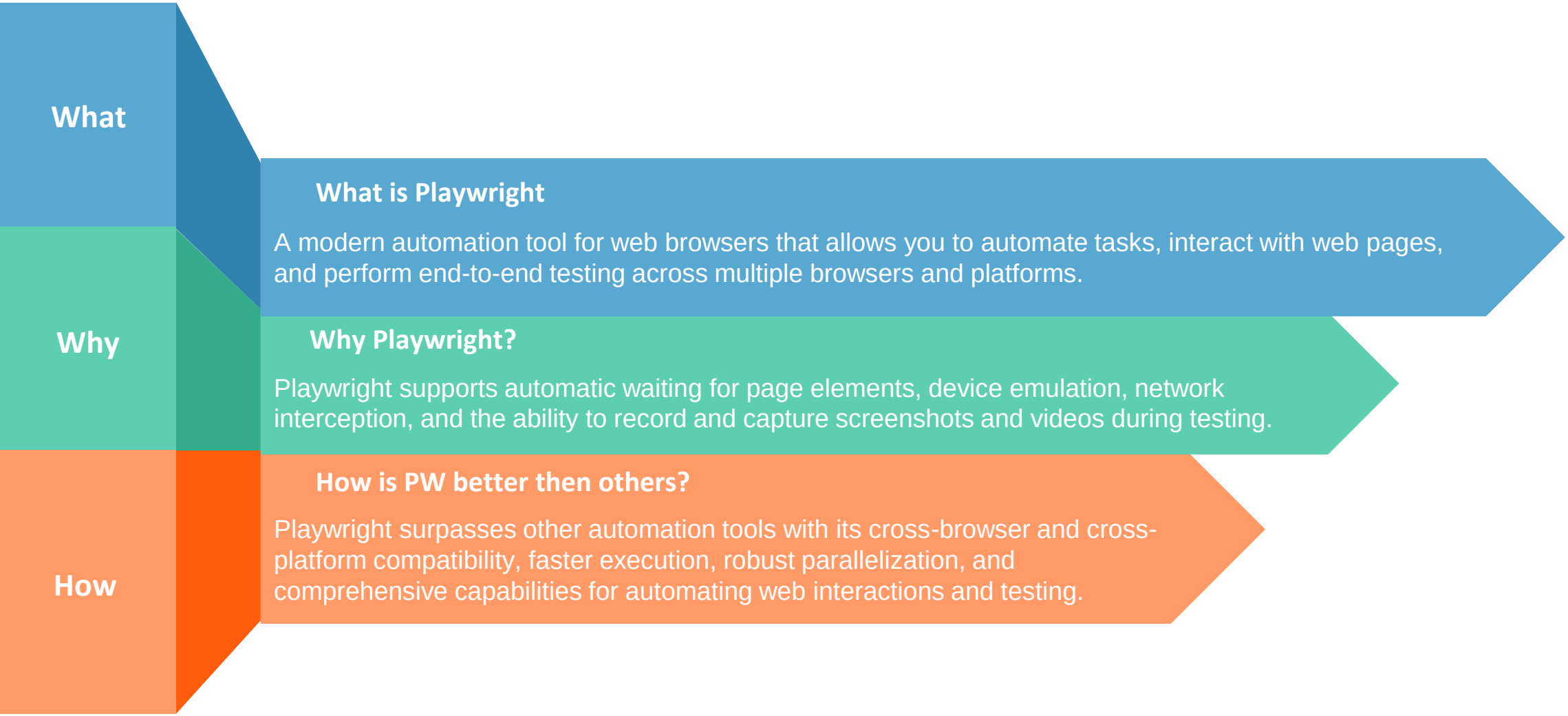
Let’s get into action with Playwright

Testleaf
Always Ahead

Agenda – 1 (Basics)

- Introduction to Playwright
- Selenium Architecture vs Playwright
- Playwright documentation

