sync and async client.

Async

```
import os
from dotenv import load_dotenv
import asyncio
from pyppeteer import connect
from hyperbrowser import AsyncHyperbrowser

load_dotenv()

client = AsyncHyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))

async def main():
    session = await client.sessions.create()

    ws_endpoint = session.ws_endpoint
    browser = await connect(browserWSEndpoint=ws_endpoint, defaultViewport=None)

    # Get pages
    pages = await browser.pages()
    if not pages:
        raise Exception("No pages available")

    page = pages[0]

    # Navigate to a website
    print("Navigating to Hacker News...")
    await page.goto("https://news.ycombinator.com/")
    page_title = await page.title()
    print("Page title:", page_title)

    await page.close()
    await browser.disconnect()
    await client.sessions.stop(session.id)
    print("Session completed!")

# Run the asyncio event loop
asyncio.get_event_loop().run_until_complete(main())
```

Sync

```
import os
from dotenv import load_dotenv
from playwright.sync_api import sync_playwright
from hyperbrowser import Hyperbrowser

load_dotenv()

client = Hyperbrowser(api_key=os.getenv("HYPERBROWSER_API_KEY"))

def main():
    session = client.sessions.create()

    ws_endpoint = session.ws_endpoint

    # Launch Playwright and connect to the remote browser
    with sync_playwright() as p:
        browser = p.chromium.connect_over_cdp(ws_endpoint)
        context = browser.new_context()

        # Get the first page or create a new one
        if len(context.pages) == 0:
            page = context.new_page()
        else:
            page = context.pages[0]

        # Navigate to a website
        print("Navigating to Hacker News...")
        page.goto("https://news.ycombinator.com/")
        page_title = page.title()
        print("Page title:", page_title)

        page.close()
        browser.close()
        print("Session completed!")
    client.sessions.stop(session.id)

main()
```

Sessions

Create Session

Creates a new browser session with optional configuration.

Method: `client.sessions.create(params?: CreateSessionParams): SessionDetail`

Endpoint: `POST /api/session`

Parameters:

- `CreateSessionParams`:
  - `use_stealth?: boolean` - Use stealth mode.
  - `use_proxy?: boolean` - Use proxy.
  - `proxy_server?: string` - Proxy server URL to route the session through.
  - `proxy_server_username?: string` - Username for proxy server authentication.
  - `proxy_server_password?: string` - Password for proxy server authentication.
  - `proxy_country?:` [`Country`] - Desired proxy country.
  - proxy_state?: [`State`] - Desired State. Currently only US states are supported. States need to be in two letter codes
  - proxy_city?: string - Desired City. Some cities might not be supported, so before using a new city, we recommend trying it out.
  - `operating_systems?:` [`OperatingSystem`] `[]` - Preferred operating systems for the session. Possible values are:
    - `OperatingSystem.WINDOWS`
    - `OperatingSystem.ANDROID`
    - `OperatingSystem.MACOS`
    - `OperatingSystem.LINUX`
    - `OperatingSystem.IOS`
  - `device?: ("desktop" | "mobile")[]` - Preferred device types. Possible values are:
    - `"desktop"`
    - `"mobile"`
  - `platform?:` [`Platform`] `[]` - Preferred browser platforms. Possible values are:
    - `Platform.CHROME`
    - `Platform.FIREFOX`
    - `Platform.SAFARI`

        - `Platform.EDGE`
    - `locales?:` `ISO639_1` `[]` - Preferred locales (languages) for the session. Use ISO 639-1 codes.
    - `screen?:` `ScreenConfig` - Screen configuration for the session.
        - `width: number` - Screen width.
        - `height: number` - Screen height.
    - `solve_captchas?: boolean` - Solve captchas.
    - `adblock?: boolean` - Block ads.
    - `trackers?: boolean` - Block trackers.
    - `annoyances?: boolean` - Block annoyances.
    - `enable_web_recording?: boolean` - Default true
    - `extension_ids?: string[]` - Array of extension Ids
    - `accept_cookies?: boolean` - Automatically Accept Cookies on the page
    - `url_blocklist?: string[]`
    - `browser_args?: string[]`

Response: `SessionDetail`

Example:

```
session = client.sessions.create()
print(session.id)
```

Get Session Details

Retrieves details of a specific session.

Method: `client.sessions.get(id: str): SessionDetail`

Endpoint: `GET /api/session/{id}`

Parameters:

- `id: string` - Session ID

Response: `SessionDetail`

Example:

```
session = client.sessions.get("182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e")
print(session.id)
```

List Sessions

Retrieves a list of all sessions with optional filtering.

