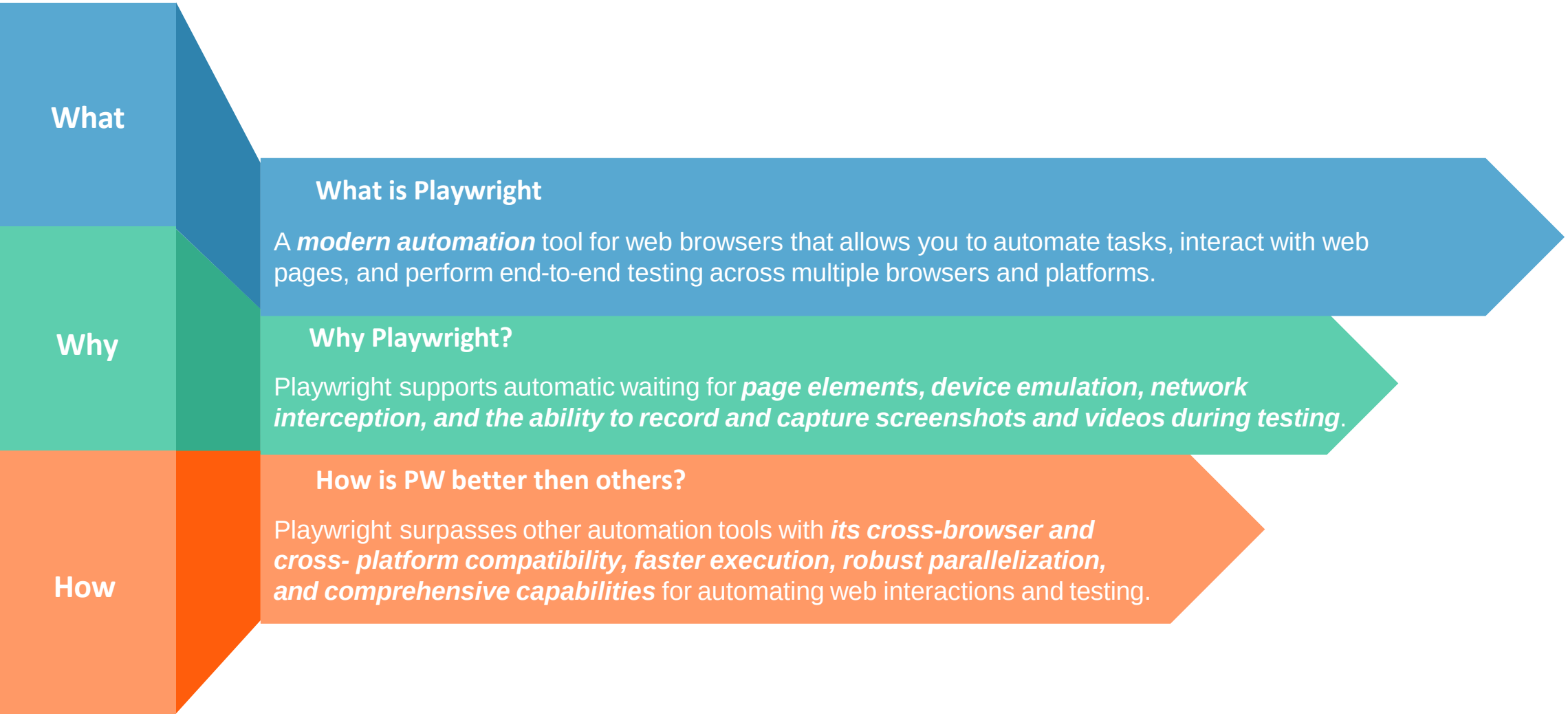
Let’s get into action with Playwright

Testleaf
Always Ahead

Agenda – Part 1 (Playwright 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

Tool – Usage across world

Selenium	Playwright	Cypress	Puppeteer
#1 – Max Downloads	Getting Popular	Decent Market Share	Developer Tool

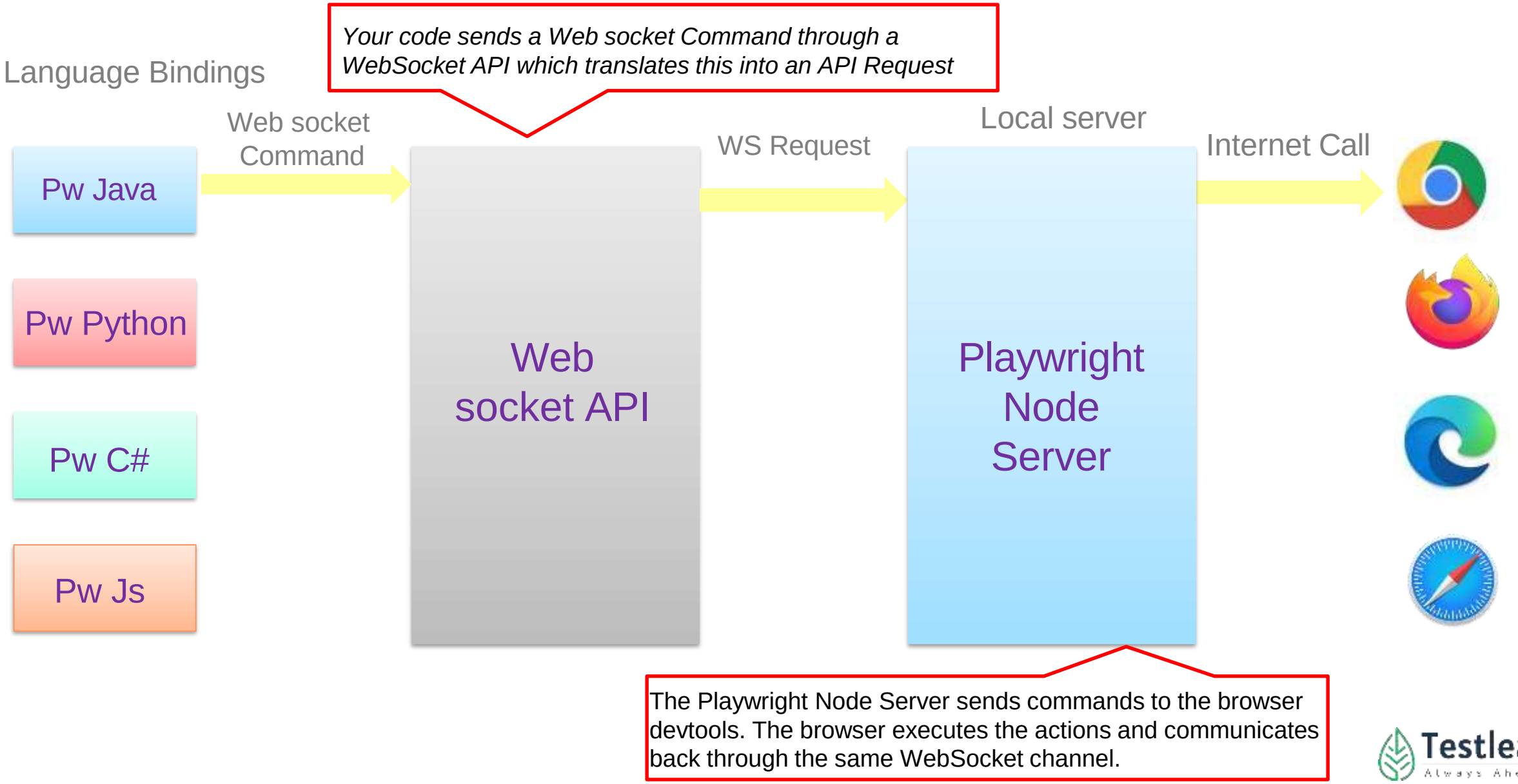
Tool – Scripting Languages

Selenium	Playwright	Cypress	Puppeteer
Core Java, Python, C#, JavaScript, Ruby, Perl	Core Java, Python, C#, JavaScript	JavaScript	JavaScript

PLAYWRIGHT IS THE NEXT SELENIUM

Selenium Architecture

Playwright Architecture



Testleaf
Always Ahead

Websocket Protocol working alongside DevTools Protocol

What is WebSocket?

The **WebSocket protocol** is a **communication protocol** that provides **full-duplex** (two-way) communication channels over a **single, long-lived connection**.