The Golden Circle



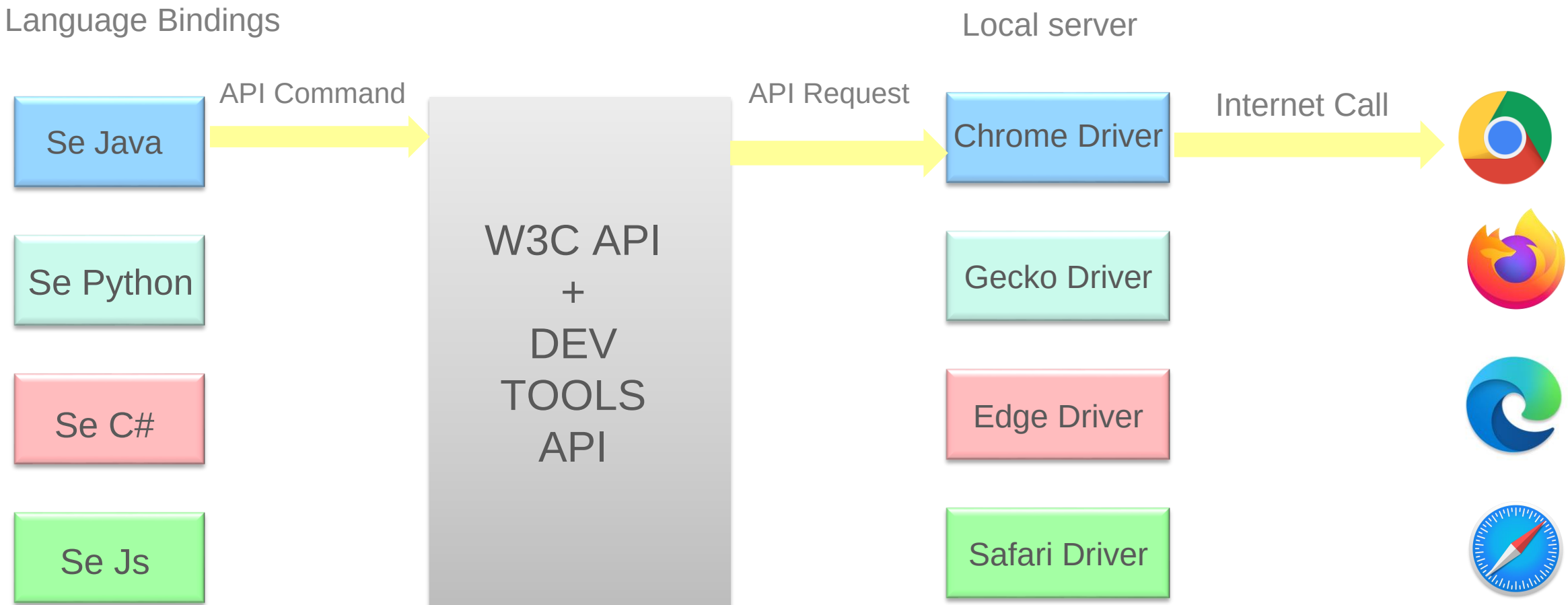
Testleaf
Always Ahead

Intro to Playwright

- A modern, open source web test framework from Microsoft
- Manipulates the browser via (superfast) debug protocols
- Works with Chromium/Chrome/Edge, Firefox, & WebKit
- Provides automatic waiting, test generation, UI mode, etc.
- Can test UIS and APIs together
- Bindings for JavaScript, Python, Java, & C#
  - TypeScript is recommended

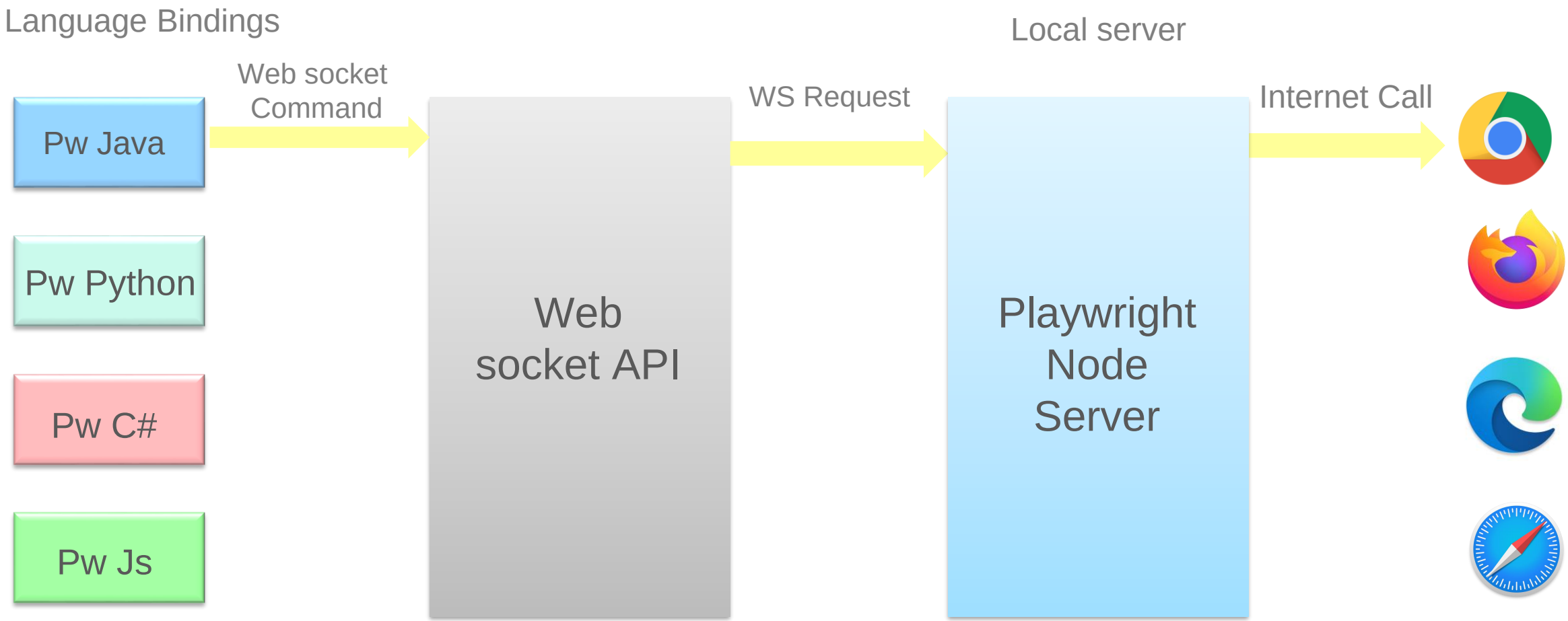
Testleaf
Always Ahead

Selenium Architecture



Testleaf
Always Ahead

Playwright Architecture



Understanding basic elements of browser

Browser Process

- *Main application process responsible for UI, handles renders and other processes*

Render Process

- *Each tab in browser has its own render process*
- *Responsible for displaying content of web page*

