**Phase 2: Anti-Bot Detection & Stealth - 45 Minutes**

**🛡️ Understanding the Bot Detection War**

**What Are We Fighting Against?**

Modern websites use sophisticated systems to detect and block automated scraping:

* **Cloudflare** - The biggest player, used by millions of sites
* **DataDome** - Advanced behavioral analysis
* **PerimeterX** - Real-time bot detection
* **Akamai** - Enterprise-level protection

**How They Detect Bots**

1. **Browser Fingerprinting** - Analyzing browser characteristics
2. **Behavioral Analysis** - Detecting inhuman patterns
3. **Rate Limiting** - Blocking too many requests
4. **JavaScript Challenges** - Running complex JS tests
5. **CAPTCHA Systems** - Human verification

**Key Insight:** The goal isn't to be completely invisible - it's to look like a normal user browsing the site.

**What is Browser Fingerprinting?**

Every browser has a unique "fingerprint" - a combination of characteristics that identify it:

* User agent string
* Screen resolution
* Installed plugins
* Timezone
* Language settings
* WebGL capabilities

**The User Agent Problem**

python

*# Default Playwright user agent (OBVIOUS BOT):*

"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/119.0.0.0 Safari/537.36"

*# Notice "HeadlessChrome" - dead giveaway!*

**Creating Realistic User Agents**

import random

from playwright.async\_api import async\_playwright

# Real user agents from actual browsers

REAL\_USER\_AGENTS = [

"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",

"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/121.0",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.1 Safari/605.1.15"

]

async def create\_realistic\_browser():

async with async\_playwright() as p:

# Choose random user agent

user\_agent = random.choice(REAL\_USER\_AGENTS)

browser = await p.chromium.launch(

headless=True,

args=[

'--no-sandbox',

'--disable-setuid-sandbox',

'--disable-blink-features=AutomationControlled', # Removes automation flags

'--disable-web-security',

'--disable-features=VizDisplayCompositor'

]

)

# Create context with realistic settings

context = await browser.new\_context(

user\_agent=user\_agent,

viewport={'width': 1920, 'height': 1080}, # Common screen size

extra\_http\_headers={

'Accept-Language': 'en-US,en;q=0.9',

'Accept-Encoding': 'gzip, deflate, br',

'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8',

'Upgrade-Insecure-Requests': '1',

'Sec-Fetch-Dest': 'document',

'Sec-Fetch-Mode': 'navigate',

'Sec-Fetch-Site': 'none'

}

)

return browser, context

| **Flag** | **Why it helps (or what it does)** |
| --- | --- |
| headless=True | Runs without a GUI (faster). ✔️ BUT can be detectable; that’s why you spoof UA & remove flags. |
| --no-sandbox & --disable-setuid-sandbox | Required on some Linux servers; otherwise Chrome won’t start without root. |
| --disable-blink-features=AutomationControlled | Removes the navigator.webdriver = true signal that many bot detectors watch for. |
| --disable-web-security | Lets you bypass some CORS restrictions if needed (optional). |
| --disable-features=VizDisplayCompositor | A minor graphics feature—disabling can slightly reduce fingerprinting entropy. |

| **Header** | **Why it matters** |
| --- | --- |
| Accept-Language | Matches locale implied by UA. |
| Accept-Encoding | Tells server you handle compressed responses—typical of browsers. |
| Accept | Mimics the broad mime‑type list real browsers send. |
| Upgrade-Insecure-Requests: 1 | Present in modern desktop Chrome for top‑level navigation. |
| Sec-Fetch-\* | Newer headers used by browsers for request context; faking them reduces suspicion. |

**Why This Works:**

* Removes "HeadlessChrome" identifier
* Uses real browser user agents
* Sets realistic HTTP headers
* Mimics normal browser behavior

**Advanced Fingerprint Masking**

async def mask\_webdriver\_properties(context):

"""

Override JavaScript properties that reveal automation

"""

await context.add\_init\_script("""

// Remove webdriver property

Object.defineProperty(navigator, 'webdriver', {

get: () => undefined,

});

// Override plugins (empty = suspicious)

Object.defineProperty(navigator, 'plugins', {

get: () => [

{

0: {type: "application/x-google-chrome-pdf", suffixes: "pdf", description: "Portable Document Format"},

description: "Portable Document Format",

filename: "internal-pdf-viewer",

length: 1,

name: "Chrome PDF Plugin"

}

],

});

// Override languages

Object.defineProperty(navigator, 'languages', {

get: () => ['en-US', 'en'],

});

// Override permissions

const originalQuery = window.navigator.permissions.query;

window.navigator.permissions.query = (parameters) => (

parameters.name === 'notifications' ?

Promise.resolve({ state: Notification.permission }) :

originalQuery(parameters)

);

""")

**Document: In-Depth Explanation of WebDriver Masking Script in Playwright**

This document provides a deep dive into the JavaScript code used with Playwright to make the browser environment appear more like a real user’s browser and avoid bot detection. This is especially relevant when websites try to detect automation tools using navigator and other browser-specific objects.

1. Object.defineProperty(navigator, 'webdriver', { get: () => undefined })

**What it does:** Overrides the navigator.webdriver property and forces it to return undefined.

**Why it matters:**

* When running headless automation (e.g., via Chrome), the navigator.webdriver property is set to true by default.
* This is a red flag to most bot detection systems (e.g., Cloudflare, Akamai, DataDome).
* Setting this to undefined hides the automation flag.

**Technical Detail:**

Object.defineProperty(navigator, 'webdriver', {  
 **get**: () **=>** **undefined**,  
});

This overwrites the native getter for navigator.webdriver.