Method: `client.sessions.list(params?: SessionListParams): SessionListResponse`

Endpoint: `GET /api/sessions`

Parameters:

- `SessionListParams`:
    - `status?: "active" | "closed" | "error"` - Filter sessions by status
    - `page?: number` - Page number for pagination

Response: `SessionListResponse`

Example:

```
response = client.sessions.list()
print(response.sessions)
```

Stop Session

Stops a running session.

Method: `client.sessions.stop(id: str): BasicResponse`

Endpoint: `PUT /api/session/{id}/stop`

Parameters:

- `id: string` - Session ID

Response: `BasicResponse`

Example:

```
response = client.sessions.stop(
  "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
)
print(f"Session stopped: {response.success}")
```

Get Session Recording

Get the recording of a session.

Method: `client.sessions.get_recording(id: str): SessionRecording[]`

Endpoint: `GET /api/session/{id}/recording`

Parameters:

- `id: string` - Session ID

Response: `SessionRecording` `[]`

Example:

```
recordingData = client.sessions.get_recording(
    "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
);
print(recordingData)
```

Types

SessionStatus

```
SessionStatus = Literal["active", "closed", "error"]
```

Country

```
Country = Literal["AD", "AE", "AF", ...]
```

States

```
State = Literal["AL", "AK", "AZ", ...]
```

OperatingSystem

```
OperatingSystem = Literal["windows", "android", "macos", "linux", "ios"]
```

Platform

```
Platform = Literal["chrome", "firefox", "safari", "edge"]
```

ISO639_1

```
ISO639_1 = Literal["aa", "ab", "ae", ...]
```

BasicResponse

```
class BasicResponse(BaseModel):
    success: bool
```

Session

```
class Session(BaseModel):
    id: str
    team_id: str = Field(alias="teamId")
    status: SessionStatus
    created_at: datetime = Field(alias="createdAt")
    updated_at: datetime = Field(alias="updatedAt")
    start_time: Optional[int] = Field(default=None, alias="startTime")
    end_time: Optional[int] = Field(default=None, alias="endTime")
    duration: Optional[int] = None
    session_url: str = Field(alias="sessionUrl")
```

SessionDetail

```
class SessionDetail(Session):
    ws_endpoint: Optional[str] = Field(alias="wsEndpoint", default=None)
    live_url: str = Field(alias="liveUrl")
    token: str = Field(alias="token")
```

SessionListResponse

```
class SessionListResponse(BaseModel):
    sessions: List[Session]
    total_count: int = Field(alias="totalCount")
    page: int
    per_page: int = Field(alias="perPage")
```

ScreenConfig

```
class ScreenConfig(BaseModel):
    width: int = Field(default=1280, serialization_alias="width")
    height: int = Field(default=720, serialization_alias="height")
```

CreateSessionParams

```
class CreateSessionParams(BaseModel):
    use_stealth: bool = Field(default=False, serialization_alias="useStealth")
    use_proxy: bool = Field(default=False, serialization_alias="useProxy")
    proxy_server: Optional[str] = Field(default=None, serialization_alias="proxyServer")
    proxy_server_password: Optional[str] = Field(
        default=None, serialization_alias="proxyServerPassword"
    )
    proxy_server_username: Optional[str] = Field(
        default=None, serialization_alias="proxyServerUsername"
    )
    proxy_country: Optional[Country] = Field(
        default="US", serialization_alias="proxyCountry"
    )
    operating_systems: Optional[List[OperatingSystem]] = Field(
        default=None, serialization_alias="operatingSystems"
    )
    device: Optional[List[Literal["desktop", "mobile"]]] = Field(default=None)
    platform: Optional[List[Platform]] = Field(default=None)
    locales: List[ISO639_1] = Field(default=["en"])
    screen: Optional[ScreenConfig] = Field(default=None)
    solve_captchas: bool = Field(default=False, serialization_alias="solveCaptchas")
    adblock: bool = Field(default=False, serialization_alias="adblock")
    trackers: bool = Field(default=False, serialization_alias="trackers")
    annoyances: bool = Field(default=False, serialization_alias="annoyances")
    enable_web_recording: Optional[bool] = Field(
        default=True, serialization_alias="enableWebRecording"
    )
    extension_ids: Optional[List[str]] = Field(
        default=None, serialization_alias="extensionIds"
    )
    accept_cookies: Optional[bool] = Field(
        default=None, serialization_alias="acceptCookies"
    )
    url_blocklist: Optional[List[str]] = Field(default=None, serialization_alias="urlBlockli
    browser_args: Optional[List[str]] = Field(default=None, serialization_alias="browserArgs
```

SessionRecording

```
class SessionRecording(BaseModel):
    type: int
    data: Any
    timestamp: int
    delay: Optional[int] = None
```

Profiles

Create Profile

Creates a new profile.