Unlike **HTTP**, which is request-response based, WebSockets allow for continuous, real-time communication between a **client** (like a web browser) and a **server**.

Where it is used :

It allows servers to **push** data to clients **without** the client having to request it, which is ideal for **real-time applications** like:

- Chat applications
- Online gaming
- Live financial dashboards

How it works :

The WebSocket connection communicates directly with **Chrome's DevTools Protocol (CDP)** to:

- Capture **network traffic**.
- Inspect **DOM elements**.
- Perform **performance audits**.

Selenium Vs Playwright Architecture Difference

Feature	Selenium	Playwright
Protocol	W3C WebDriver Protocol	WebSocket API
Drivers	Requires browser-specific drivers (e.g., ChromeDriver)	No drivers needed; uses a single Node server
Speed	Slower due to extra layers of communication	Faster due to WebSocket's direct communication
Setup Complexity	Higher (driver installation needed)	Easier (drivers are bundled with Playwright)
Parallel Testing	Limited without additional setup	Built-in support for parallel testing

Browser Architecture: Key Components

- User Interface [UI]
- Browser Engine [Controller]
- Rendering Engine [Layout Painting]
- JavaScript Engine [Code Execution]
- Networking[Fetching Resources]
- UI Backend [Rendering Browser UI]
- Data Storage.

Browser Architecture: Key Components

User Interface [UI] :

This includes:

- Address bar (for entering URLs).
- Back, forward, refresh buttons.
- Bookmarks, tabs, settings, and menus.

☞ It provides the interface for user interaction.

Browser Engine (Controller):

- Acts as the bridge between the UI and the Rendering Engine.
- Handles navigation (e.g., when you type a URL and press enter).
- Manages browser settings and security.

☞ It decides how the webpage should be processed and displayed.

🔹 Examples of browser engines:

❖ Gecko → Used in Mozilla Firefox.
❖ Blink → Used in Google Chrome, Edge, and Opera.
❖ WebKit → Used in Safari.

Browser Architecture: Key Components

Rendering Engine (Layout & Painting)

This includes:

- Parses HTML and CSS to construct a DOM (Document Object Model) and CSSOM (CSS Object Model).
- Creates a Render Tree and computes the layout of elements.
- Paints the final layout onto the screen.

👉 It is responsible for rendering web pages visually.

🔹 Examples of browser engines:

❖ Blink (Chrome, Edge).
❖ Gecko (Firefox).
❖ WebKit (Safari).

Networking (Fetching Resources)

- Handles HTTP/HTTPS requests for web pages, images, stylesheets, scripts, etc.
- Uses protocols like HTTP/2, WebSockets for fast data transfer.

👉 It downloads web content from servers.

Browser Architecture: Key Components

JavaScript Engine (Code Execution)

- Parses and executes JavaScript code.
- Handles DOM manipulation and user interactions dynamically.
- Uses JIT (Just-in-Time) Compilation to optimize JavaScript execution.

☞It powers dynamic content in web pages.

🔹 Common JavaScript Engines:

❖ V8 (Chrome, Node.js).
❖ SpiderMonkey (Firefox).
❖ JavaScriptCore (JSCore) (Safari).

UI Backend (Rendering Browser UI)

- Handles drawing of UI components like buttons, text boxes, scrollbars, etc.
- Uses OS-native graphics APIs for rendering.

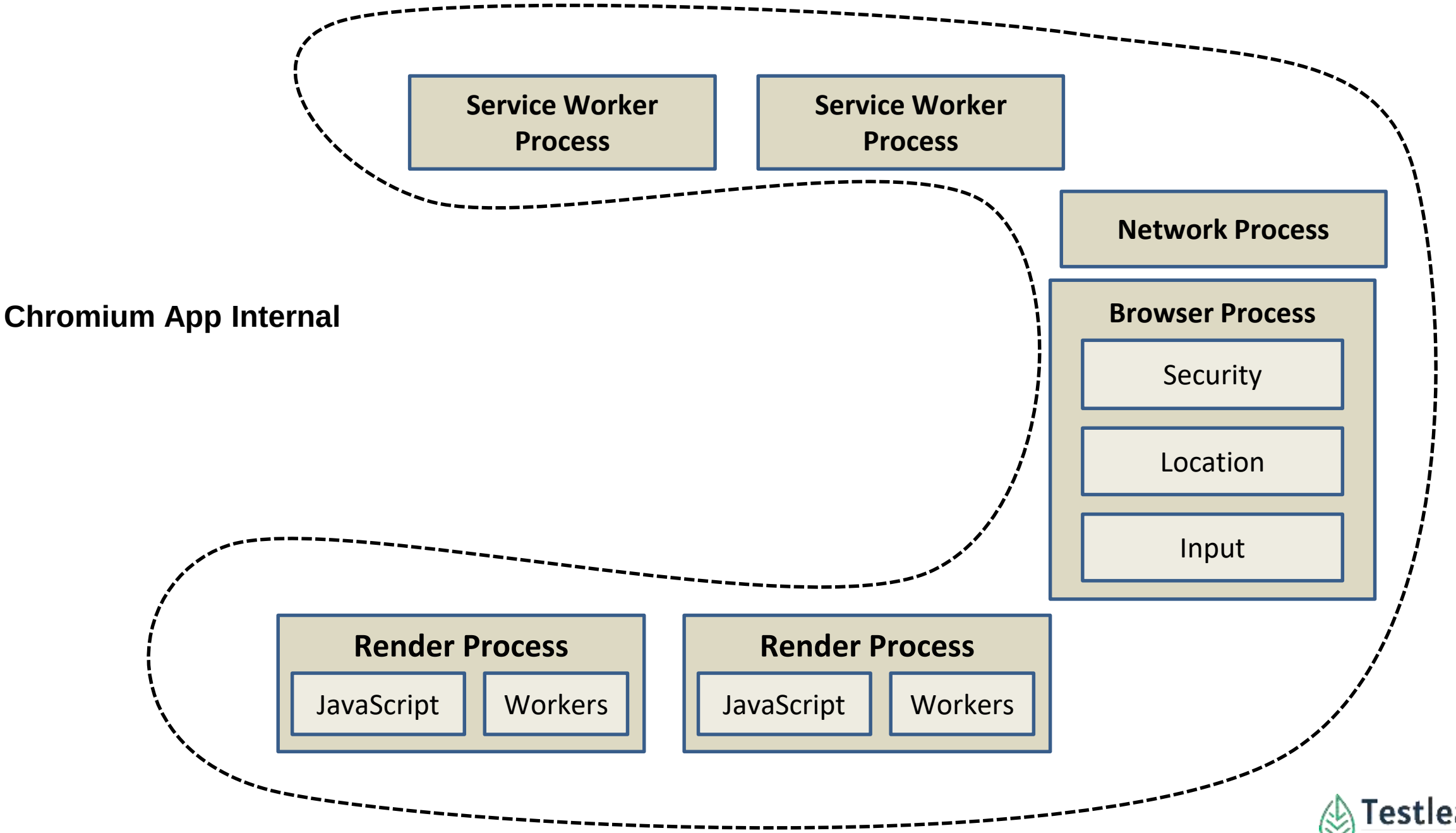
☞ It draws the browser's UI elements.