2. Object.defineProperty(navigator, 'plugins', {...})

**What it does:** Mocks a real browser plugin list by overriding the navigator.plugins property.

**Why it matters:**

* A real browser will return a list of plugins (PDF viewer, Flash, etc.).
* A headless browser (especially automated) usually returns an empty array, which looks suspicious.
* This creates a fake but believable structure.

**Technical Detail:**

Object.defineProperty(navigator, 'plugins', {  
 **get**: () **=>** [  
 {  
 0: {type: "application/x-google-chrome-pdf", suffixes: "pdf", description: "Portable Document Format"},  
 description: "Portable Document Format",  
 filename: "internal-pdf-viewer",  
 length: 1,  
 name: "Chrome PDF Plugin"  
 }  
 ]  
});

This mimics the Chrome PDF Plugin commonly found in actual Chrome installs.

3. Object.defineProperty(navigator, 'languages', { get: () => ['en-US', 'en'] })

**What it does:** Overrides the navigator.languages property to return a list of commonly used languages.

**Why it matters:**

* In real browsers, this usually returns ['en-US', 'en'] or something similar.
* In headless environments, this property may be empty or not return a realistic array.
* Bots can be detected when navigator.languages.length === 0.

**Technical Detail:**

Object.defineProperty(navigator, 'languages', {  
 **get**: () **=>** ['en-US', 'en']  
});

This returns a believable language array.

4. navigator.permissions.query Override

**What it does:** Overrides the default permissions.query behavior.

**Why it matters:**

* Sites use navigator.permissions.query({ name: 'notifications' }) to test if you’re using a bot.
* In headless browsers, this may return denied, prompt, or other non-default states.

**Technical Detail:**

**const** originalQuery = window.navigator.permissions.query;  
window.navigator.permissions.query = (parameters) **=>** (  
 parameters.name === 'notifications' ?  
 Promise.resolve({ state: Notification.permission }) :  
 originalQuery(parameters)  
);

This intercepts calls to query permissions for notifications and returns a controlled value, bypassing inconsistent or bot-flagging behavior.

Summary Table

| Property Affected | Purpose | Result |
| --- | --- | --- |
| navigator.webdriver | Hide automation flag | undefined |
| navigator.plugins | Mimic real plugins | Fake Chrome PDF Plugin |
| navigator.languages | Return realistic languages | ['en-US', 'en'] |
| permissions.query | Prevent bot flag on query | Return controlled permission state |

Should You Learn JS to Use This?

Not required — but understanding this doc helps you:

* Customize values for specific scraping targets
* Understand what browser features cause detection
* Debug when sites block your bot

This script is highly effective for minimizing bot detection in Playwright automation. Copy it safely and use it within context.add\_init\_script().

🚦 Identifying When You Need This Masking Script

Before adding any stealth patches, confirm that the target site actually performs JavaScript‐based bot checks.  
Here are the most reliable **signals** and a quick **DevTools checklist** you can run every time you start scraping a new domain.

🔑 Common Red‑Flag Symptoms

| Symptom | What It Usually Means |
| --- | --- |
| Page instantly redirects, stays blank, or collapses in **headless** mode only | The site is inspecting navigator.webdriver, headless flags, or missing browser features. |
| “Access Denied”, Cloudflare challenge, or infinite CAPTCHA | JavaScript fingerprinting scored your session as a bot. |
| Works fine with headless=False but fails with headless=True | Headless fingerprint detected. |
| DevTools Console shows references to bot, antibot.js, cf/challenge etc. | Explicit bot‑detector script is running. |

🧪 Quick Console Test (Run in DevTools)

console.log("webdriver:", navigator.webdriver);  
console.log("languages:", navigator.languages);  
console.log("plugins:", navigator.plugins);  
  
**await** navigator.permissions.query({ name: 'notifications' }).then(console.log);

**Interpretation**

* navigator.webdriver === true ➜ immediate giveaway ⚠️
* navigator.languages.length === 0 ➜ suspicious ⚠️
* navigator.plugins.length === 0 ➜ suspicious ⚠️
* Permissions query returns { state: 'denied' } instantly ➜ often headless.

If you see any of these red flags, inject the masking script **before** navigating to your target pages:

context = **await** browser.new\_context(...)  
**await** mask\_webdriver\_properties(context) *# ➜ adds all patches*  
page = **await** context.new\_page()

Now you can scrape with much lower risk of detection.

📌 Takeaway

1. **Detect manually** – Use the console checklist above to confirm bot checks.
2. **Mask** – Inject the navigator stealth script.
3. **Automate** – Optionally run the same test with page.evaluate() (code below).
4. **Iterate** – If the site still blocks you, add extra stealth (canvas/WebGL patches, IP rotation, human‑like delays).

🤖 Automated Detection Inside Playwright (Headless)

If you don’t want to open DevTools manually, run the same checks in headless mode and print the results to your terminal.

**async** **def** bot\_fingerprint\_report(page):  
 *"""Return a dict of bot‑fingerprint signals (works in headless)"""*  
 js = """  
 /\* ⬇ Wrap in an IIFE so we can return an object literal safely \*/  
 (() => ({  
 webdriver: navigator.webdriver,  
 languages: navigator.languages,  
 pluginsLength: navigator.plugins.length,  
 notificationPerm: (() => {  
 return navigator.permissions  
 .query({ name: 'notifications' })  
 .then(r => r.state);  
 })()  
 }))()  
 """  
 **return** **await** page.evaluate(js)

Typical safe output you want:

{  
 'webdriver': False,  
 'languages': ['en-US'],  
 'pluginsLength': 4,  
 'notificationPerm': 'prompt'  
}

If you see webdriver: true or pluginsLength: 0, inject the stealth script **before** navigation.