Method: `client.profiles.create(): CreateProfileResponse`

Endpoint: `POST /api/profile`

Response: `CreateProfileResponse`

Example:

```
profile = client.profiles.create()
print(profile.id)
```

Get Profile

Get details of an existing profile.

Method: `client.profiles.get(id: str): ProfileResponse`

Endpoint: `GET /api/profile/{id}`

Parameters:

- `id: string` - Profile ID

Response: `ProfileResponse`

Example:

```
profile = client.profiles.get("36946080-9b81-4288-81d5-b15f2191f222")
print(profile.id)
```

Delete Profile

Delete an existing profile.

Method: `client.profiles.delete(id: str): BasicResponse`

Endpoint: `DELETE /api/profile/{id}`

Parameters:

- `id: string` - Profile ID

Response: `BasicResponse`

Example:

```
response = client.profiles.delete("36946080-9b81-4288-81d5-b15f2191f222")
print(response)
```

Types

CreateProfileResponse

```
class CreateProfileResponse(BaseModel):
    id: str
```

ProfileResponse

```
class ProfileResponse(BaseModel):
    id: str
    team_id: str = Field(alias="teamId")
    created_at: datetime = Field(alias="createdAt")
    updated_at: datetime = Field(alias="updatedAt")
```

Scrape

Start Scrape Job

Starts a scrape job for a given URL.

Method: `client.scrape.start(params: StartScrapeJobParams): StartScrapeJobResponse`

Endpoint: `POST /api/scrape`

Parameters:

- `StartScrapeJobParams`:
  - `url: string` - URL to scrape
  - `session_options?:` `CreateSessionParams`
  - `scrape_options?:` `ScrapeOptions`

Response: `StartScrapeJobResponse`

Example:

```
response = client.scrape.start(StartScrapeJobParams(url="https://example.com"))
print(response.jobId)
```

Get Scrape Job

Retrieves details of a specific scrape job.

Method: `client.scrape.get(id: str): ScrapeJobResponse`

Endpoint: `GET /api/scrape/{id}`

Parameters:

- `id: string` - Scrape job ID

Response: `ScrapeJobResponse`

Example:

```
response = client.scrape.get(
  "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
)
print(response.status)
```

Start Scrape Job and Wait

Start a scrape job and wait for it to complete

Method: `client.scrape.start_and_wait(params: StartScrapeJobParams): ScrapeJobResponse`

Parameters:

- `StartScrapeJobParams`:
  - `url: string` - URL to scrape
  - `session_options?:` `CreateSessionParams`
  - `scrape_options?:` `ScrapeOptions`

Response: `ScrapeJobResponse`

Example:

```
response = client.scrape.start_and_wait(StartScrapeJobParams(url="https://example.com"))
print(response.status)
```

Types

ScrapeFormat

```
ScrapeFormat = Literal["markdown", "html", "links", "screenshot"]
```

ScrapeJobStatus

```
ScrapeJobStatus = Literal["pending", "running", "completed", "failed"]
```

ScrapeOptions

```
class ScrapeOptions(BaseModel):
    formats: Optional[List[ScrapeFormat]] = None
    include_tags: Optional[List[str]] = Field(
        default=None, serialization_alias="includeTags"
    )
    exclude_tags: Optional[List[str]] = Field(
        default=None, serialization_alias="excludeTags"
    )
    only_main_content: Optional[bool] = Field(
        default=None, serialization_alias="onlyMainContent"
    )
    wait_for: Optional[int] = Field(default=None, serialization_alias="waitFor")
    timeout: Optional[int] = Field(default=None, serialization_alias="timeout")
```

StartScrapeJobResponse

```
class StartScrapeJobResponse(BaseModel):
    job_id: str = Field(alias="jobId")
```

ScrapeJobData

```
class ScrapeJobData(BaseModel):
    metadata: Optional[dict[str, Union[str, list[str]]]] = None
    html: Optional[str] = None
    markdown: Optional[str] = None
    links: Optional[List[str]] = None
```

ScrapeJobResponse

```
class ScrapeJobResponse(BaseModel):
    job_id: str = Field(alias="jobId")
    status: ScrapeJobStatus
    error: Optional[str] = None
    data: Optional[ScrapeJobData] = None
```

Crawl

Start Crawl Job

Starts a crawl job for a given URL.