Network process

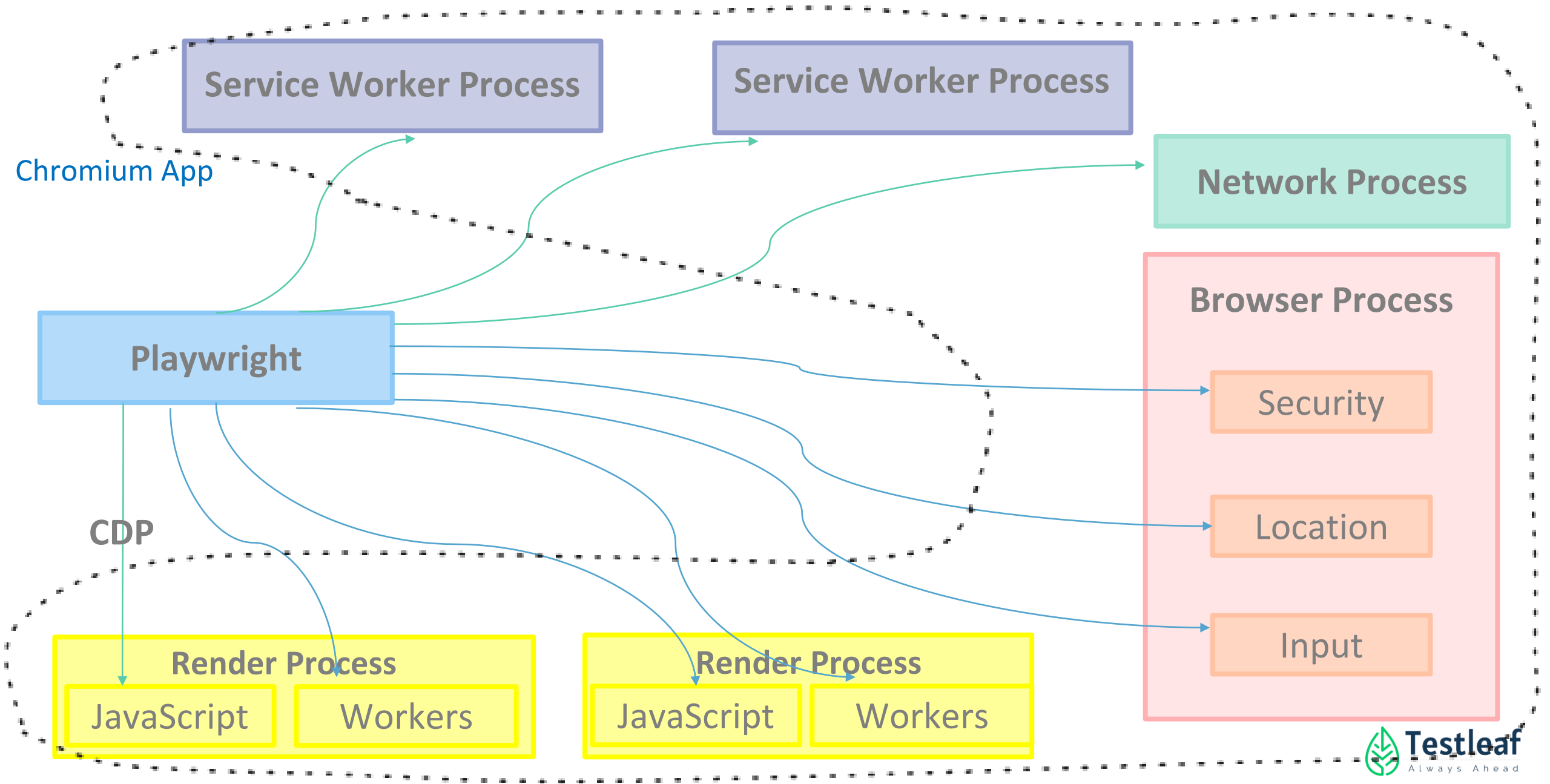
- *Handles all network related tasks, fetching resources like css, js etc , making http requests etc and caching*

Service worker process

- *Background script separate from main browser thread, they can intercept and handle network requests and they ensure that they do not block main process. -* ***Used heavily by extensions***

Testleaf
Always Ahead

What Playwright gets because of CDP



Testleaf
Always Ahead

Playwright Documentations

Playwright Docs API 1.22 Node.js Search ⌘ K

Playwright enables reliable end-to-end testing for modern web apps.

GET STARTED Star 39k+

Any browser • Any platform • One API

Link to doc: https://playwright.dev/

Check!

What protocol does Playwright employ to establish communication with web browsers?