That’s all you need to decide **when** and **why** to apply these JavaScript masking techniques.

➕ Additional Bot‑Fingerprint Checks (Advanced)

Below are extra properties advanced bot‑detectors may inspect. Add them to your automated report if you notice blocking even after basic masking.

🧪 Extended Headless‑Check Snippet

**async** **def** extended\_fingerprint\_report(page):  
 js = """  
 (async () => ({  
 webdriver: navigator.webdriver,  
 languages: navigator.languages,  
 pluginsLength: navigator.plugins.length,  
 maxTouchPoints: navigator.maxTouchPoints,  
 hardwareConcurrency: navigator.hardwareConcurrency,  
 deviceMemory: navigator.deviceMemory,  
 webglVendor: (() => {  
 const canvas = document.createElement('canvas');  
 const gl = canvas.getContext('webgl');  
 if (!gl) return 'unavailable';  
 const debugInfo = gl.getExtension('WEBGL\_debug\_renderer\_info');  
 return debugInfo ? gl.getParameter(debugInfo.UNMASKED\_VENDOR\_WEBGL) : 'unknown';  
 })(),  
 webglRenderer: (() => {  
 const canvas = document.createElement('canvas');  
 const gl = canvas.getContext('webgl');  
 if (!gl) return 'unavailable';  
 const debugInfo = gl.getExtension('WEBGL\_debug\_renderer\_info');  
 return debugInfo ? gl.getParameter(debugInfo.UNMASKED\_RENDERER\_WEBGL) : 'unknown';  
 })(),  
 canvasFingerprint: (() => {  
 const canvas = document.createElement('canvas');  
 const ctx = canvas.getContext('2d');  
 ctx.textBaseline = 'top';  
 ctx.font = '14px Arial';  
 ctx.fillText('bot‑test', 2, 2);  
 return canvas.toDataURL(); // returns data URI; hash or length can be inspected  
 })(),  
 notificationPerm: await navigator.permissions.query({ name: 'notifications' }).then(r => r.state)  
 }))()  
 """  
 **return** **await** page.evaluate(js)

🚦 How to Interpret the Extended Report

| Field | Real‑Browser Range | Red‑Flag Value |
| --- | --- | --- |
| maxTouchPoints | 0–2 on desktop |  |
| 5+ on mobile | 0 when UA claims mobile |  |
| hardwareConcurrency | 2–16 | 0 or 1 |
| deviceMemory | 1–16 (GB) | undefined or 0 |
| webglVendor/Renderer | nVIDIA, AMD, Intel | SwiftShader, Google (software) |
| canvasFingerprint | Random per device | Identical for many sessions (bot net) |

**Action:** If any red‑flag appears, you may need additional JS patches (canvas spoofing, WebGL spoofing) or switch to headless=False.

You now have both **basic** and **advanced** fingerprint tests and know exactly what to patch when a site blocks your scraper.

# Behavioral Mimicking & Timing

# The Human Pattern Problem

Bots are fast and predictable. Humans are slow and random. Here's how to act human:

## Random Delays Between Actions

import random

import asyncio

async def human\_delay(min\_ms=500, max\_ms=2000):

"""

Random delay that mimics human thinking time

"""

delay = random.uniform(min\_ms, max\_ms) / 1000

await asyncio.sleep(delay)

async def human\_type(page, selector, text):

"""

Types text like a human - with pauses and occasional mistakes

"""

await page.click(selector)

await human\_delay(200, 500) # Think before typing

for char in text:

await page.keyboard.type(char)

# Random typing speed

await asyncio.sleep(random.uniform(0.05, 0.2))

# Sometimes humans pause mid-typing

if random.random() < 0.3:

await human\_delay(500, 1500)

**Mouse Movement Patterns**

async def human\_scroll(page, distance=None):

"""

Scrolls page like a human - in chunks with pauses

"""

if distance is None:

distance = random.randint(500, 1500)

# Scroll in chunks

chunk\_size = random.randint(100, 300)

scrolled = 0

while scrolled < distance:

await page.mouse.wheel(0, chunk\_size)

scrolled += chunk\_size

await asyncio.sleep(random.uniform(0.1, 0.3))

# Pause after scrolling

await human\_delay(1000, 3000)

async def human\_click(page, selector):

"""

Clicks element like a human - with hover and delay

"""

# First hover over element

await page.hover(selector)

await human\_delay(200, 800)

# Then click

await page.click(selector)

await human\_delay(500, 1500)

**Realistic Browsing Patterns**

async def browse\_like\_human(page, url):

"""

Visits page with human-like behavior

"""

# Navigate to page

await page.goto(url)

# Wait for page load (humans don't instantly interact)

await human\_delay(2000, 5000)

# Random scrolling (humans explore pages)

for \_ in range(random.randint(2, 5)):

await human\_scroll(page)

# Sometimes go back up

if random.random() < 0.3:

await page.mouse.wheel(0, -random.randint(200, 800))

await human\_delay(1000, 3000)

# Random mouse movements

for \_ in range(random.randint(1, 3)):

x = random.randint(100, 1000)

y = random.randint(100, 700)

await page.mouse.move(x, y)

await human\_delay(500, 1500)

**Why This Works:**

* Introduces realistic delays between actions
* Mimics human exploration behavior
* Varies timing to avoid detection patterns
* Adds "unnecessary" actions humans do

### Session Consistency

async def maintain\_session(context, base\_url):

"""

Maintains realistic session by visiting related pages

"""

page = await context.new\_page()

# Start with homepage (like humans do)

await page.goto(base\_url)

await human\_delay(3000, 7000)

# Visit a few random pages first

common\_pages = ['/about', '/contact', '/help', '/faq']

for page\_url in random.sample(common\_pages, 2):

try:

await page.goto(f"{base\_url}{page\_url}")

await human\_delay(2000, 5000)

await human\_scroll(page)

except:

pass # Page might not exist

return page

**Request Optimization & Resource Management**

**The Resource Loading Problem**

Normal browsers load images, CSS, fonts, ads - everything. Bots often block these to save bandwidth, which is suspicious.