Method: `client.crawl.start(params: StartCrawlJobParams): StartCrawlJobResponse`

Endpoint: `POST /api/crawl`

Parameters:

- `StartCrawlJobParams`:
  - `url: string` - URL to scrape
  - `max_pages?: number` - Max number of pages to crawl
  - `follow_links?: boolean` - Follow links on the page
  - `ignore_sitemap?: boolean` - Ignore sitemap when finding links to crawl
  - `exclude_patterns?: string[]` - Patterns for paths to exclude from crawl
  - `include_patterns?: string[]` - Patterns for paths to include in the crawl
  - `session_options?:` `CreateSessionParams`
  - `scrape_options?:` `ScrapeOptions`

Response: `StartCrawlJobResponse`

Example:

```
response = client.crawl.start(StartCrawlJobParams(url="https://example.com"))
print(response.status)
```

Get Crawl Job

Retrieves details of a specific crawl job.

Method: `client.crawl.get(id: str): CrawlJobResponse`

Endpoint: `GET /api/crawl/{id}`

Parameters:

- `id: string` - Crawl job ID

Response: `CrawlJobResponse`

Example:

```
response = client.crawl.get(
  "182bd5e5-6e1a-4fe4-a799-aa6d9a6ab26e"
)
print(response.status)
```

Start Crawl Job and Wait

Start a crawl job and wait for it to complete

Method: `client.crawl.start_and_wait(params: StartCrawlJobParams): CrawlJobResponse`

Parameters:

- `StartCrawlJobParams`:
  - `url: string` - URL to scrape
  - `max_pages?: number` - Max number of pages to crawl
  - `follow_links?: boolean` - Follow links on the page
  - `ignore_sitemap?: boolean` - Ignore sitemap when finding links to crawl
  - `exclude_patterns?: string[]` - Patterns for paths to exclude from crawl
  - `include_patterns?: string[]` - Patterns for paths to include in the crawl
  - `session_options?:` `CreateSessionParams`
  - `scrape_options?:` `ScrapeOptions`

Response: `CrawlJobResponse`

Example:

```
response = client.crawl.start_and_wait(StartCrawlJobParams(url="https://example.com"))
print(response.status)
```

Types

CrawlPageStatus

```
CrawlPageStatus = Literal["completed", "failed"]
```

CrawlJobStatus

```
CrawlJobStatus = Literal["pending", "running", "completed", "failed"]
```

StartCrawlJobResponse

```
class StartCrawlJobResponse(BaseModel):
    job_id: str = Field(alias="jobId")
```

CrawledPage

```
class CrawledPage(BaseModel):
    metadata: Optional[dict[str, Union[str, list[str]]]] = None
    html: Optional[str] = None
    markdown: Optional[str] = None
    links: Optional[List[str]] = None
    url: str
    status: CrawlPageStatus
    error: Optional[str] = None
```

CrawlJobResponse

```
class CrawlJobResponse(BaseModel):
    job_id: str = Field(alias="jobId")
    status: CrawlJobStatus
    error: Optional[str] = None
    data: List[CrawledPage] = Field(alias="data")
    total_crawled_pages: int = Field(alias="totalCrawledPages")
    total_page_batches: int = Field(alias="totalPageBatches")
    current_page_batch: int = Field(alias="currentPageBatch")
    batch_size: int = Field(alias="batchSize")
```

Extensions

Add an extension

Adds a new chrome **manifest V3** extension

Method: `client.extensions.create(params:` CreateExtensionParams `):` `Promise<` ExtensionResponse `>`

Endpoint: `POST /api/extensions/add`

Parameters:

- params: CreateExtensionParams
    - `file_path: string` - Path to the zip containing the manifest V3 compliant extension
    - `name?: string` - Optional name to give to the extension. Does not affect functionality.

Response: ExtensionResponse

Example:

```
extension = client.extensions.create(
    params=CreateExtensionParams(file_path="/Users/test-user/Downloads/extension.zip")
)
print(extension);
```

List all available extension

List all available extensions

Method: `client.extensions.list(): Promise< ListExtensionsResponse >`

Endpoint: `POST /api/extensions/list`

Parameters:

- N/A

Response: List[ExtensionResponse]

Example:

```
extensionsList = client.extensions.list();
for i in range(len(extensionsList)):
    extension = extensionsList[i]
    print(f"extension-{i} => ", json.dumps(extension))
```

Types

CreateExtensionParams

```
class CreateExtensionParams(BaseModel):
    name: Optional[str] = Field(default=None, serialization_alias="name")
    file_path: str = Field(serialization_alias="filePath")
```

ExtensionResponse

```
class ExtensionResponse(BaseModel):
    id: str = Field(serialization_alias="id")
    name: str = Field(serialization_alias="name")
    created_at: datetime = Field(serialization_alias="createdAt",alias="createdAt")
    updated_at: datetime = Field(serialization_alias="updatedAt",alias="updatedAt")
```

API Reference

Hyperbrowser API endpoints are documented using OpenAPI specification. It offers methods to create new sessions, get session details, stop sessions, and more.