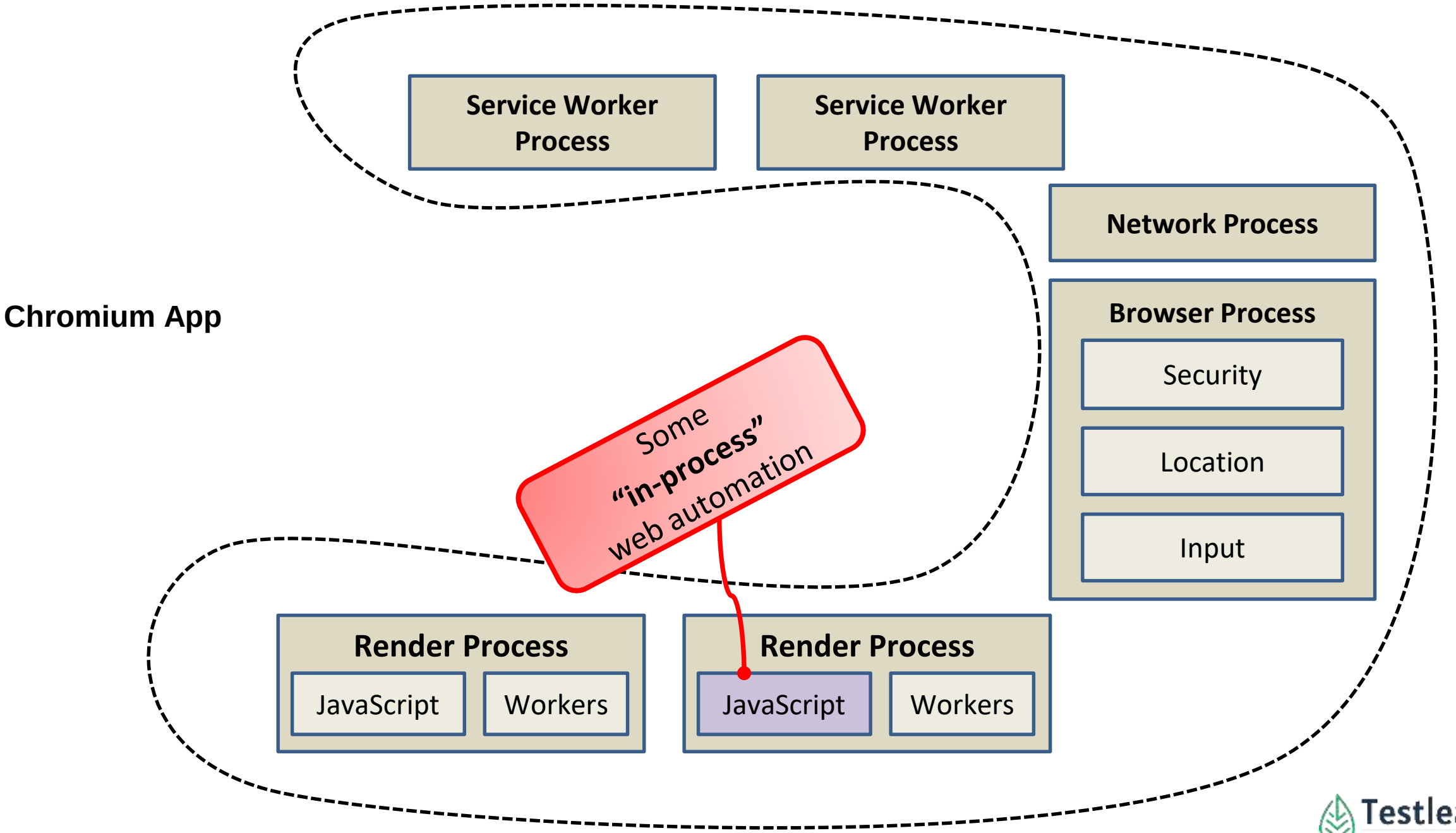
Browser Architecture: Key Components

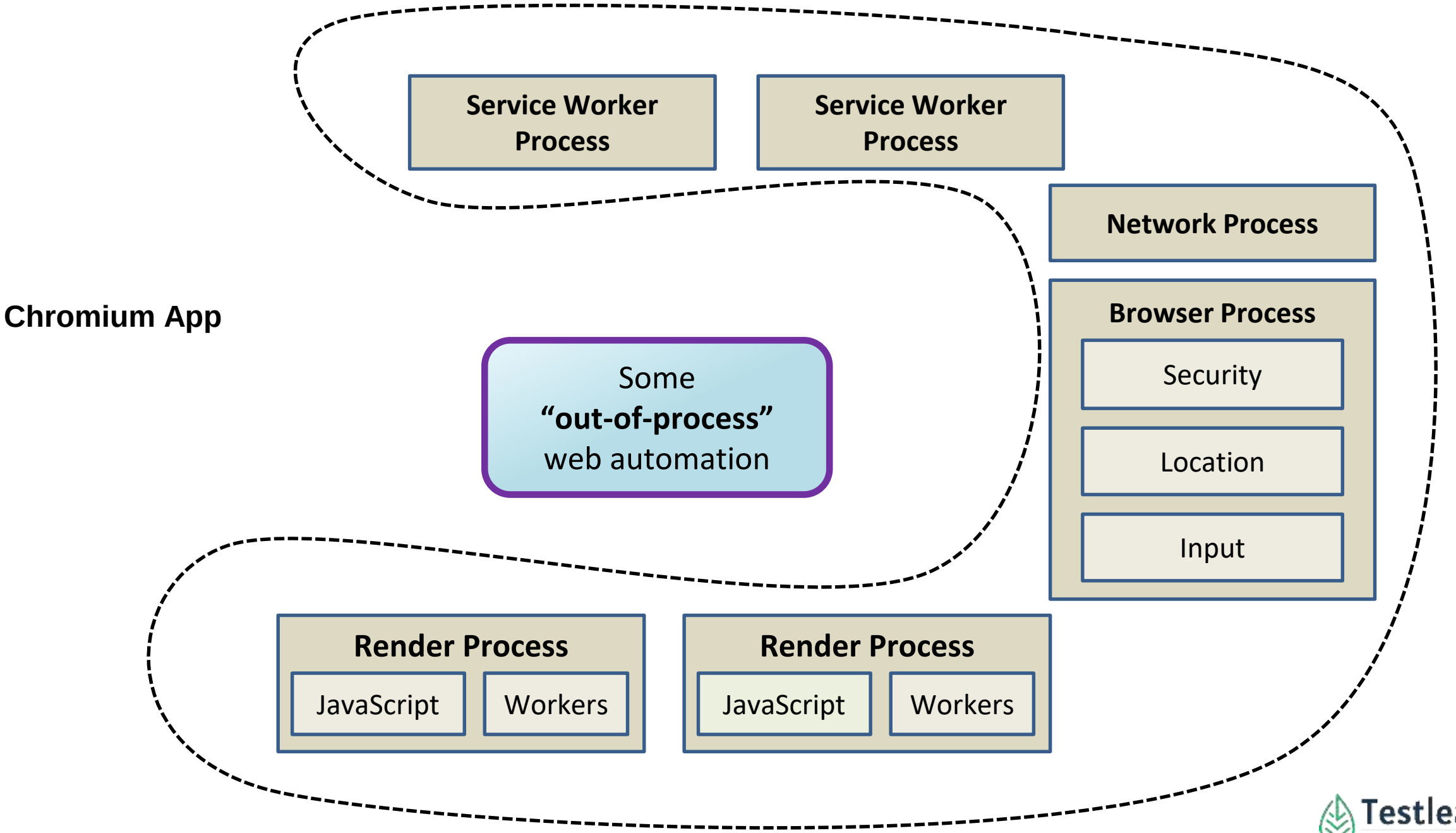
Data Storage (Local Storage & Caching)