**Smart Resource Blocking**

async def setup\_stealth\_context(p):

"""

Creates context that blocks suspicious resources but keeps important ones

"""

browser = await p.chromium.launch(

headless=True,

args=[

'--no-sandbox',

'--disable-setuid-sandbox',

'--disable-blink-features=AutomationControlled',

'--disable-web-security',

'--disable-features=VizDisplayCompositor',

'--disable-background-timer-throttling',

'--disable-backgrounding-occluded-windows',

'--disable-renderer-backgrounding'

]

)

context = await browser.new\_context(

user\_agent=random.choice(REAL\_USER\_AGENTS),

viewport={'width': 1920, 'height': 1080},

extra\_http\_headers={

'Accept-Language': 'en-US,en;q=0.9',

'Accept-Encoding': 'gzip, deflate, br',

'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8',

}

)

# Block heavy resources but keep important ones

await context.route("\*\*/\*", lambda route: (

route.abort() if route.request.resource\_type in ["image", "media", "font"] and

not any(keyword in route.request.url for keyword in ["logo", "icon", "avatar"])

else route.continue\_()

))

return browser, context

**Persisting Sessions Using storage\_state (Cookies & LocalStorage)**

**1  Overview**

storage\_state is Playwright’s built‑in mechanism to **export** and **import** all session‑related data (cookies, localStorage, sessionStorage). By saving this state after you finish a login flow, you can launch future browser contexts that already contain the saved credentials—just like a returning user who never has to log in again.

**2  Why Persist Sessions?**

| **Benefit** | **Details** |
| --- | --- |
| **Skip repetitive logins** | Speeds up scraping/testing; reduces CAPTCHA and 2FA prompts. |
| **Mimic real user behaviour** | Sites see a consistent set of cookies & local data across visits. |
| **Reduce bot‑detection friction** | Returning sessions look less suspicious than brand‑new ones. |
| **Share auth between scripts** | One script logs in, others reuse the saved state. |

**3  Saving Session State**

After you authenticate (or set preferences) in a context or page:

# After successful login

await context.storage\_state(path="auth\_state.json")

* Creates **auth\_state.json** containing cookies + storage for every domain touched in that context.
* Store multiple profiles by saving to different files (e.g., user1\_state.json, user2\_state.json).

**Sample JSON Snippet**

{

"cookies": [

{"name": "sessionid", "value": "ABCD123", "domain": ".example.com", ... }

],

"origins": [

{"origin": "https://example.com", "localStorage": [{"name": "theme", "value": "dark"}]}

]

}

**4  Reusing Session State**

Pass the saved file when creating a **new** context:

context = await browser.new\_context(

storage\_state="auth\_state.json",

user\_agent=random.choice(REAL\_USER\_AGENTS),

viewport={"width": 1920, "height": 1080}

)

page = await context.new\_page()

await page.goto("https://example.com/dashboard") # Already logged in!

* You can still add other context options (UA, viewport, proxy, etc.).
* If cookies expire, simply rerun the login flow and overwrite the JSON.

**5  Practical Tips & Common Pitfalls**

* **Set cookies before navigation** when injecting manually; otherwise some sites reset them.
* **Domain must match** — a cookie for .example.com won’t apply to sub.example.com unless the domain is set correctly.
* **Secure flag** — if the site is HTTPS‑only, make sure cookies are saved with "secure": true.
* **Avoid committing auth JSON files to version control** — treat them like passwords.
* **Update when cookies expire** — many session cookies have short TTLs.

**6  End‑to‑End Example Workflow**

async def login\_and\_save(playwright):

browser = await playwright.chromium.launch()

context = await browser.new\_context()

page = await context.new\_page()

# ---- 1. Perform login ----

await page.goto("https://example.com/login")

await page.fill("#user", "alice")

await page.fill("#pass", "s3cr3t")

await page.click("button:has-text('Sign in')")

await page.wait\_for\_url("\*\*/dashboard")

# ---- 2. Save session ----

await context.storage\_state(path="auth\_state.json")

await browser.close()

async def reuse\_session(playwright):

browser = await playwright.chromium.launch()

context = await browser.new\_context(storage\_state="auth\_state.json")

page = await context.new\_page()

# Directly navigate to a protected page

await page.goto("https://example.com/dashboard")

# ... continue actions ...

**When to Refresh the Session File**

| **Symptom** | **Likely Cause** | **Fix** |
| --- | --- | --- |
| Redirected to login page | Session cookie expired | Rerun login flow and overwrite auth\_state.json. |
| 401 Unauthorized API | CSRF token rotated | Save new tokens in a fresh storage state. |
| Site shows CAPTCHA | Bot‑detection cookies missing | Ensure they are included in the saved file, or harvest them again. |

**7  Resources**

* **Playwright Docs – storage\_state**: <https://playwright.dev/python/docs/auth>
* **Playwright Github Examples**: look for auth.json saved states.

**Next step** ➜ Combine storage\_state with your **stealth context** (UA spoofing, resource blocking) for a fully automated yet human‑like browsing bot.