Sessions

Create new session

POST `/api/session` Test it

Authorizations

Body

`useStealth` boolean · default: false

`useProxy` boolean · default: false

`proxyServer` string

`proxyServerPassword` string

`proxyServerUsername` string

`proxyCity` string | nullable

Desired Country. Some cities might not be supported, so before using a new city, we recommend trying it out

Example: `new york`

`solveCaptchas` boolean · default: false

`adblock` boolean · default: false

`trackers` boolean · default: false

`annoyances` boolean · default: false

`enableWebRecording` boolean

`profile` object

+ Show child attributes

`acceptCookies` boolean

`proxyCountry` string · enum · default: US

Options: `US`, `GB`, `CA`

`proxyState` string · enum | nullable

Optional state code for proxies to US states. Takes in two letter state code.

Options: `AL`, `AK`, `AZ`

`extensionIds` string[] · default:

`urlBlocklist` string[] · default:

`browserArgs` string[] · default:

`operatingSystems` string · enum[]

+ Show child attributes

`device` string · enum[]

+ Show child attributes

`platform` string · enum[]

+ Show child attributes

`locales` string · enum[] · default: en

+ Show child attributes

`screen` object

+ Show child attributes

Responses

> 200 Session created

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url 'https://app.hyperbrowser.ai/api/session' \
  --header 'x-api-key: YOUR_API_KEY' \
  --header 'Content-Type: application/json' \
  --data '{
    "useStealth": false,
    "useProxy": false,
    "proxyServer": "text",
    "proxyServerPassword": "text",
    "proxyServerUsername": "text",
    "proxyCity": "new york",
    "solveCaptchas": false,
    "adblock": false,
    "trackers": false,
    "annoyances": false,
    "enableWebRecording": true,
    "profile": {
      "id": "text",
      "persistChanges": true
    },
    "acceptCookies": true,
    "proxyCountry": "US",
    "proxyState": "AL",
    "extensionIds": [],
    "urlBlocklist": [],
    "browserArgs": [],
    "operatingSystems": [
      "windows"
    ],
    "device": [
      "desktop"
    ],
    "platform": [
      "chrome"
    ],
    "locales": [
      "en"
    ],
    "screen": {
      "width": 1280,
      "height": 720
    }
  }'
```

200

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "teamId": "123e4567-e89b-12d3-a456-426614174000",
  "startTime": "text",
  "endTime": "text",
  "createdAt": "text",
  "updatedAt": "text",
  "status": "active",
  "sessionUrl": "text",
  "liveUrl": "text",
  "token": "text",
  "wsEndpoint": "text"
}
```

Session created

Get session by ID

GET /api/session/{id}

Test it

Authorizations

Path parameters

id string required

Responses

200 Session details

404 Session not found

cURL JavaScript Python HTTP

```
curl -L \
  --url 'https://app.hyperbrowser.ai/api/session/{id}' \
  --header 'x-api-key: YOUR_API_KEY'
```

200 404

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "teamId": "123e4567-e89b-12d3-a456-426614174000",
  "startTime": "text",
  "endTime": "text",
  "createdAt": "text",
  "updatedAt": "text",
  "status": "active",
  "sessionUrl": "text",
  "liveUrl": "text",
  "token": "text",
  "wsEndpoint": "text"
}
```

Session details

Stop a session

PUT /api/session/{id}/stop

Test it

Authorizations

Path parameters

id string required

Responses

200 Session stopped successfully

404 Session not found

500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --request PUT \
  --url 'https://app.hyperbrowser.ai/api/session/{id}/stop' \
  --header 'x-api-key: YOUR_API_KEY'
```

200 404 500

```
{
  "success": true
}
```

Session stopped successfully

Get list of sessions

GET /api/sessions

Test it

> Authorizations

Query parameters

page number · default: 1

status string · enum

Options: `active`, `closed`, `error`

Responses

> 200 List of sessions

> 400 Invalid query parameters

> 500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --url 'https://app.hyperbrowser.ai/api/sessions' \
  --header 'x-api-key: YOUR_API_KEY'
```

200 400 500