- HTTP
- WebSocket
- SMTP
- FTP

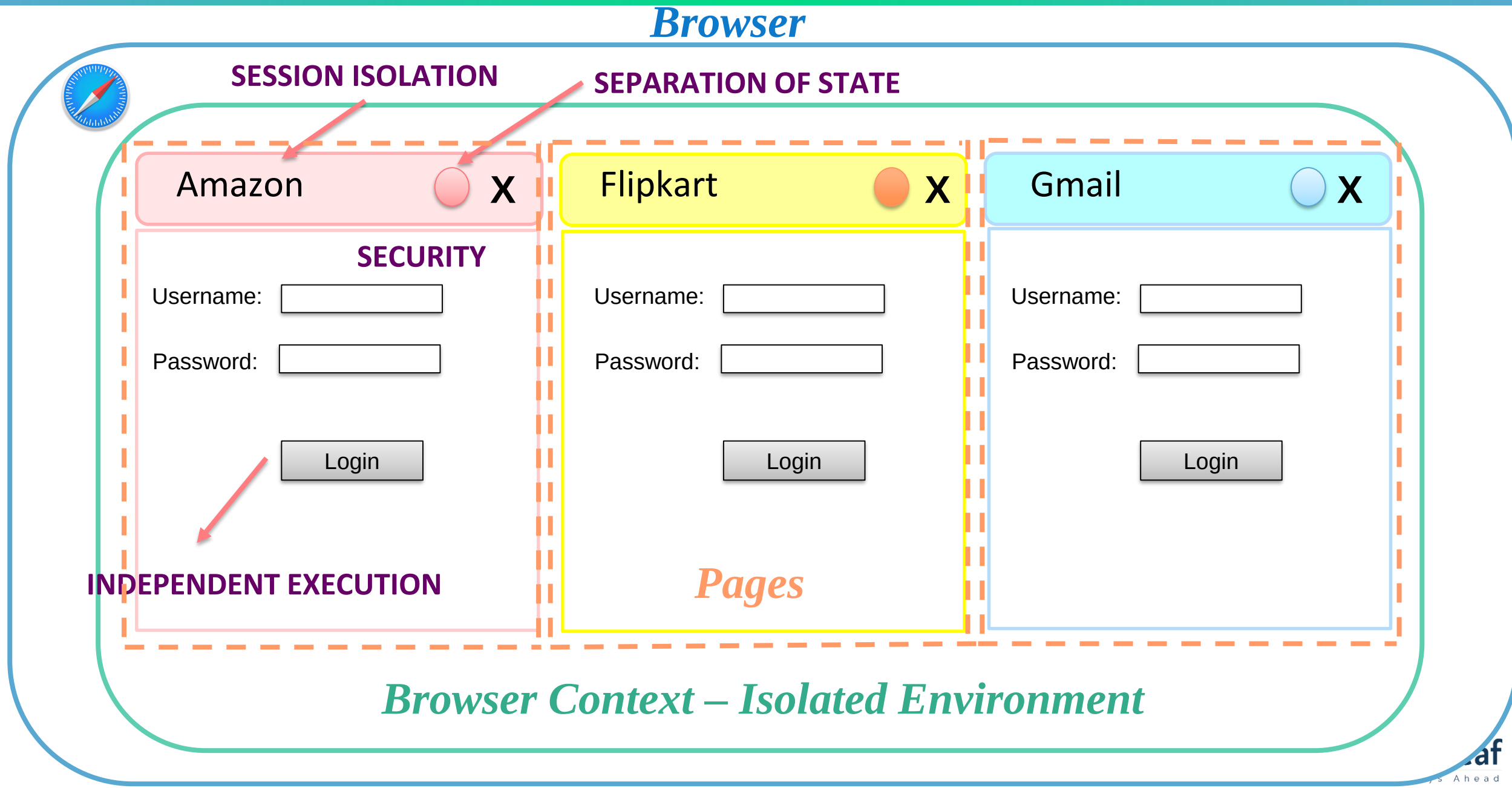
Testleaf
Always Ahead

Agenda – Part 2 (Playwright Key Concepts)

- Browser
- BrowserContext
- Page
- DOM
- WebElement

Testleaf
Always Ahead

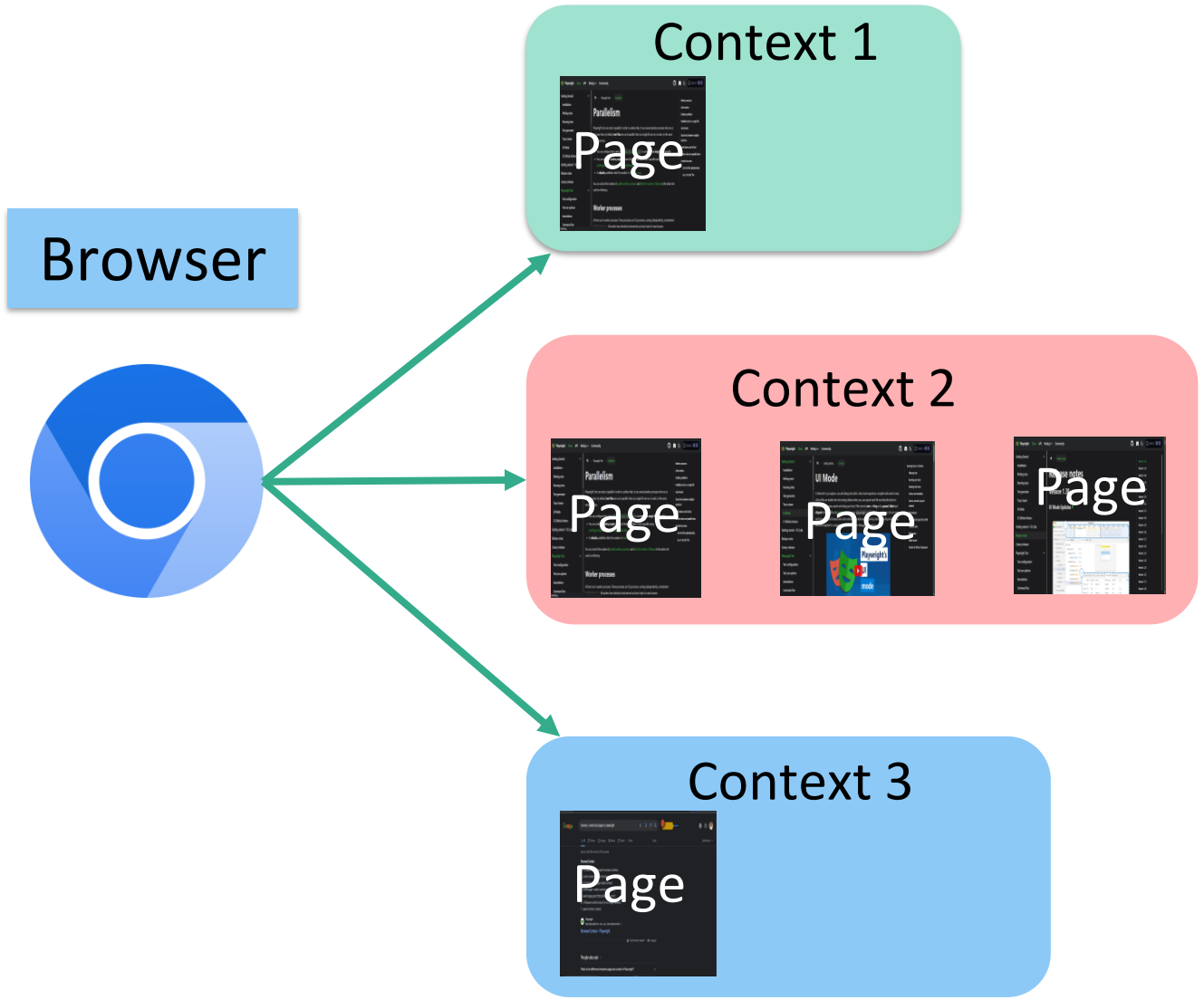
Browser, Browser Context & Page



Browser & Browser Context

Context

- Creates a unique *browser context* from that instance for each test.
- A browser context is essentially like an incognito session: it has its own session storage and tabs that are not shared with any other context.
- Browser contexts are very fast to create and destroy