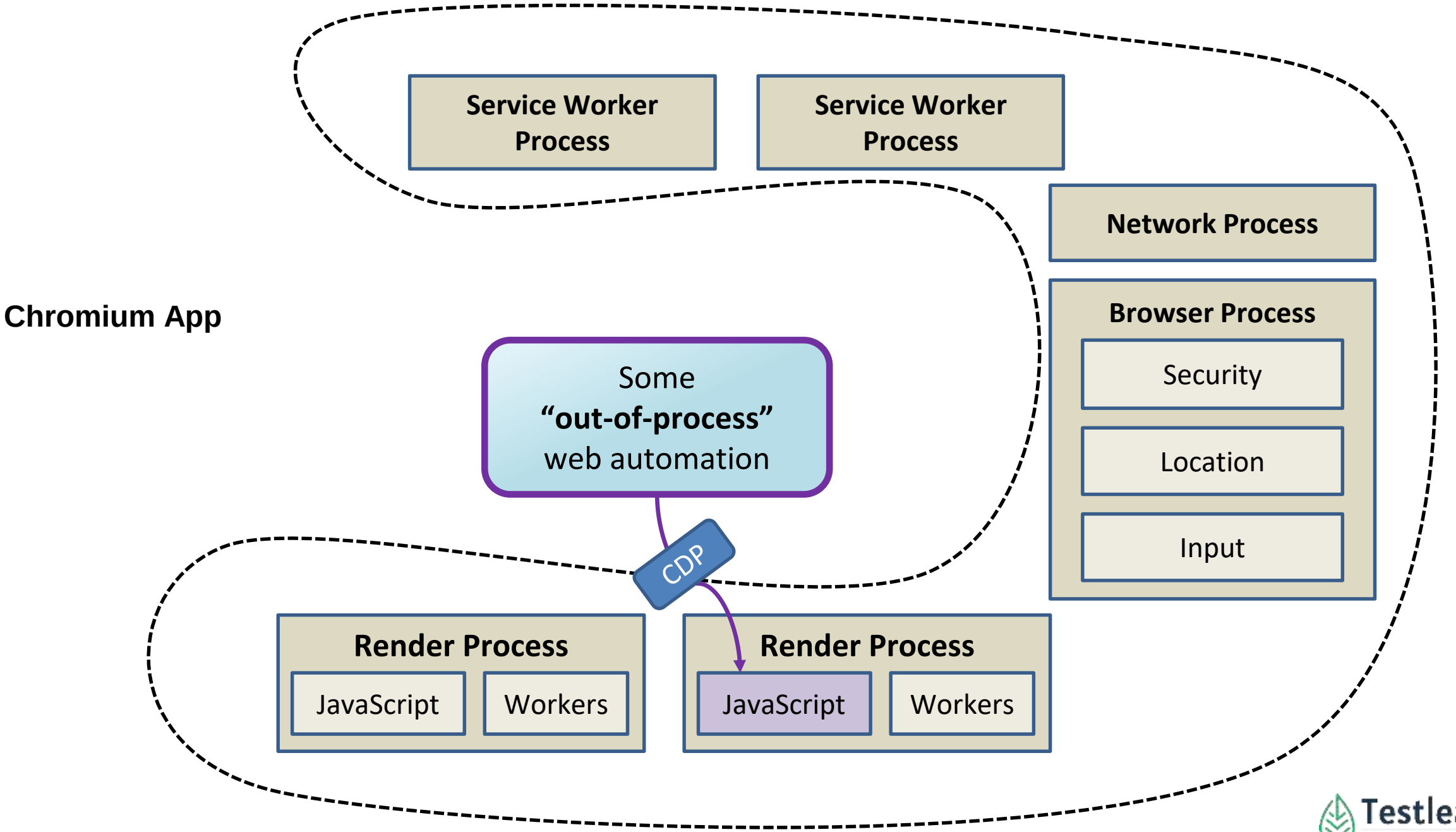
- Stores data such as:
- Cookies (for user sessions).
- Cache (for faster page loads).
- LocalStorage (for storing data persistently).
- Session Storage (temporary storage per tab).

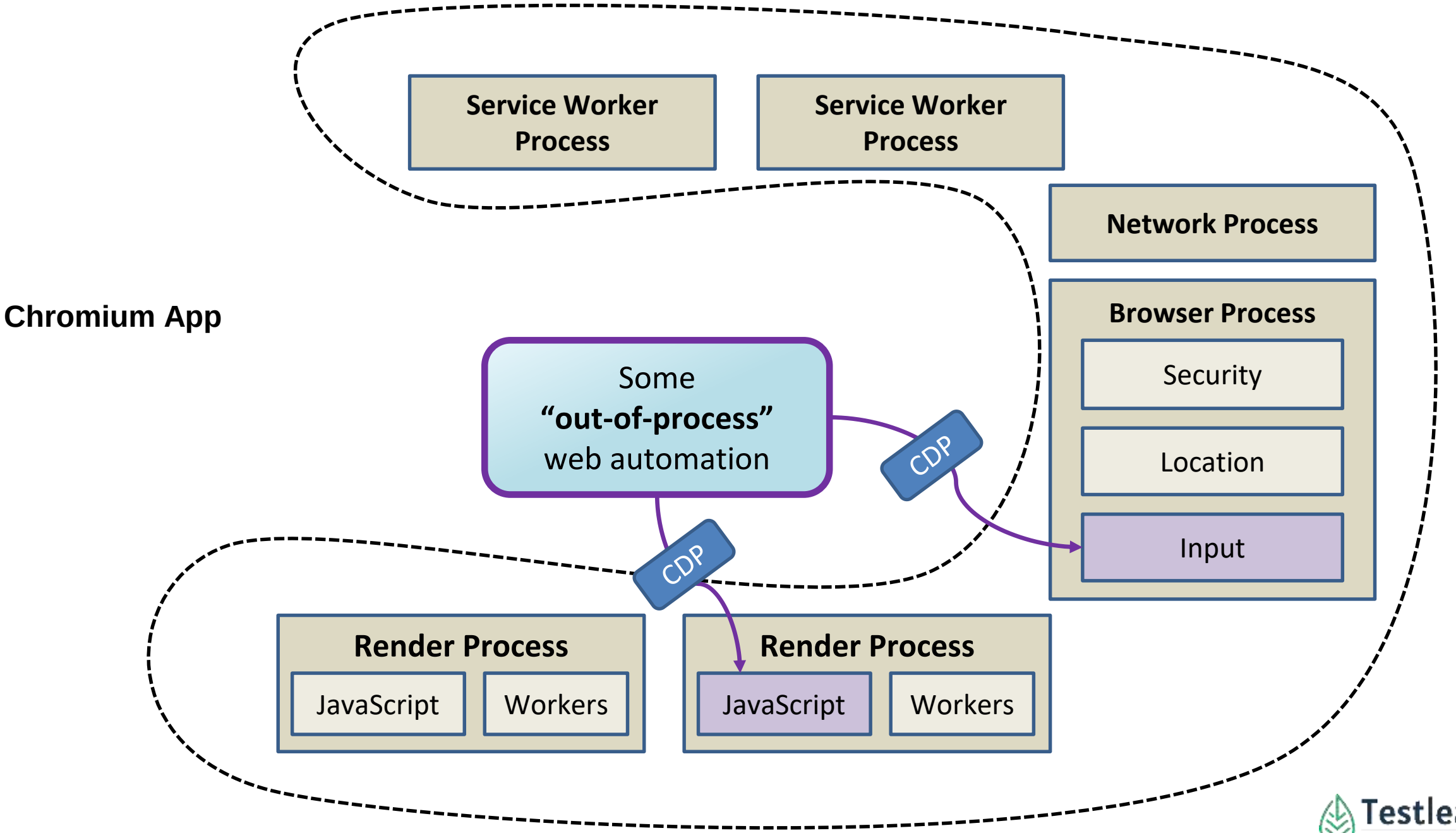
☞ It improves performance and offline capabilities.