```
{
  "totalCount": 100,
  "page": 1,
  "pageSize": 10,
  "sessions": [
    {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "teamId": "123e4567-e89b-12d3-a456-426614174000",
      "startTime": "text",
      "endTime": "text",
      "createdAt": "text",
      "updatedAt": "text",
      "status": "active"
    }
  ]
}
```

List of sessions

Scrape

Create new scrape job

POST /api/scrape

Test it

Authorizations

Body

`url` string · min: 1 required

`sessionOptions` object

+ Show child attributes

`scrapeOptions` object

+ Show child attributes

Responses

200 Scrape job created

400 Invalid request parameters

500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url 'https://app.hyperbrowser.ai/api/scrape' \
  --header 'x-api-key: YOUR_API_KEY' \
  --header 'Content-Type: application/json' \
  --data '{
    "url": "text",
    "sessionOptions": {
      "useStealth": false,
      "useProxy": false,
      "proxyServer": "text",
      "proxyServerPassword": "text",
      "proxyServerUsername": "text",
      "proxyCity": "new york",
      "solveCaptchas": false,
      "adblock": false,
      "trackers": false,
      "annoyances": false,
      "enableWebRecording": true,
      "profile": {
        "id": "text",
        "persistChanges": true
      },
      "acceptCookies": true,
      "proxyCountry": "US",
      "proxyState": "AL",
      "extensionIds": [],
      "urlBlocklist": [],
      "browserArgs": [],
      "operatingSystems": [
        "windows"
      ],
      "device": [
        "desktop"
      ],
      "platform": [
        "chrome"
      ],
      "locales": [
        "en"
      ],
      "screen": {
        "width": 1280,
        "height": 720
      }
    },
    "scrapeOptions": {
      "onlyMainContent": true,
      "waitFor": 0,
      "timeout": 30000,
      "waitUntil": "load",
      "includeTags": [
        "text"
      ],
      "excludeTags": [
        "text"
      ],
      "formats": [
        "markdown"
      ],
      "screenshotOptions": {
```

200 400 500

```
{
  "jobId": "text"
}
```

```
    {
  }'
```

Get scrape job status and result

GET /api/scrape/{id}

Test it

Authorizations

Path parameters

id string · uuid required

Responses

200 Scrape job details

404 Job not found

500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --url 'https://app.hyperbrowser.ai/api/scrape/{id}' \
  --header 'x-api-key: YOUR_API_KEY'
```

200 404 500

```json
{
  "jobId": "text",
  "data": {
    "markdown": "text",
    "html": "text",
    "screenshot": "text",
    "links": [
      "text"
    ],
    "metadata": {
      "ANY_ADDITIONAL_PROPERTY": "text"
    }
  },
  "error": "text",
  "status": "pending"
}
```

Scrape job details

Start a batch scrape job

POST `/api/scrape/batch` Test it

> Authorizations

Body

`sessionOptions` object

+ Show child attributes

`scrapeOptions` object

+ Show child attributes

`urls` string[] required

Responses

> 200 Batch scrape job started successfully

> 400 Invalid request parameters

> 402 Insufficient plan

> 429 Too many concurrent batch scrape jobs

> 500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url 'https://app.hyperbrowser.ai/api/scrape/batch' \
  --header 'x-api-key: YOUR_API_KEY' \
  --header 'Content-Type: application/json' \
  --data '{
    "sessionOptions": {
      "useStealth": false,
      "useProxy": false,
      "proxyServer": "text",
      "proxyServerPassword": "text",
      "proxyServerUsername": "text",
      "proxyCity": "new york",
      "solveCaptchas": false,
      "adblock": false,
      "trackers": false,
      "annoyances": false,
      "enableWebRecording": true,
      "profile": {
        "id": "text",
        "persistChanges": true
      },
      "acceptCookies": true,
      "proxyCountry": "US",
      "proxyState": "AL",
      "extensionIds": [],
      "urlBlocklist": [],
      "browserArgs": [],
      "operatingSystems": [
        "windows"
      ],
      "device": [
        "desktop"
      ],
      "platform": [
        "chrome"
      ],
      "locales": [
        "en"
      ],
      "screen": {
        "width": 1280,
        "height": 720
      }
    },
    "scrapeOptions": {
      "onlyMainContent": true,
      "waitFor": 0,
      "timeout": 30000,
      "waitUntil": "load",
      "includeTags": [
        "text"
      ],
      "excludeTags": [
        "text"
      ],
        "formats": [
200  400  "markdown"  500
        ]
 {
   "jobId": "text"
 }
        }
      },
      "urls": [
        "text"
      ]
    }'
```

Get batch scrape job status and results

GET /api/scrape/batch/{id}

Test it

Authorizations

Path parameters

id string required

Responses

200 Batch scrape job details

400 Invalid request parameters

404 Batch scrape job not found

500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --url 'https://app.hyperbrowser.ai/api/scrape/batch/{id}' \
  --header 'x-api-key: YOUR_API_KEY'
```

200 400 404 500