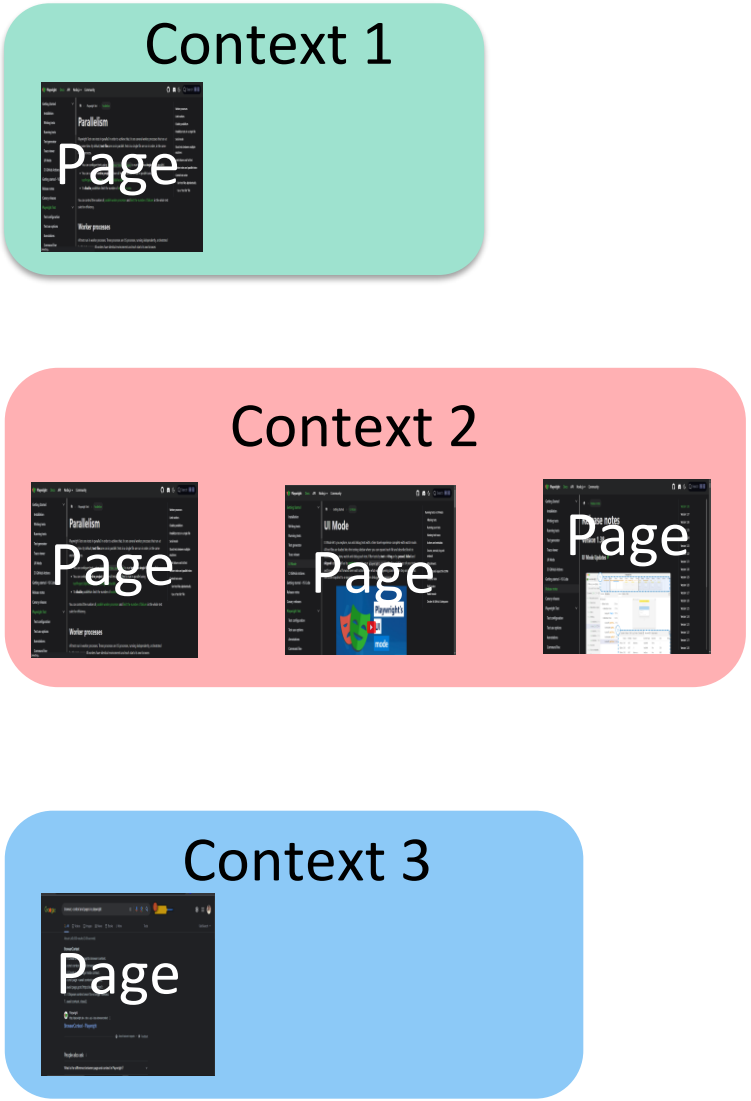


Testleaf
Always Ahead

Browser Context & Pages

Page

- Each browser context can have one or more *pages*.
- All Playwright interactions happen through a page, like clicks and scrapes.
- Most tests only ever need one page.



Testleaf
Always Ahead

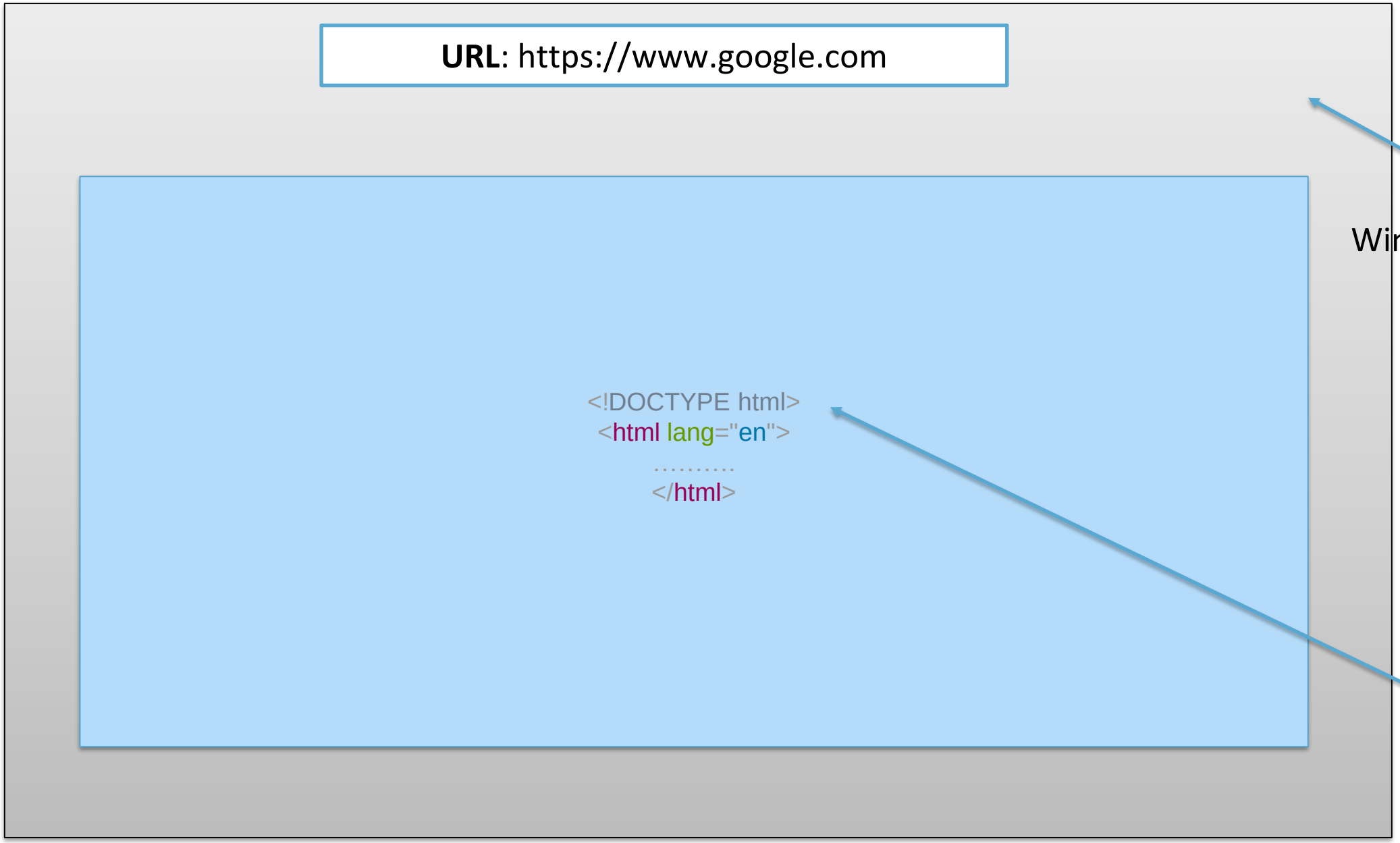
Check!

What is the correct hierarchical order when structuring your automation workflow in Playwright?