**Proxy Rotation Strategy**

class ProxyRotator:

def \_\_init\_\_(self, proxy\_list):

self.proxies = proxy\_list

self.current\_index = 0

def get\_next\_proxy(self):

proxy = self.proxies[self.current\_index]

self.current\_index = (self.current\_index + 1) % len(self.proxies)

return proxy

async def create\_proxy\_context(p, proxy\_rotator):

"""

Creates browser context with rotating proxy

"""

proxy = proxy\_rotator.get\_next\_proxy()

browser = await p.chromium.launch(headless=True)

context = await browser.new\_context(

proxy={

"server": f"http://{proxy['ip']}:{proxy['port']}",

"username": proxy.get('username'),

"password": proxy.get('password')

},

user\_agent=random.choice(REAL\_USER\_AGENTS)

)

return browser, context

**🔁 Example with 3 Proxies:**

self.proxies = [proxy1, proxy2, proxy3]

| **Call #** | **self.current\_index** | **(index + 1) % 3** | **Resulting Index** |
| --- | --- | --- | --- |
| **1** | **0** | **(0+1)%3 = 1** | **1** |
| **2** | **1** | **(1+1)%3 = 2** | **2** |
| **3** | **2** | **(2+1)%3 = 0 ✅** | **0 (loops back)** |
| **4** | **0** | **(0+1)%3 = 1** | **1** |

**Complete Stealth Scraper Example**

**import asyncio**

**import random**

**from playwright.async\_api import async\_playwright**

**from datetime import datetime**

**class StealthScraper:**

**def \_\_init\_\_(self):**

**self.user\_agents = [**

**"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",**

**"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",**

**"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/121.0"**

**]**

**async def create\_stealth\_browser(self):**

**"""Creates a browser that looks like a real user"""**

**async with async\_playwright() as p:**

**browser = await p.chromium.launch(**

**headless=True,**

**args=[**

**'--no-sandbox',**

**'--disable-setuid-sandbox',**

**'--disable-blink-features=AutomationControlled',**

**'--disable-web-security',**

**'--disable-features=VizDisplayCompositor'**

**]**

**)**

**context = await browser.new\_context(**

**user\_agent=random.choice(self.user\_agents),**

**viewport={'width': 1920, 'height': 1080},**

**extra\_http\_headers={**

**'Accept-Language': 'en-US,en;q=0.9',**

**'Accept-Encoding': 'gzip, deflate, br',**

**}**

**)**

**# Override automation detection**

**await context.add\_init\_script("""**

**Object.defineProperty(navigator, 'webdriver', {**

**get: () => undefined,**

**});**

**Object.defineProperty(navigator, 'plugins', {**

**get: () => [1, 2, 3, 4, 5],**

**});**

**""")**

**return browser, context**

**async def human\_delay(self, min\_ms=500, max\_ms=2000):**

**"""Random delay to mimic human behavior"""**

**delay = random.uniform(min\_ms, max\_ms) / 1000**

**await asyncio.sleep(delay)**

**async def scrape\_with\_stealth(self, url):**

**"""Main scraping function with stealth features"""**

**async with async\_playwright() as p:**

**browser, context = await self.create\_stealth\_browser()**

**page = await context.new\_page()**

**try:**

**# Navigate like a human**

**await page.goto(url, wait\_until='networkidle')**

**await self.human\_delay(2000, 5000)**

**# Random scrolling**

**for \_ in range(random.randint(1, 3)):**

**await page.mouse.wheel(0, random.randint(200, 800))**

**await self.human\_delay(500, 1500)**

**# Extract data**

**data = await self.extract\_data(page)**

**# More human behavior**

**await self.human\_delay(1000, 3000)**

**return data**

**except Exception as e:**

**print(f"Error scraping {url}: {e}")**

**return None**

**finally:**

**await browser.close()**

**async def extract\_data(self, page):**

**"""Extract data with error handling"""**

**try:**

**# Wait for content to load**

**await page.wait\_for\_selector('h1', timeout=10000)**

**return {**

**'title': await page.text\_content('h1') or 'N/A',**

**'price': await page.text\_content('.price, .a-price-whole') or 'N/A',**

**'rating': await page.text\_content('.rating, .a-icon-alt') or 'N/A',**

**'scraped\_at': datetime.now().isoformat()**

**}**

**except Exception as e:**

**print(f"Error extracting data: {e}")**

**return {}**

**# Usage example**

**async def main():**

**scraper = StealthScraper()**

**url = "https://example-store.com/product/123"**

**data = await scraper.scrape\_with\_stealth(url)**

**print(data)**

**asyncio.run(main())**

### 🎯 Testing Your Stealth Setup

### Bot Detection Tests

async def test\_bot\_detection(page):

"""Test if your setup is detected as a bot"""

test\_sites = [

"https://bot.sannysoft.com/", # Comprehensive bot detection

"https://pixelscan.net/", # Fingerprint analysis

"https://www.whatismybrowser.com/" # Browser info

]

for site in test\_sites:

print(f"\nTesting: {site}")

try:

await page.goto(site)

await page.wait\_for\_load\_state('networkidle')

await page.screenshot(path=f'test\_{site.split("/")[2]}.png')

print("✅ Passed - No blocks detected")

except Exception as e:

print(f"❌ Failed - {e}")

**✅ Summary Comparison**

| **Site** | **Focus** | **What You Want** |
| --- | --- | --- |
| **bot.sannysoft.com** | **JS detection & fingerprint** | **All green flags, no webdriver, no empty plugins** |
| **pixelscan.net** | **Full fingerprint + IP/proxy** | **IP matches proxy, no headless/bot flags** |
| **whatismybrowser.com** | **UA + basic fingerprint** | **Real user-agent, no "headless"** |