```
{
  "jobId": "text",
  "error": "text",
  "totalScrapedPages": 1,
  "totalPageBatches": 1,
  "currentPageBatch": 1,
  "batchSize": 1,
  "status": "pending",
  "data": [
    {
      "url": "text",
      "error": "text",
      "markdown": "text",
      "html": "text",
      "screenshot": "text",
      "status": "completed",
      "links": [
        "text"
      ],
      "metadata": {
        "ANY_ADDITIONAL_PROPERTY": "text"
      }
    }
  ]
}
```

Batch scrape job details

Crawl

Start a crawl job

POST `/api/crawl` Test it

Authorizations

Body

`url` string required

`maxPages` integer · min: 1

`followLinks` boolean · default: true

`ignoreSitemap` boolean · default: false

`sessionOptions` object

+ Show child attributes

`scrapeOptions` object

+ Show child attributes

`excludePatterns` string[]

`includePatterns` string[]

Responses

200 Crawl job started successfully

400 Invalid request parameters

500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url 'https://app.hyperbrowser.ai/api/crawl' \
  --header 'x-api-key: YOUR_API_KEY' \
  --header 'Content-Type: application/json' \
  --data '{
    "url": "text",
    "maxPages": 1,
    "followLinks": true,
    "ignoreSitemap": false,
    "sessionOptions": {
      "useStealth": false,
      "useProxy": false,
      "proxyServer": "text",
      "proxyServerPassword": "text",
      "proxyServerUsername": "text",
      "proxyCity": "new york",
      "solveCaptchas": false,
      "adblock": false,
      "trackers": false,
      "annoyances": false,
      "enableWebRecording": true,
      "profile": {
        "id": "text",
        "persistChanges": true
      },
      "acceptCookies": true,
      "proxyCountry": "US",
      "proxyState": "AL",
      "extensionIds": [],
      "urlBlocklist": [],
      "browserArgs": [],
      "operatingSystems": [
        "windows"
      ],
      "device": [
        "desktop"
      ],
      "platform": [
        "chrome"
      ],
      "locales": [
        "en"
      ],
      "screen": {
        "width": 1280,
        "height": 720
      }
    },
    "scrapeOptions": {
      "onlyMainContent": true,
      "waitFor": 0,
      "timeout": 30000,
      "waitUntil": "load",
      "includeTags": [
        "text"
      ],
      "excludeTags": [
        "text"
      ],
      "formats": [
        "markdown"
      ],
      "screenshotOptions": {
        "fullPage": false,
        "format": "webp"
      }
    },
    "excludePatterns": [
      "text"
    ],
    "includePatterns": [
      "text"
    ]
  }'
```

200 400 500

```
{
  "jobId": "text"
}
```

Crawl job started successfully

Get crawl job status and results

GET `/api/crawl/{id}` Test it

› Authorizations

Path parameters

id `string` required

Query parameters

page `integer`

batchSize `integer · min: 1`

Responses

> 200 Crawl job details retrieved successfully

> 404 Crawl job not found

> 500 Server error

cURL JavaScript Python HTTP

```
curl -L \
  --url 'https://app.hyperbrowser.ai/api/crawl/{id}' \
  --header 'x-api-key: YOUR_API_KEY'
```

200 404 500

```
{
  "jobId": "123e4567-e89b-12d3-a456-426614174000",
  "error": "text",
  "totalCrawledPages": 1,
  "totalPageBatches": 1,
  "currentPageBatch": 1,
  "batchSize": 1,
  "status": "pending",
  "data": [
    {
      "url": "text",
      "error": "text",
      "markdown": "text",
      "html": "text",
      "screenshot": "text",
      "status": "completed",
      "links": [
        "text"
      ],
      "metadata": {
        "ANY_ADDITIONAL_PROPERTY": "text"
      }
    }
  ]
}
```

Crawl job details retrieved successfully

Profiles

Creates a new profile

POST /api/profile

Test it

> Authorizations

Responses

> 200 Profile created

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url 'https://app.hyperbrowser.ai/api/profile' \
  --header 'x-api-key: YOUR_API_KEY'
```

200

```
{
  "id": "text"
}
```

Profile created

Get profile by ID

GET /api/profile/{id}

Test it

> Authorizations

Path parameters

id string required

Responses

200 Profile details

404 Profile not found

cURL JavaScript Python HTTP

```
curl -L \
  --url 'https://app.hyperbrowser.ai/api/profile/{id}' \
  --header 'x-api-key: YOUR_API_KEY'
```

200 404

```
{
  "id": "text",
  "teamId": "text",
  "createdAt": "text",
  "updatedAt": "text"
}
```

Profile details

Delete profile by ID

`DELETE /api/profile/{id}`

Test it

Authorizations

Path parameters

id `string` required

Responses

200 Profile deleted

cURL JavaScript Python HTTP