- BrowserContext, Browser instance, Page
- Page, Browser instance, BrowserContext
- Browser instance, BrowserContext, Page
- Browser instance, Page, BrowserContext

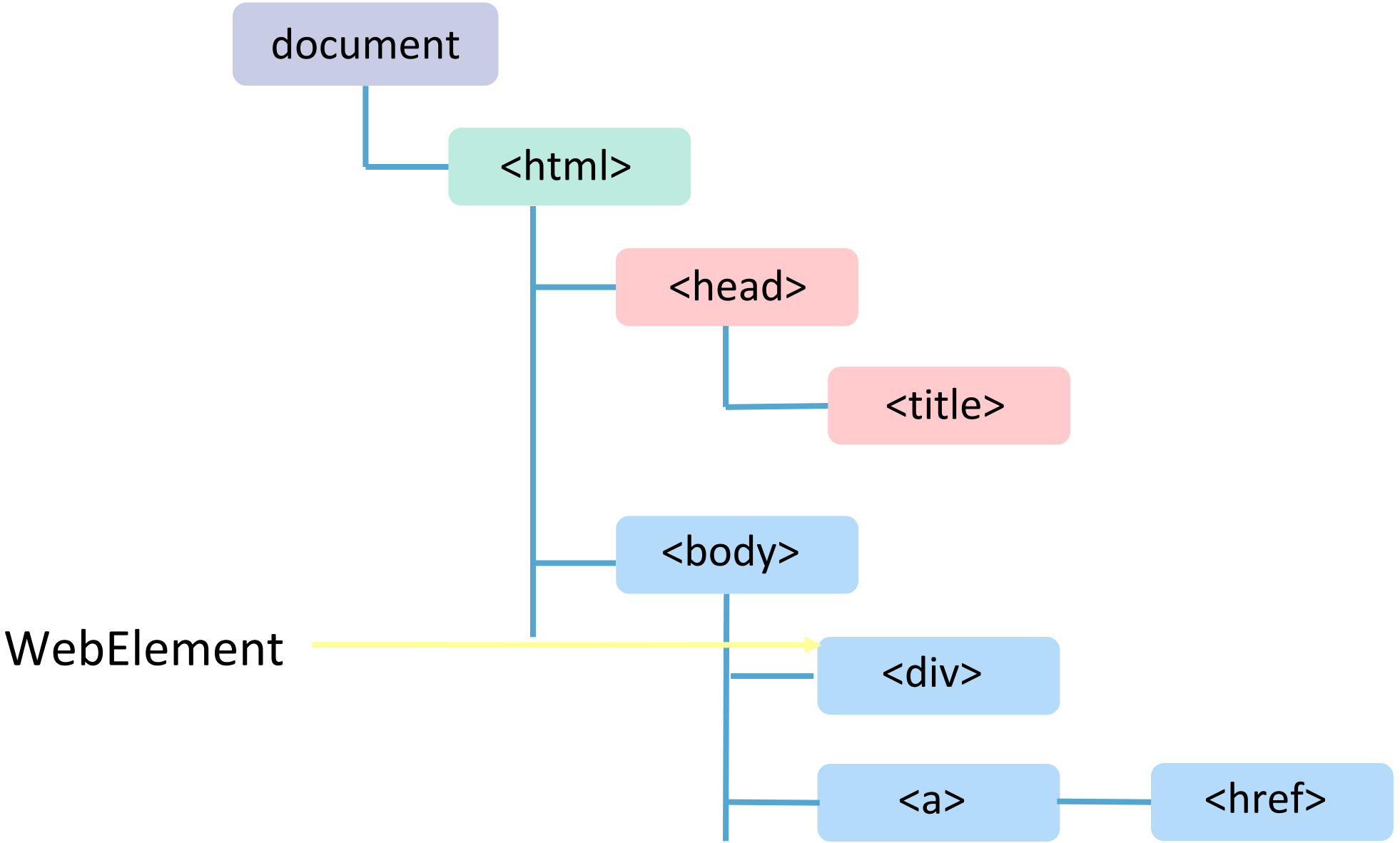
Testleaf
Always Ahead

DOM - Document object model



Testleaf
Always Ahead

DOM & WebElement



Testleaf
Always Ahead

Classwork - Let’s write our first test

- Create a new browser instance
- Create a new context
- Create a new page
- Open a url
- Wait for 10 seconds
- Print the current url

Testleaf
Always Ahead

Playwright vs Selenium

Playwright

- Create a new browser instance, context and page

```
const  browserInstance = await chromium.launch()
const  browserContext = await browserInstance.
                        newContext()
const  page = await browserContext.newPage()
```

- Open url

```
await page.goto("url")
```

- Wait for 10 seconds

```
await page.waitForTimeout(1000)
```

- Print the url

```
const currentUrl = page.url()
console.log(currentUrl)
```

Selenium

- Initialize driver

```
WebDriver driver = new ChromeDriver();
```

- Open url

```
driver.get("url");
```

- Wait for 10 seconds

```
Thread.sleep(1000);
```

- Print the url

```
String currentUrl = driver.getCurrentUrl();
System.out.println(currentUrl);
```

Testleaf
Always Ahead

Agenda – Part 3

- Locators
- Selectors
- Diving into Selector Strategies
- Which strategy to prefer?
- Industry best practices

Testleaf
Always Ahead

Locators & Selectors

```
        else{
            awa
            awa
            awa
            con
            exp
            con
        }
    }
})

test('Locators'
    await page.locator('#user-name')
});
```