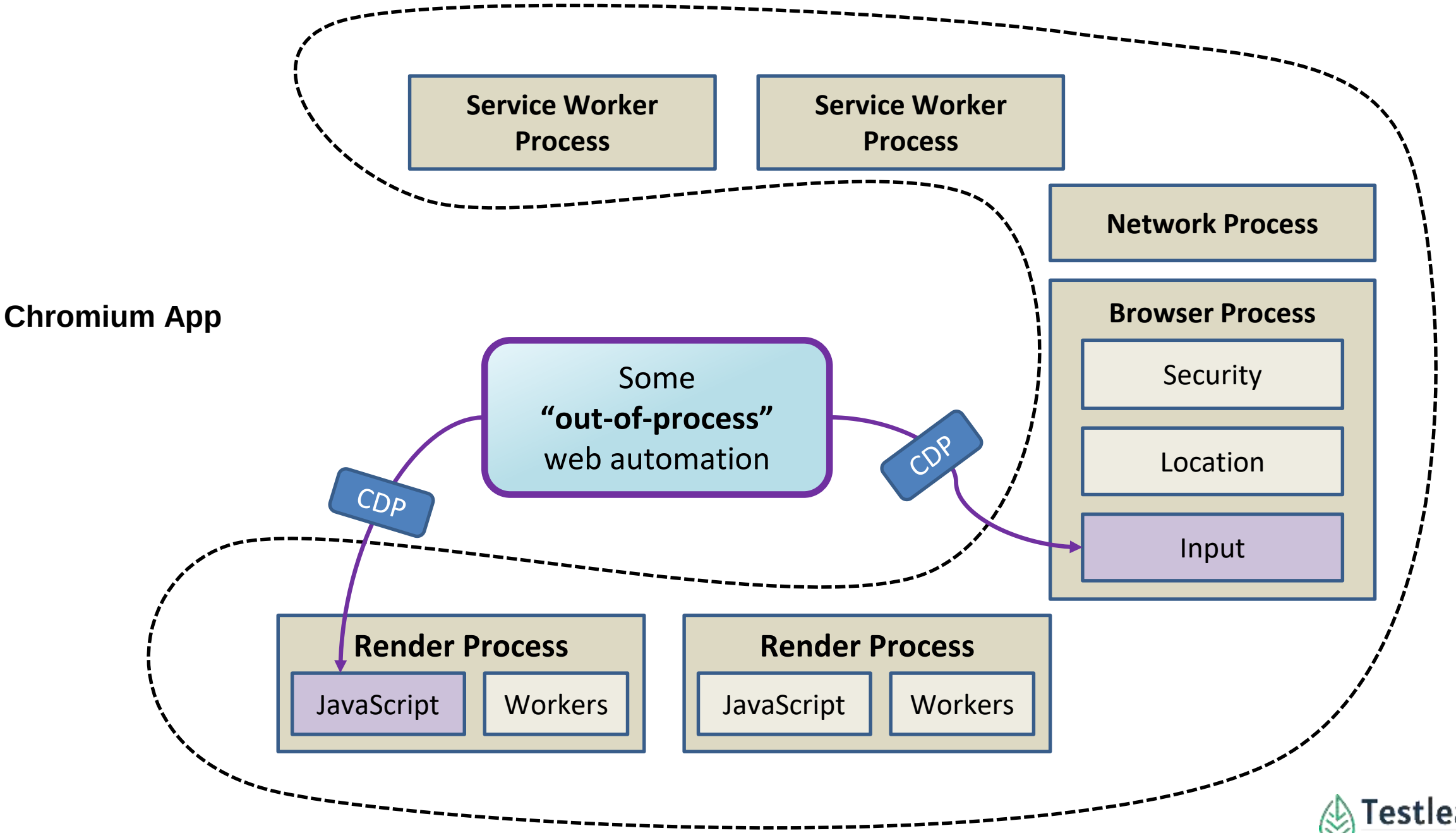


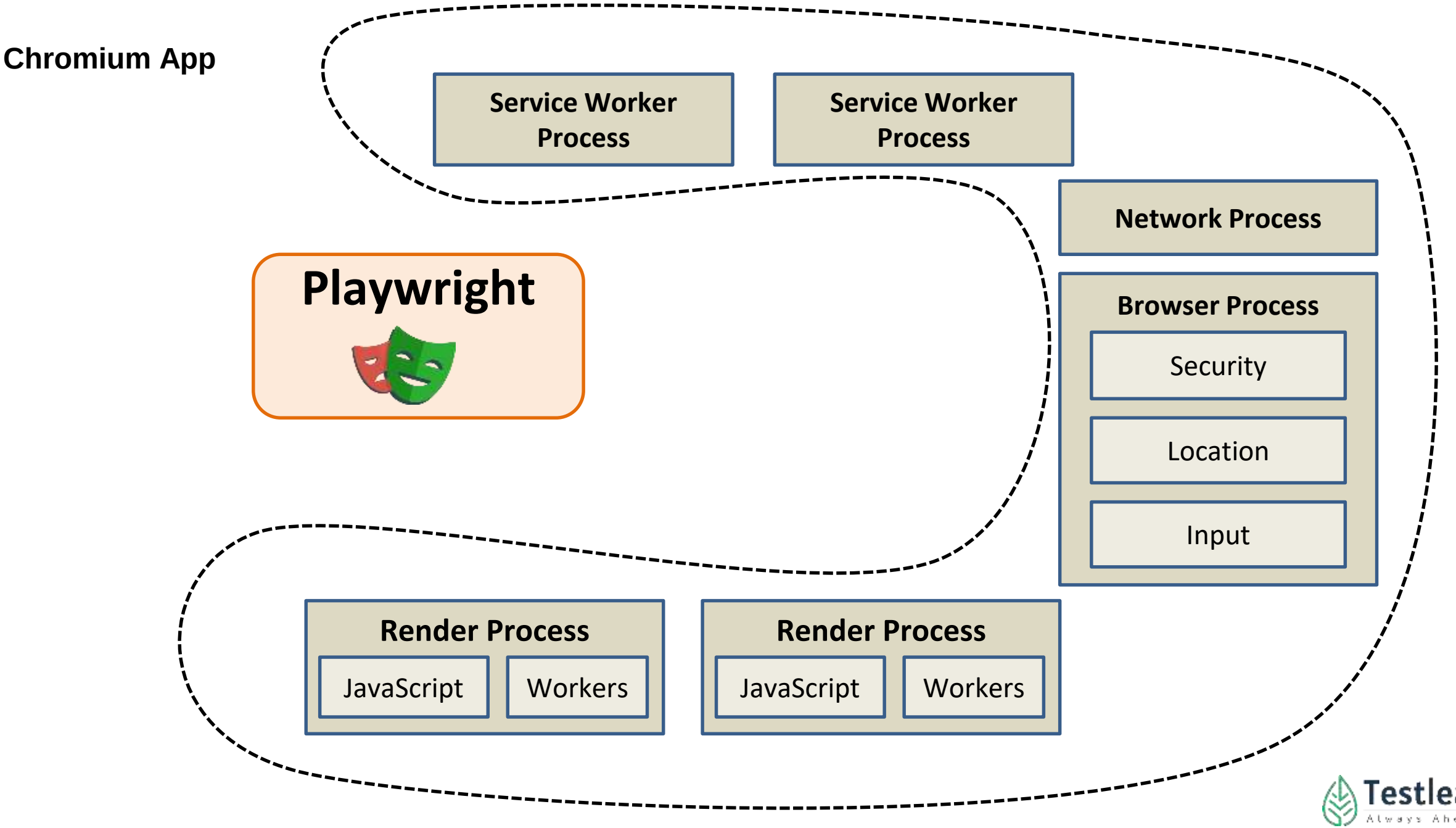




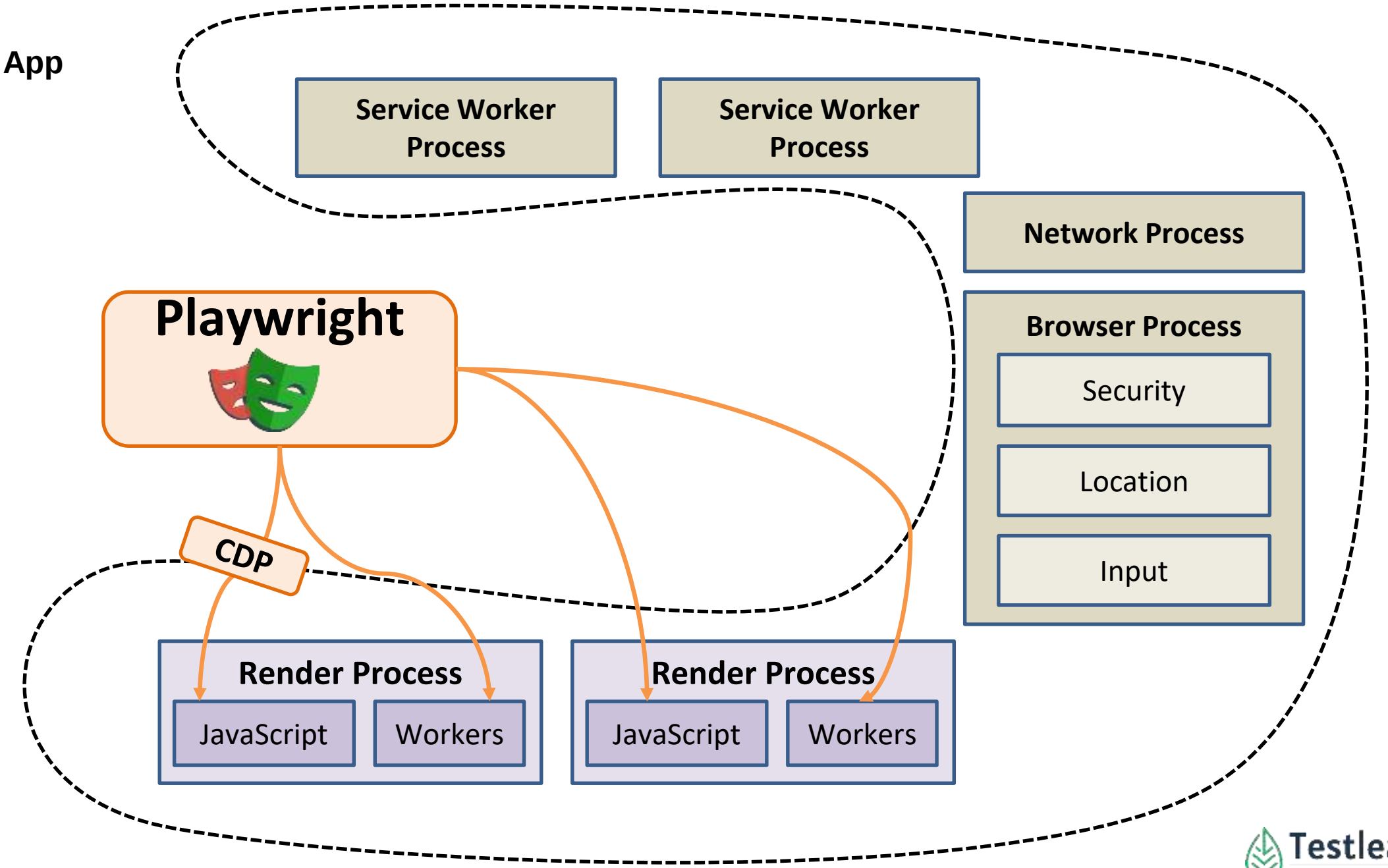




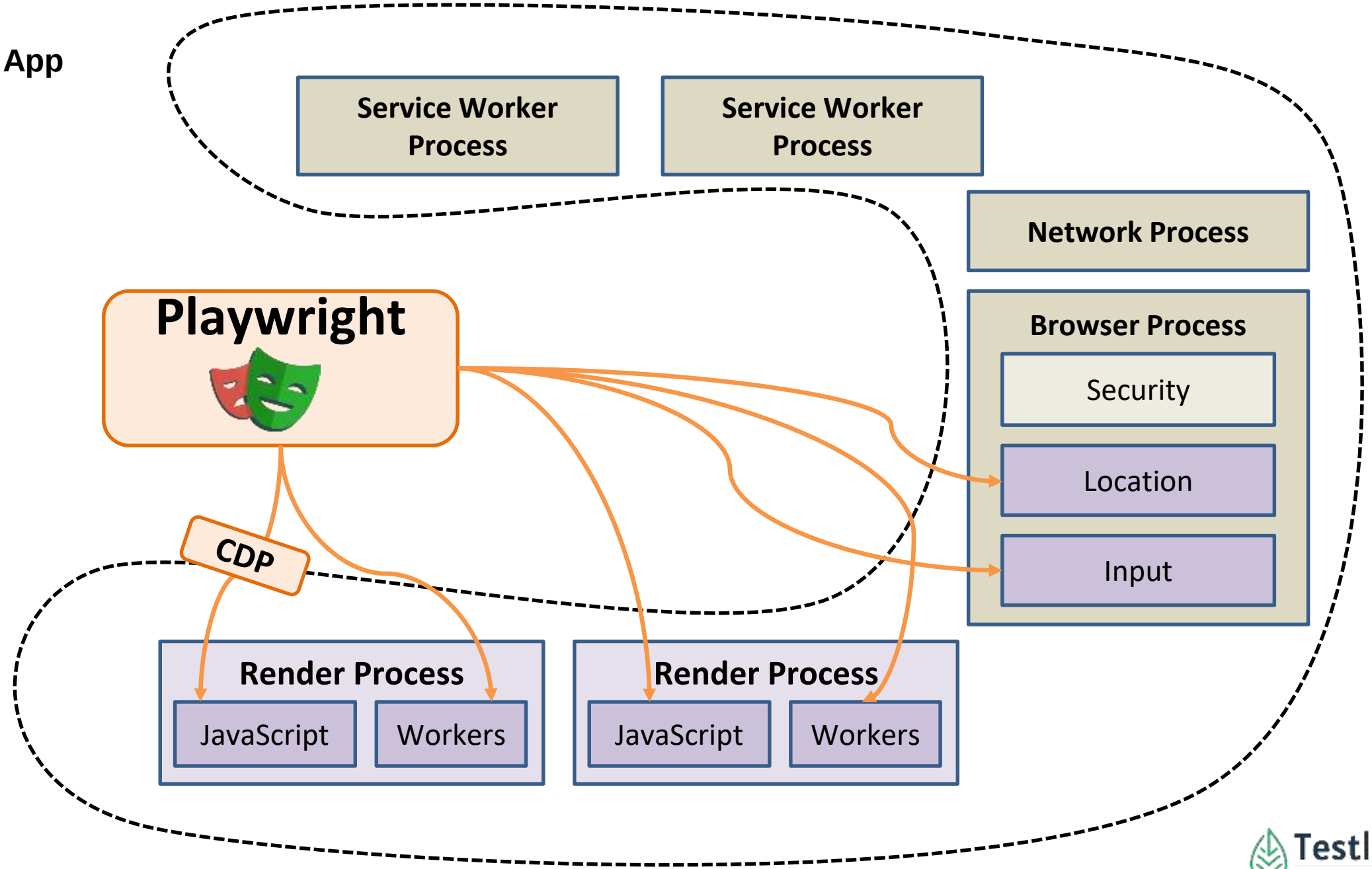




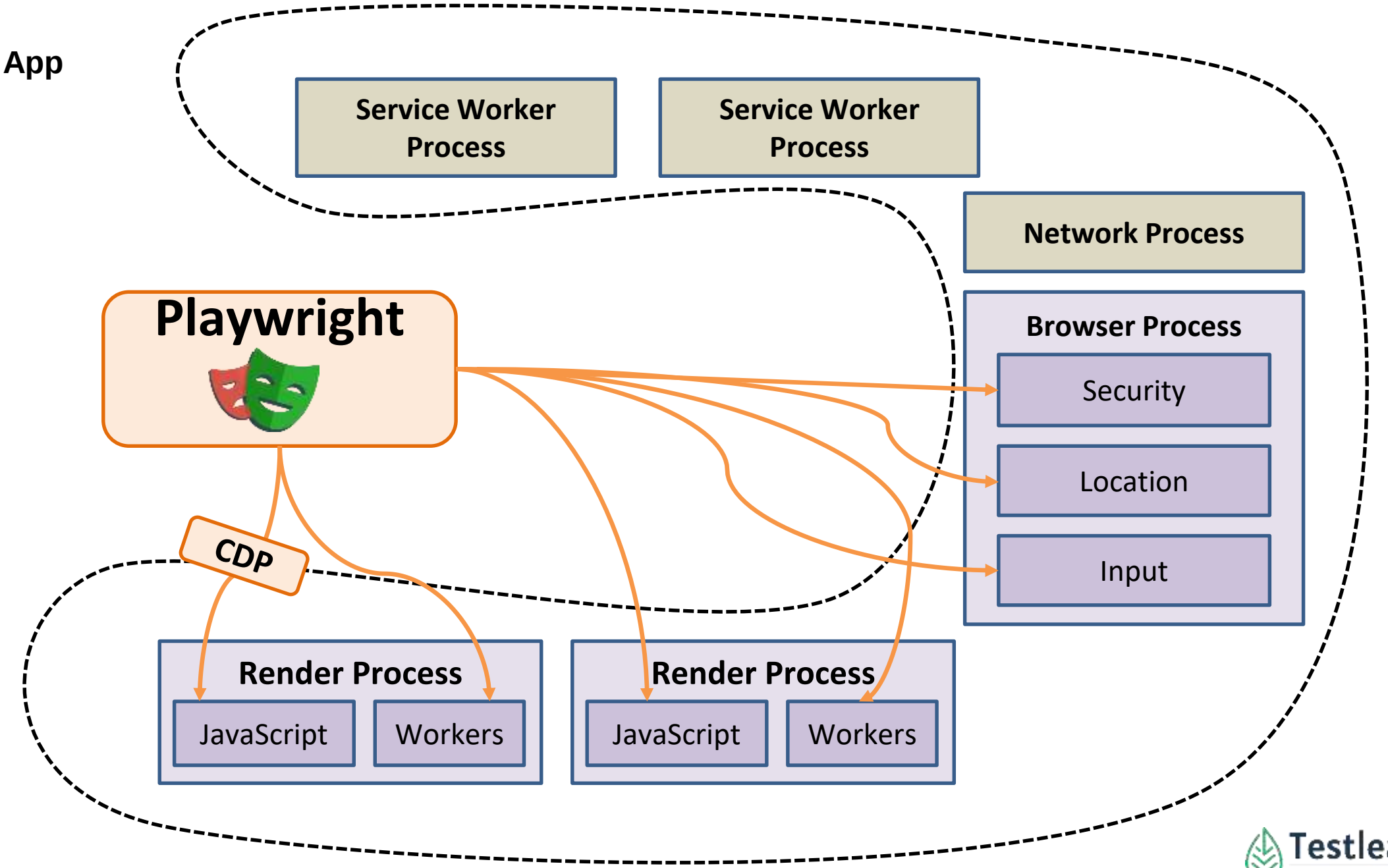
Chromium App