```
curl -L \
  --request DELETE \
  --url 'https://app.hyperbrowser.ai/api/profile/{id}' \
  --header 'x-api-key: YOUR_API_KEY'
```

200

```
{
  "success": true
}
```

Profile deleted

Extensions

Add a new extension

POST /api/extensions/add

Test it

Authorizations

Body

file string · binary required

name string

Responses

200 Extension added successfully

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url 'https://app.hyperbrowser.ai/api/extensions/add' \
  --header 'x-api-key: YOUR_API_KEY' \
  --header 'Content-Type: multipart/form-data' \
  --form 'file=text' \
  --form 'name=text'
```

200

```
{
  "id": "text",
  "name": "text",
  "createdAt": "text",
  "updatedAt": "text"
}
```

Extension added successfully

List all extensions

GET /api/extensions/list

Test it

Authorizations

Responses

200 Extension added successfully

cURL JavaScript Python HTTP

```
curl -L \
  --url 'https://app.hyperbrowser.ai/api/extensions/list' \
  --header 'x-api-key: YOUR_API_KEY'
```

200

```
[
  {
    "id": "text",
    "name": "text",
    "createdAt": "text",
    "updatedAt": "text"
  }
]
```

Extension added successfully

Integrations

LangChain

Using Hyperbrowser's Document Loader Integration

Hyperbrowser provides a Document Loader integration with LangChain via the `langchain-hyperbrowser` package. It can be used to load the metadata and contents(in formatted markdown or html) of any site as a LangChain `Document`.

Installation and Setup

To get started with `langchain-hyperbrowser`, you can install the package using pip:

```
pip install langchain-hyperbrowser
```

And you should configure credentials by setting the following environment variables:

`HYPERBROWSER_API_KEY=<your-api-key>`

You can get an API Key easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` or you can pass it via the `api_key` argument in the constructor.

Document Loader

The `HyperbrowserLoader` class in `langchain-hyperbrowser` can easily be used to load content from any single page or multiple pages as well as crawl an entire site. The content can be loaded as markdown or html.

```
from langchain_hyperbrowser import HyperbrowserLoader

loader = HyperbrowserLoader(urls="https://example.com")
docs = loader.load()

print(docs[0])
```

Advanced Usage

You can specify the operation to be performed by the loader. The default operation is `scrape`. For `scrape`, you can provide a single URL or a list of URLs to be scraped. For `crawl`, you can only provide a single URL. The `crawl` operation will crawl the provided page and subpages and return a document for each page.

```
loader = HyperbrowserLoader(
  urls="https://hyperbrowser.ai", api_key="YOUR_API_KEY", operation="crawl"
)
```

Optional params for the loader can also be provided in the `params` argument. For more information on the supported params, you can see the params for scraping or crawling.

```
loader = HyperbrowserLoader(
  urls="https://example.com",
  api_key="YOUR_API_KEY",
  operation="scrape",
  params={"scrape_options": {"include_tags": ["h1", "h2", "p"]}}
)
```

LlamaIndex

Using Hyperbrowser's Web Reader Integration

Installation and Setup

To get started with LlamaIndex and Hyperbrowser, you can install the necessary packages using pip:

```
pip install llama-index-core llama-index-readers-web hyperbrowser
```

And you should configure credentials by setting the following environment variables:

`HYPERBROWSER_API_KEY=<your-api-key>`

You can get an API Key easily from the dashboard. Once you have your API Key, add it to your `.env` file as `HYPERBROWSER_API_KEY` or you can pass it via the `api_key` argument in the `HyperbrowserWebReader` constructor.

Usage

Once you have your API Key and have installed the packages you can load webpages into LlamaIndex using `HyperbrowserWebReader`.

```
from llama_index.readers.web import HyperbrowserWebReader

reader = HyperbrowserWebReader(api_key="your_api_key_here")
```

To load data, you can specify the operation to be performed by the loader. The default operation is `scrape`. For `scrape`, you can provide a single URL or a list of URLs to be scraped. For `crawl`, you can only provide a single URL. The `crawl` operation will crawl the provided page and subpages and return a document for each page. `HyperbrowserWebReader` supports loading and lazy loading data in both sync and async modes.

```
documents = reader.load_data(
    urls=["https://example.com"],
    operation="scrape",
)
```

Optional params for the loader can also be provided in the `params` argument. For more information on the supported params, you can see the params for scraping or crawling.

```
documents = reader.load_data(
    urls=["https://example.com"],
    operation="scrape",
    params={"scrape_options": {"include_tags": ["h1", "h2", "p"]}},
)
```