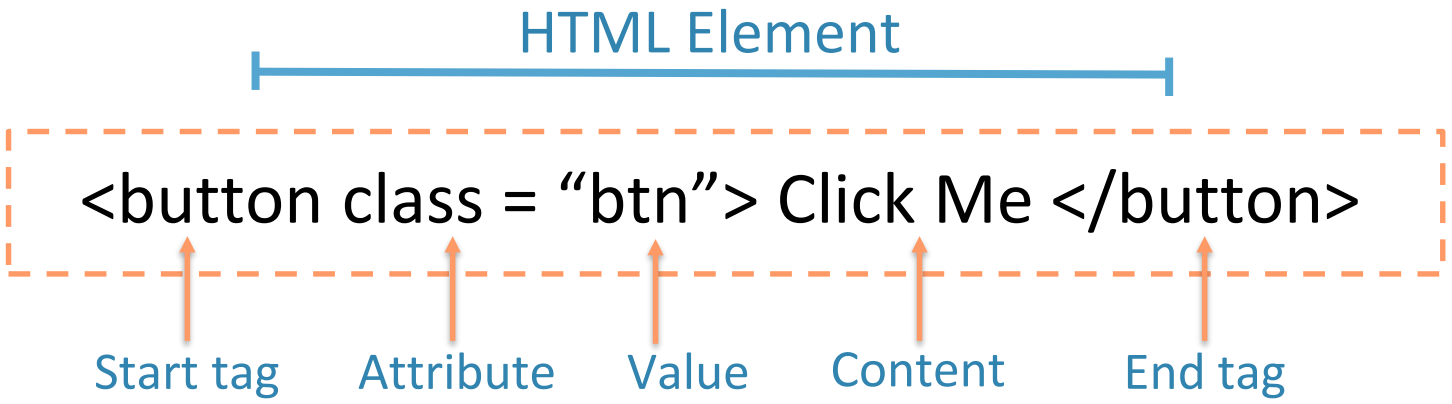
```
(method) Page.locator(selector: string, options?: {
    has?: Locator | undefined;
    hasText?: string | RegExp | undefined;
} | undefined): Locator
```

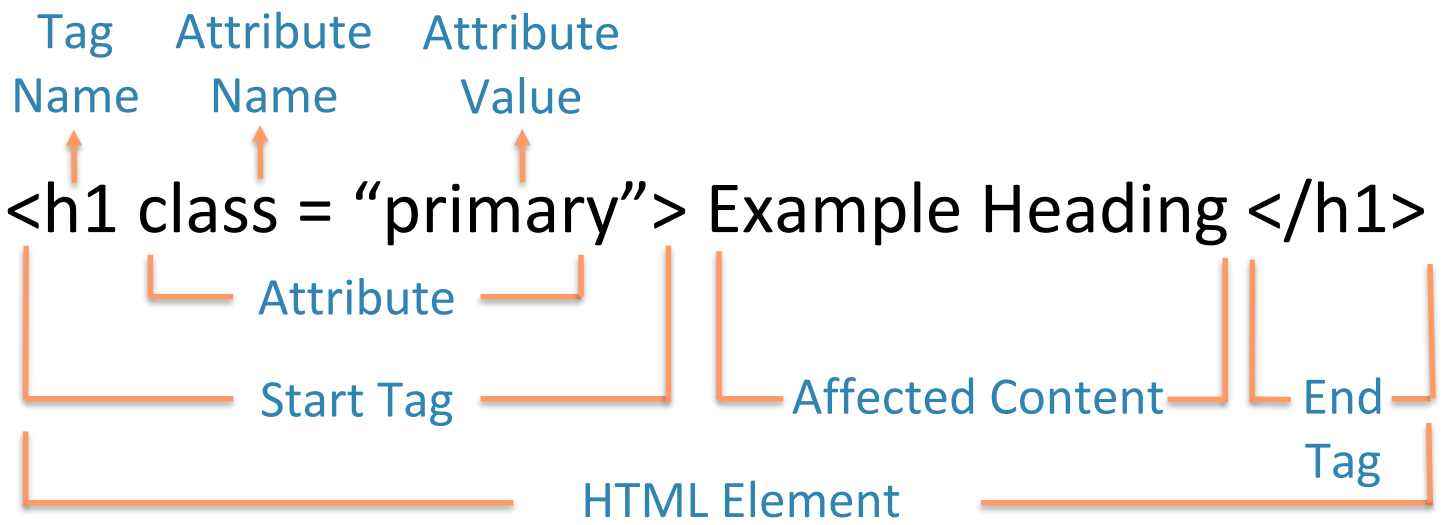
The method returns an element locator that can be used to perform actions on this page / frame. Locator is resolved to the element immediately before performing an action, so a series of actions on the same locator can in fact be performed on different DOM elements. That would happen if the DOM structure between those actions has changed.

Learn more about locators.

@param `selector` — A selector to use when resolving DOM element. See working with selectors for more details

Let’s understand structure of an element





Testleaf
Always Ahead

Diving into Selector Strategies

- CSS
- XPath
- Text Selectors
- Playwright recommended selectors

CSS most common used strategies

- Find an element by tag name

  Syntax: tagName[attribute=value]