**Success Metrics**

A good stealth setup should:

* **Pass bot detection tests** (bot.sannysoft.com shows green)
* **Load pages consistently** (no 403/429 errors)
* **Mimic human timing** (2-10 seconds between actions)
* **Vary behavior** (different patterns each run)

## 🚨 Common Stealth Mistakes

### 1. **Too Fast**

# BAD - Superhuman speed

await page.click('button')

await page.fill('input', 'text')

await page.click('submit')

# GOOD - Human-like timing

await page.click('button')

await human\_delay(500, 1500)

await page.fill('input', 'text')

await human\_delay(1000, 2000)

await page.click('submit')

**2. Predictable Patterns**

# BAD - Same delay every time

await asyncio.sleep(1)

# GOOD - Random delays

await asyncio.sleep(random.uniform(0.5, 2.0))

**3. Ignoring Resource Loading**

# BAD - Blocks everything (suspicious)

await context.route("\*\*/\*", lambda route: route.abort())

# GOOD - Selective blocking

await context.route("\*\*/\*", lambda route: (

route.abort() if route.request.resource\_type in ["image", "media"]

else route.continue\_()

))

**In‑Depth Guide: Selective Request Blocking in Playwright for Stealth Scraping**

**1. Why Request Interception Matters**

Modern sites employ multilayered bot‑detection stacks that analyse both **network fingerprints** (headers, timing, resource mix) and **behavioural signals** (mouse, scroll, typing). Intercepting and aborting requests lets you:

* Save bandwidth and speed up crawls.
* Hide tell‑tale signs like ad‑tech beacons.
* Reduce JavaScript execution that might detect automation.

But done incorrectly, interception itself becomes a **giant red flag**.

**2. Crude vs Selective Blocking**

**❌ Crude: Block Everything**

await context.route("\*\*/\*", lambda r: r.abort())

**Side‑effects**

1. **Missing assets** – CSS/JS/images never load; the DOM often stays empty.
2. **Abnormal timing** – Human sessions fetch dozens of resources; you fetch zero in ≈0 ms.
3. **Server‑side errors** – Logs show a burst of aborted TCP connections.

Result: immediate flagging by ML models trained on normal traffic.

**✅ Selective: Block Only Non‑Critical Resources**

await context.route(

"\*\*/\*", # intercept every URL once

lambda route: (

route.abort() if route.request.resource\_type in {"image", "media"}

else route.continue\_()

)

)

**Benefits**

* Loads HTML, CSS, JS, XHR – realistic request mix.
* Skips heavy images/media – bandwidth ↓, speed ↑.
* Retains believable timing + sequence patterns.

**3. Anatomy of a Playwright Request**

Key attributes you can inspect inside a route handler:

| **Property** | **Example** | **Use‑case** |
| --- | --- | --- |
| url | https://site.com/img/logo.png | pattern/regex blocking |
| resource\_type | "document", "image", "media", "script", "xhr", "font" | quick category filtering |
| method | "GET", "POST" | abort tracking POSTs |
| headers | dict | strip/override headers |

**4. Step‑by‑Step Walk‑Through of the GOOD Pattern**

1. **Intercept** every outgoing request with the glob "\*\*/\*".
2. **Categorise** using route.request.resource\_type (constant‑time lookup, no regex).
3. **Abort** only if it belongs to the block‑list {"image", "media"}.
4. **Continue** everything else unchanged (route.continue\_()).

**Tip:** Keep the abort‑list small; add more types only after profiling.

**5. Advanced Refinements**

| **Technique** | **Code Snippet** | **Why It Helps** |  |
| --- | --- | --- | --- |
| **Regex domain filters** | `if re.search(r"analytics | doubleclick", req.url):` | Drop tracking scripts without killing core JS |
| **Hero‑image allowance** | Load the first image, then block rest: maintain layout screenshotability |  |  |
| **Random abort probability** | if random.random() < 0.8: | Imitate flaky human network conditions |  |
| ``\*\* placeholders\*\* | Serve 1‑byte transparent PNG instead of aborting | Keeps 200 OK status, avoids 4xx noise |  |
| **Adaptive throttling** | Sleep random.randint(300,1200) ms after each abort | Smooths request cadence |  |

**6. Re‑usable Helper Module**

import random, re

from typing import Iterable

BLOCK\_TYPES = {"image", "media"}

TRACKING\_RE = re.compile(r"analytics|doubleclick|gtm")

async def add\_smart\_blocking(context, allow\_first\_image: bool = True):

seen\_image = False

async def \_handler(route):

nonlocal seen\_image

req = route.request

# block noisy domains

if TRACKING\_RE.search(req.url):

return await route.abort()

# selectively block images/media

if req.resource\_type in BLOCK\_TYPES:

if allow\_first\_image and not seen\_image:

seen\_image = True

else:

# 80% chance to abort, 20% chance to continue for realism

if random.random() < 0.8:

return await route.abort()

await route.continue\_()

await context.route("\*\*/\*", \_handler)

Copy‑paste the helper, call it right after browser.new\_context().

**7. Integrating With Proxy Rotation & Human‐Like Behaviour**

* **Proxy first**, then **context**, then **route** – order matters.
* Combine with:
  + Random user‑agent + viewport.
  + Mouse moves (page.mouse.move) and scroll dribbles.
  + Delay between page navigation and data extraction.