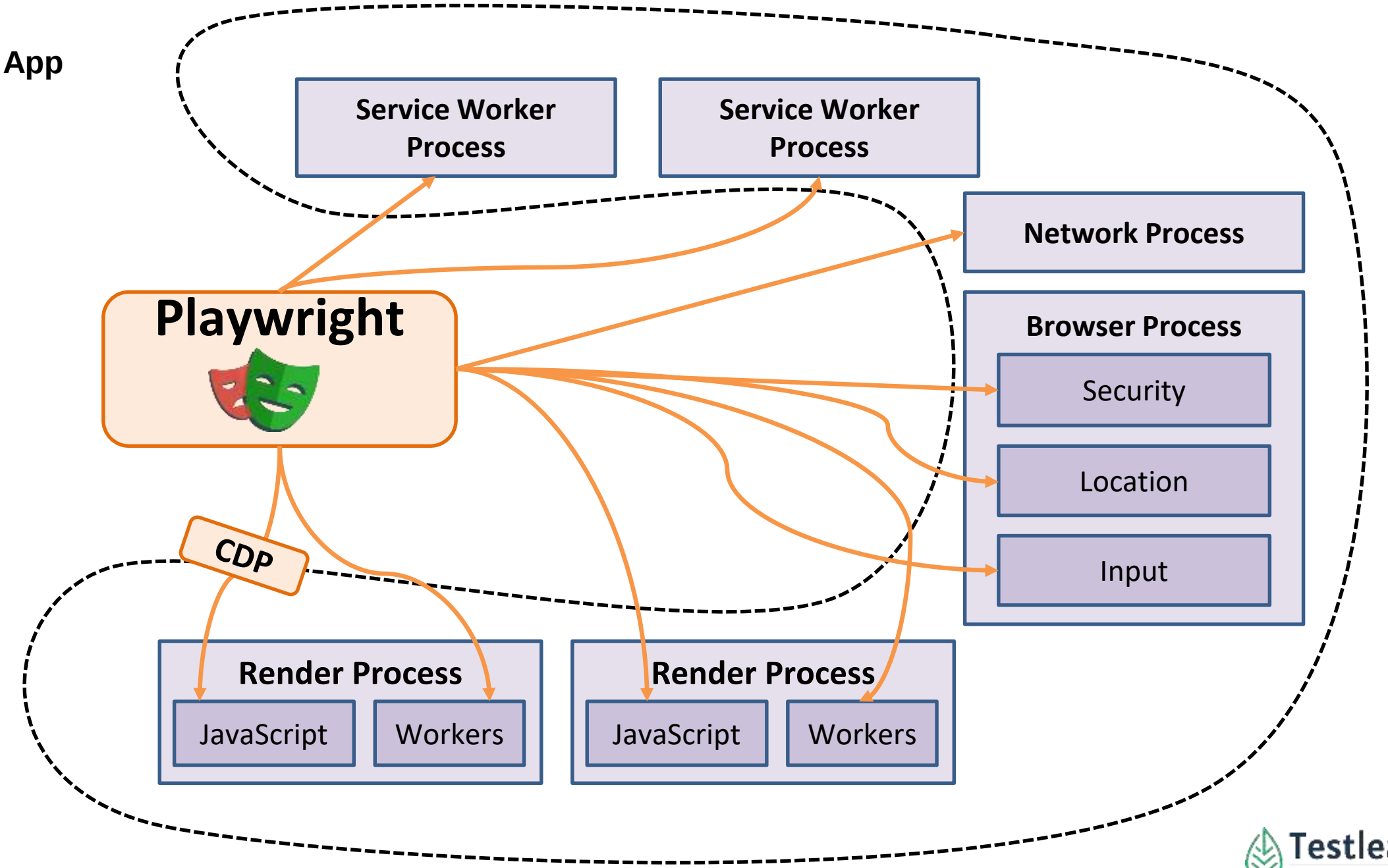
Chromium App

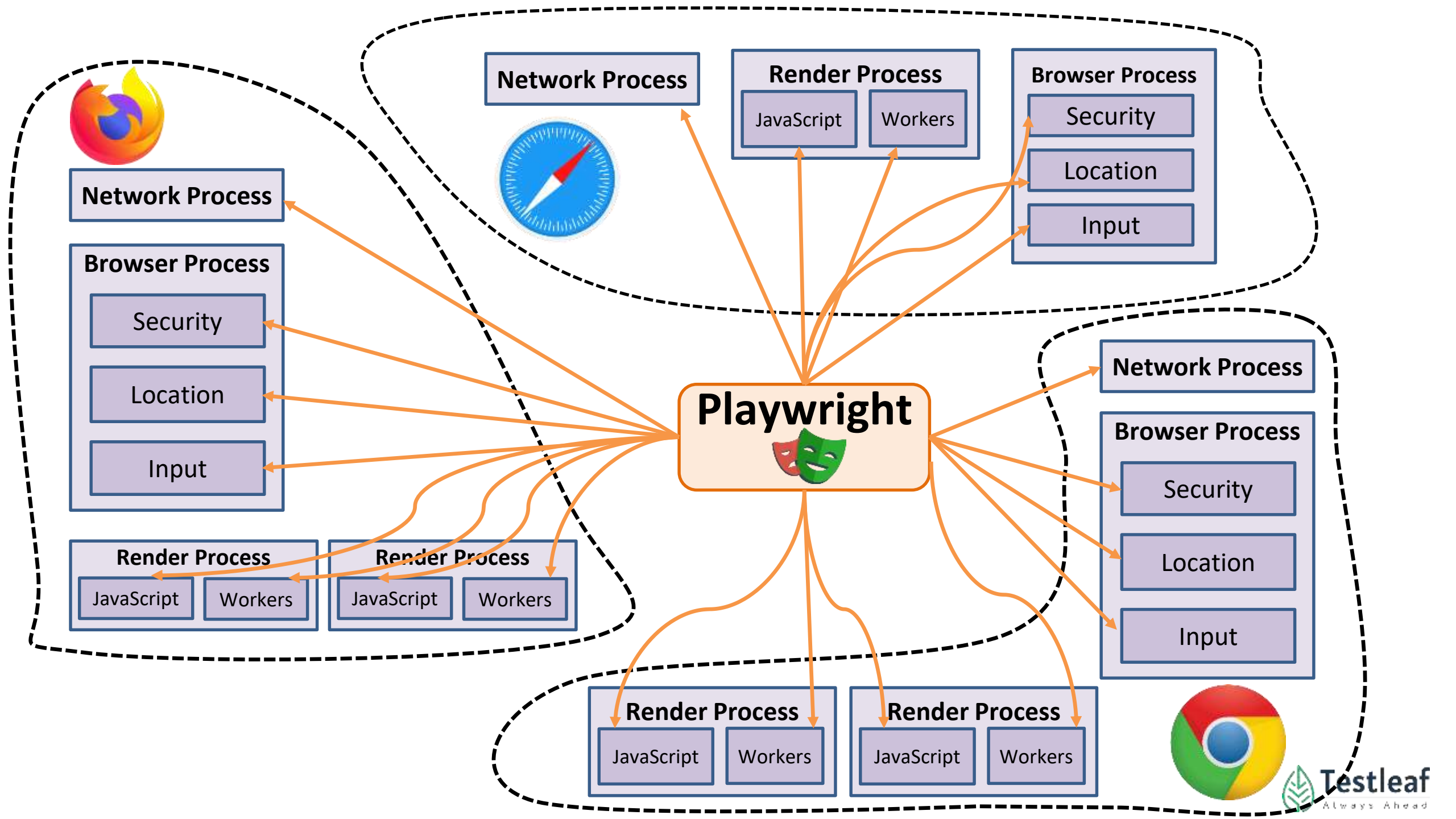


Chromium App



Chromium App




Network Process
Render Process
JavaScript
Workers
Browser Process
Security
Location
Input
Network Process
Browser Process
Security
Location
Input
Playwright
Render Process
JavaScript
Workers
Render Process
JavaScript
Workers
Network Process
Browser Process
Security
Location
Input
Render Process
JavaScript
Workers
Render Process
JavaScript
Workers
Testleaf
Always Ahead