- Find an element by ID

  Syntax: tagName[id=value] OR *#idvalue*

  Hash "#" to denote id

- Find an element by class

  Syntax: tagName[class=value] OR *.classname*

  Dot "." to denote class

Testleaf
Always Ahead

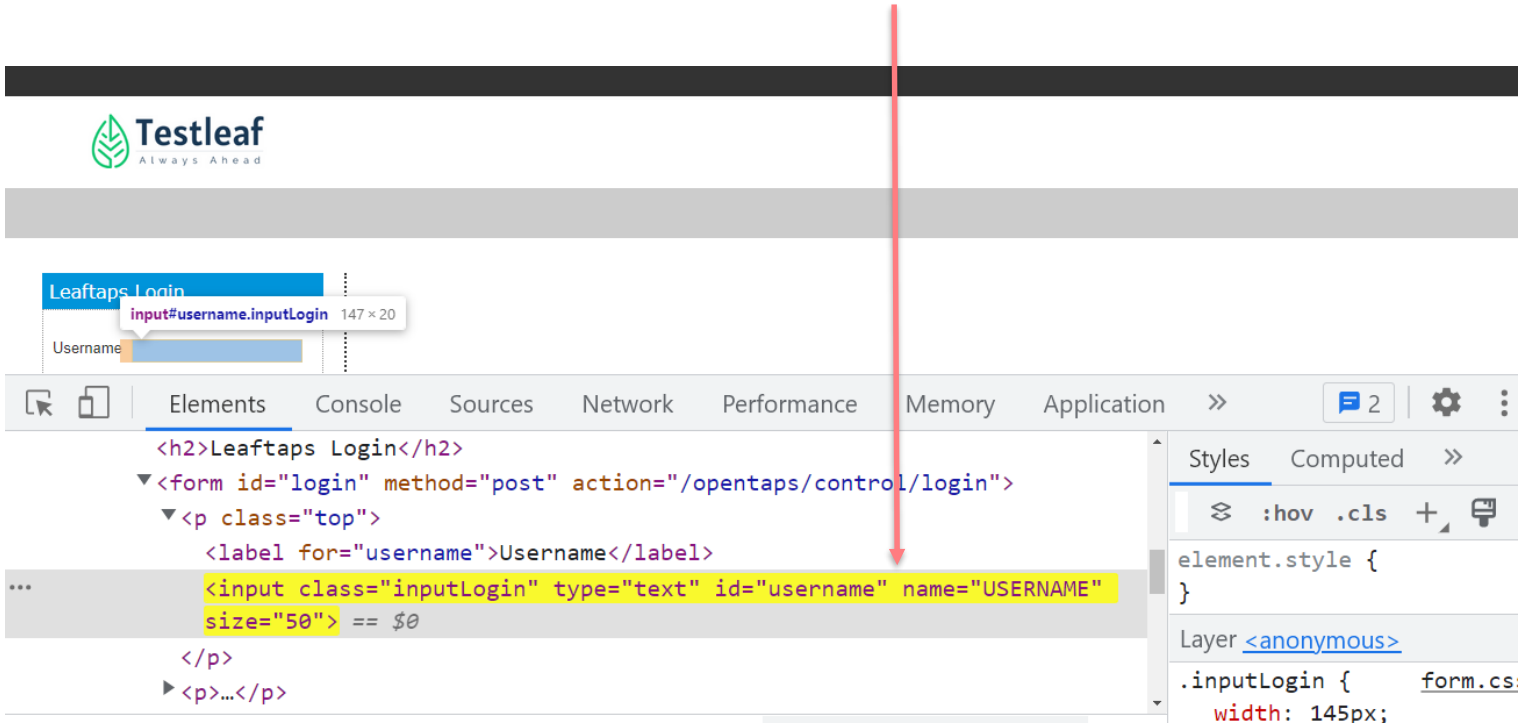
CSS Rules for different situations

- You have a static prefix for your attribute value => tagName[attribute^='prefixVal']
- You have a static suffix for your attribute value => tagName[attribute$='suffixVal']
- You have a static substring for your attribute value => tagName[attribute*='suffixVal']
- You must combine multiple attributes to find element => tagName[attribute1='suffixVal'][attribute2='suffixVal']
- You need to use multiple class name with OR condition => tagName[class='classA'], tagName[class='classB']
- You need to use multiple class name with AND condition => .classA.classB

Testleaf
Always Ahead

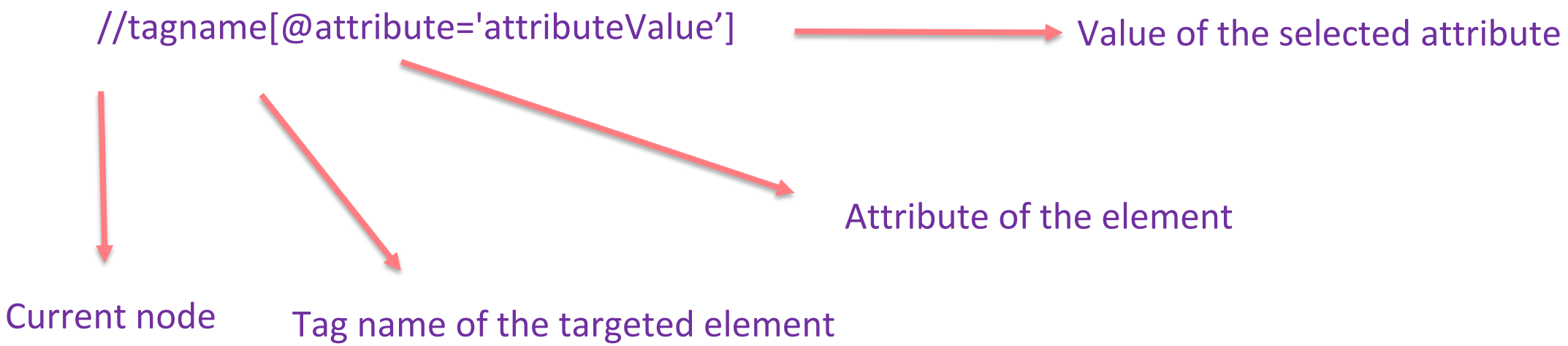
Absolute XPath

- Starts from the root element.
- Specifies the full hierarchy to target an element.
- Prone to breaking when the page structure changes.
- Syntax: /html/body/div[2]/div/div/form/p/input



Relative XPath

- Begins from the current element, not necessarily the root.
- Defines a shorter, context-based path to locate an element
- Offers more robust and flexible element targeting, suitable for dynamic web pages.
- General syntax of the Relative XPath is:



Testleaf
Always Ahead

Relative XPath

- Find element which contains exact text => *//tag[text()='expectedText']*
- Find element which contains substring => *//tag[contains(text(),'substring')]*
- Find element which has class value as given string => *//tag[contains(@class='classValue')]*
- Find element which has class value as given substring => *//tag[contains(@class,'substring')]*
- Find element which has id value as given string => *//tag[contains(@id='idValue')]*
- Find element which has id value as given substring => *//tag[contains(@id,'substring')]*

Testleaf
Always Ahead

Check!

What does the "descendant selector" in CSS do?

- Selects all child elements of a parent element
- Selects all elements with the same class
- Selects elements based on their IDs
- Selects the first child element of a parent

Testleaf
Always Ahead

Playwright recommended Locator Strategy

- getByRole()
- getByText()
- getByLabel()
- getByPlaceholder()
- getByAltText()
- getByTitle()
- getByTestId()

Testleaf
Always Ahead

Industry Best Practices

- Writing locator which are unique
- Writing locator which are readable and can provide some context
- Using text based locator
- Using a unique attribute dedicated for testing like data test id etc.,

Q&A

Testleaf
Always Ahead

Thank you!

Testleaf
Always Ahead