**8. Measuring Your Stealth**

| **Metric** | **Target Range** |
| --- | --- |
| Avg. resources / page | 30‑90 (varies by site) |
| First contentful paint | 1–4 s (avoid <500 ms) |
| Image abort ratio | 70‑90 % |
| Server 4xx/5xx logs | Near zero |

Use **DevTools > Network** or proxy tools like **mitmproxy** to audit.

**9. Future‑Proofing Against Evolving ML Models**

1. Rotate **TLS fingerprints** (JA3) using patched Chromium builds.
2. Periodically capture **real human traffic** baseline to compare.
3. Stay up‑to‑date on stealth plugins (playwright‑stealth, puppeteer‑extra‑plugin‑stealth).
4. Introduce **task‑level randomisation**: different navigator languages, time‑zones, cookie jars.

**10. Key Takeaways**

* Blocking **everything** = instant detection & broken pages.
* Selective blocking of *non‑critical* resources strikes the right balance.
* Add random‑ness, domain filters, and placeholder fulfilment for extra realism.
* Measure, iterate, and keep logs — stealth is an *ongoing* process, not a one‑time tweak.

**Need more?**

Ping me for:

* A ready‑to‑run Playwright starter repo.
* Proxy validation + rotation snippets.
* Complete stealth automation template (incl. captcha solving & metrics).

## 🎯 Advanced Stealth Techniques

### 1. **Canvas Fingerprinting Protection**

await context.add\_init\_script("""

const getImageData = HTMLCanvasElement.prototype.getImageData;

HTMLCanvasElement.prototype.getImageData = function(sx, sy, sw, sh) {

const imageData = getImageData.apply(this, arguments);

for (let i = 0; i < imageData.data.length; i += 4) {

imageData.data[i] += Math.floor(Math.random() \* 10) - 5;

imageData.data[i + 1] += Math.floor(Math.random() \* 10) - 5;

imageData.data[i + 2] += Math.floor(Math.random() \* 10) - 5;

}

return imageData;

};

""")

**Bot Detection Signals and Anti-Bot Evasion Techniques**

This document compiles in-depth details about anti-bot detection signals, how websites detect bots, and how to evade those detections using tools like Playwright. It includes practical examples and concepts discussed around proxy rotation, canvas fingerprinting, and more.

**1. Bot Detection Signals**

Websites employ multiple signals to determine whether the traffic is generated by a bot. These include:

**1.1 Behavioral Signals**

* **Mouse Movements**: Bots typically lack real human-like mouse trails.
* **Keyboard Input**: Simulated typing vs real-time input.
* **Scrolling & Click Patterns**: Humans scroll in patterns; bots often jump or use fixed intervals.

**1.2 Fingerprinting Techniques**

* **Canvas Fingerprinting**: Websites draw an image using <canvas> and analyze its pixel data.
* **Audio Fingerprinting**: Using the Web Audio API to generate and fingerprint audio output.
* **WebGL Fingerprinting**: Measures GPU and rendering differences.
* **Fonts & Plugins Enumeration**: Abnormalities in available fonts or plugin list.
* **Timezone, Language, and Screen Resolution**: Inconsistent settings raise suspicion.

**1.3 Network & Request Signals**

* **Headers (User-Agent, Accept-Language)**: Missing or generic headers.
* **IP Address (Proxy/VPN)**: Blacklisted or datacenter IPs.
* **TLS Fingerprinting**: Browser TLS handshake anomalies.
* **Request Patterns**: Too many rapid requests from the same IP or account.

**2. Proxy Rotation System**

A proxy rotation system helps avoid detection from IP-based restrictions. Here's a sample class:

class ProxyRotator:

def \_\_init\_\_(self, proxy\_list):

self.proxies = proxy\_list

self.current\_index = 0

def get\_next\_proxy(self):

proxy = self.proxies[self.current\_index]

self.current\_index = (self.current\_index + 1) % len(self.proxies)

return proxy

**Example Context with Proxy and Real User-Agent**

async def create\_proxy\_context(p, proxy\_rotator):

proxy = proxy\_rotator.get\_next\_proxy()

browser = await p.chromium.launch(headless=True)

context = await browser.new\_context(

proxy={

"server": f"http://{proxy['ip']}:{proxy['port']}",

"username": proxy.get('username'),

"password": proxy.get('password')

},

user\_agent=random.choice(REAL\_USER\_AGENTS)

)

return browser, context

**3. Request Blocking vs Filtering**

**❌ BAD - Block All Requests**

await context.route("\*\*/\*", lambda route: route.abort())

This blocks everything and appears suspicious.

**✅ GOOD - Block Only Unnecessary Resources**

await context.route("\*\*/\*", lambda route: (

route.abort() if route.request.resource\_type in ["image", "media"]

else route.continue\_()

))

**4. Canvas Fingerprint Evasion**

Websites use canvas fingerprinting to identify bots. You can intercept it using JavaScript.

**Example Script to Tamper Canvas Data:**

await context.add\_init\_script("""

const getImageData = HTMLCanvasElement.prototype.getImageData;

HTMLCanvasElement.prototype.getImageData = function(sx, sy, sw, sh) {

const imageData = getImageData.apply(this, arguments);

for (let i = 0; i < imageData.data.length; i += 4) {

imageData.data[i] += Math.floor(Math.random() \* 10) - 5;

imageData.data[i + 1] += Math.floor(Math.random() \* 10) - 5;

imageData.data[i + 2] += Math.floor(Math.random() \* 10) - 5;

}

return imageData;

};

""")

This makes each canvas fingerprint slightly different to prevent consistent tracking.

**What is Canvas Image?**

The website draws an invisible image (e.g., text or shape) on an HTML <canvas> before the page fully loads. Then it extracts the pixel data (ImageData), hashes it, and uses it to fingerprint your system.

**5. How to Detect If a Website Uses Bot Detection?**

**5.1 Developer Tools**

Use browser developer tools (F12):

* Check network requests for scripts with names like bot-detection.js, cf-challenge, fingerprint.js, etc.
* Look at XHR/WebSocket requests.
* Monitor console for errors or strange behaviors (e.g., blocked by Cloudflare).

**5.2 Third-Party Tools**

* **Puppeteer Detector**: Used to test how detectable your browser environment is.
* **CreepJS / FingerprintJS**: Open-source or demo versions show how a site may track you.

**5.3 Behavior-Based Responses**

* Are you getting CAPTCHA immediately?
* Are some features hidden or broken unless you use a real browser?
* Does the page redirect, return 403/503, or load slowly for no reason?

**6. Do Free Proxies Work?**

* Sometimes, but they are often unreliable.
* Many are already flagged on IP blacklists.
* Paid rotating residential proxies are more effective but costly.

**7. Mindset for Scraping and Freelancing**

* Scraping is **not just about code** — it's about **thinking like a human**.
* Fiverr/Upwork clients are often **non-technical**, so highlight your ability to:
  + Avoid detection
  + Gather specific niche-based data
  + Clean, analyze, and visualize the results

Combining scraping with **ML/Data Analysis** makes you a **premium-level freelancer** over time.

**8. Final Advice**

* Think in terms of human behavior while designing scrapers.
* Never over-request from one IP/device.
* Mimic browser environments.
* Build multiple fallback mechanisms.

Let me know if you want this formatted as a downloadable PDF or need a version focused on Fiverr/Upwork gigs.

**2. WebGL Fingerprinting Protection**

await context.add\_init\_script("""

const getParameter = WebGLRenderingContext.prototype.getParameter;

WebGLRenderingContext.prototype.getParameter = function(parameter) {

if (parameter === 37445) {

return 'Intel Inc.';

}

if (parameter === 37446) {

return 'Intel(R) Iris(TM) Graphics 6100';

}

return getParameter.apply(this, arguments);

};

""")

| **Human-readable name** | **Internal number** |
| --- | --- |
| gl.UNMASKED\_VENDOR\_WEBGL | 37445 |
| gl.UNMASKED\_RENDERER\_WEBGL | 37446 |
| gl.VERSION | 7938 |
| gl.VENDOR | 7936 |

**What Data Does Your Browser Share with Websites?**

When you visit a website, your browser automatically shares a variety of information. This data helps websites function properly, personalize content, and sometimes track users. Here's a breakdown:

**1. Basic Info (Always Sent)**

| **What** | **Example** |
| --- | --- |
| IP Address | 123.45.67.89 (can reveal your rough location) |
| User-Agent | Mozilla/5.0 (Windows NT 10.0; Win64; x64)...Chrome/... |
| Referrer | URL of the page you came from |
| Language | en-US, ml-IN, etc. |
| Browser Type | Chrome, Firefox, Safari |
| Operating System | Windows, macOS, Linux, Android |

**2. Device and Hardware Info (Used for Fingerprinting)**

| **What** | **Example** |
| --- | --- |
| Screen Resolution | 1920x1080 |
| Timezone | Asia/Kolkata |
| Installed Fonts | List of system fonts |
| GPU Info (WebGL) | NVIDIA GTX 1050, Intel Iris |
| CPU Cores | 8 |
| Battery Level | 60% (if accessed with permission) |
| Audio Fingerprint | Subtle data from sound rendering |
| Touch Support | Touch screen present or not |

**3. Browser Features and Settings**

| **What** | **Example** |
| --- | --- |
| Plugins | PDF Viewer, Flash, etc. |
| Cookies Enabled? | Yes/No |
| LocalStorage | Browser-stored site data |
| IndexedDB | Structured data storage |
| Do Not Track | Enabled or Disabled |
| Canvas Fingerprint | Unique image from invisible drawing |
| WebGL Fingerprint | Unique GPU rendering info |

**4. Custom and Behavioral Data**

| **What** | **Example** |
| --- | --- |
| Login Status | Logged into Google, Facebook, etc. |
| Mouse Movements | Human-like or robotic? |
| Typing Speed | Realistic or too fast? |
| Scroll Patterns | Human-style scrolling? |

**5. During Automation (e.g., Playwright, Puppeteer)**

| **What** | **How It's Detected** |
| --- | --- |
| Headless Mode | Hidden browser flags |
| Modified WebGL | Missing or spoofed values |
| Timing Differences | Scripts load too perfectly |
| Missing Extensions | Real users usually have some |

Understanding what your browser shares helps in both privacy protection and automation stealth. You can use various tools and techniques to mask or minimize this data.

**✅ Phase 2 Mastery Checklist**

You've mastered anti-bot detection when you can:

* Explain how bot detection works
* Create realistic browser fingerprints
* Implement human-like timing patterns
* Use proper resource blocking strategies
* Pass bot detection tests
* Handle rate limiting gracefully
* Rotate user agents and proxies effectively

**Success Test:** Your scraper can consistently access protected sites like Amazon product pages without getting blocked.

**🎯 Ready for Phase 3?**

You're ready for **Phase 3: Data Pipeline & Automation** when you can:

* Scrape protected sites without detection
* Implement realistic human behavior
* Handle different types of bot protection
* Debug stealth issues effectively

**Next Up:** We'll build a complete data pipeline with scheduling, error recovery, and professional data handling!

*💡 Pro Tip: Stealth is about being believable, not perfect. A few human-like "mistakes" actually make you less suspicious.*