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.*
- *Monitor and intercept network traffic. Mock or modify API calls*

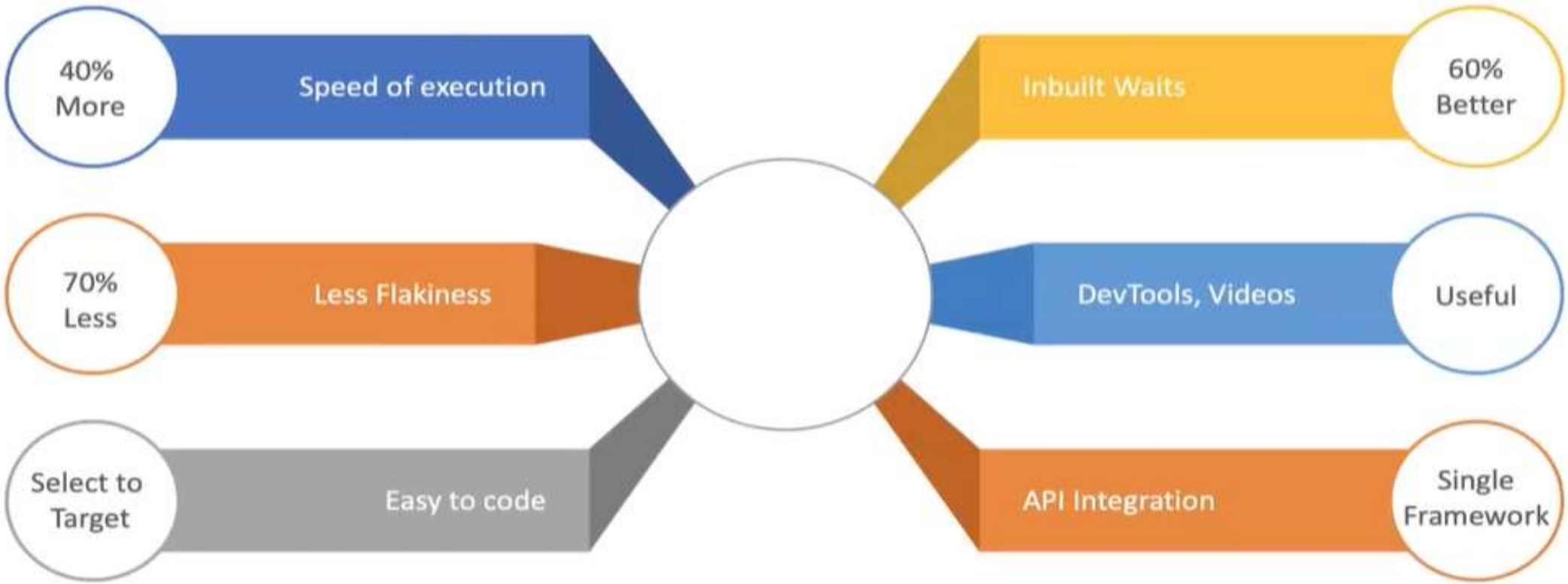
Service worker process

- *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***

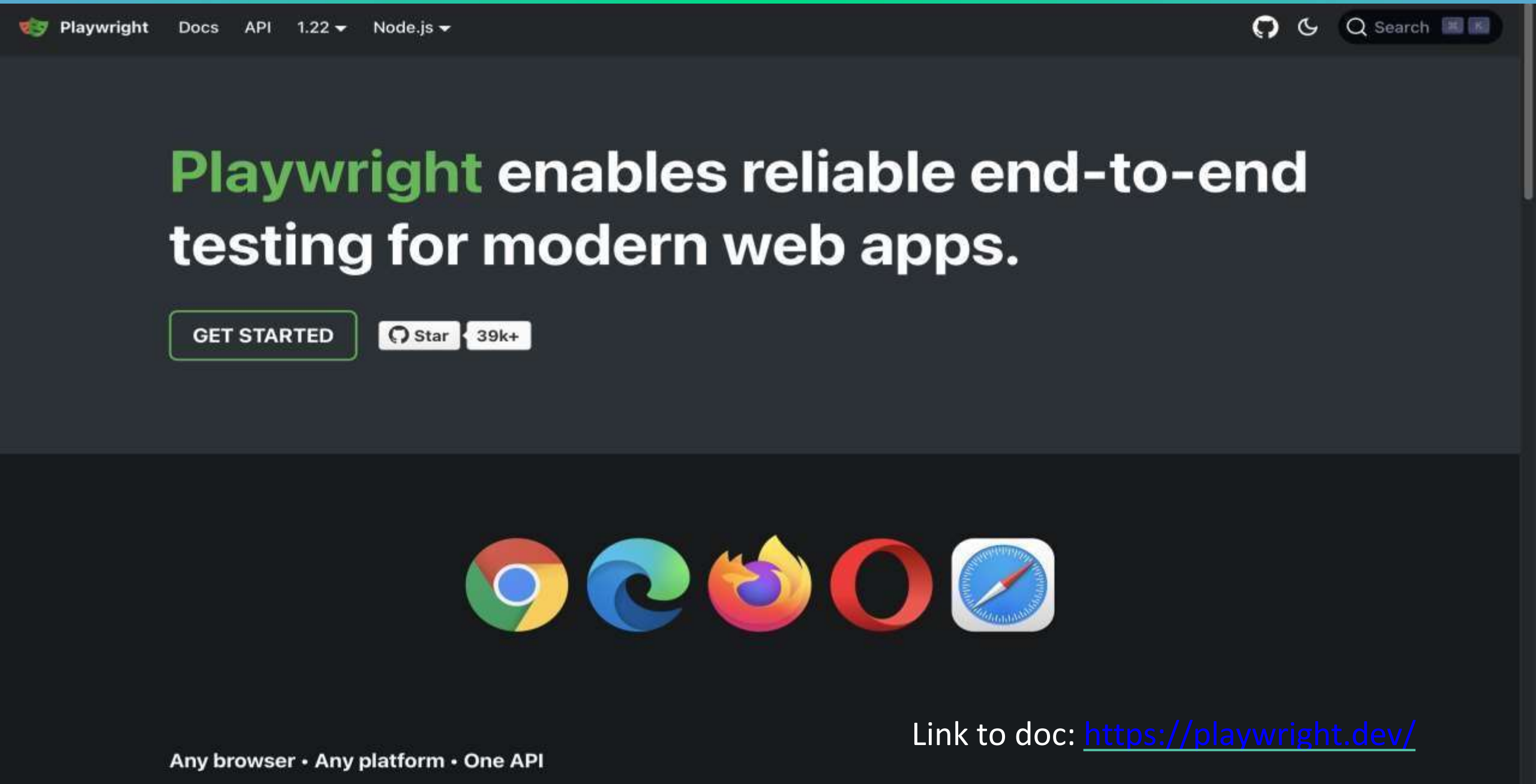
About DevTools and benefits using Playwright

Component	What it Does	Playwright's Benefit
Service Worker Process	Handles background tasks for web applications, such as **push notifications and caching**.	Control over offline functionality testing. Inspect and debug service worker behaviour for progressive web apps (PWAs).
Network Process	Manages all network communication, including **HTTP requests, responses, and web sockets.**	Monitor and intercept network traffic. Mock or modify API calls during tests (e.g., simulate slow networks or error responses).
Browser Process	Ensures browser **security, location permissions, and user** input handling.	Automates testing permissions like geolocation or notifications. Controls input events like mouse clicks and keyboard actions with precision.
Render Process	Executes the **JavaScript code and manages rendering workers** (visual elements on the webpage).	Access to the DOM for precise manipulation and assertion during tests. Simulate complex user interactions with JavaScript-heavy applications.

WHY PLAYWRIGHT?



Playwright Documentations

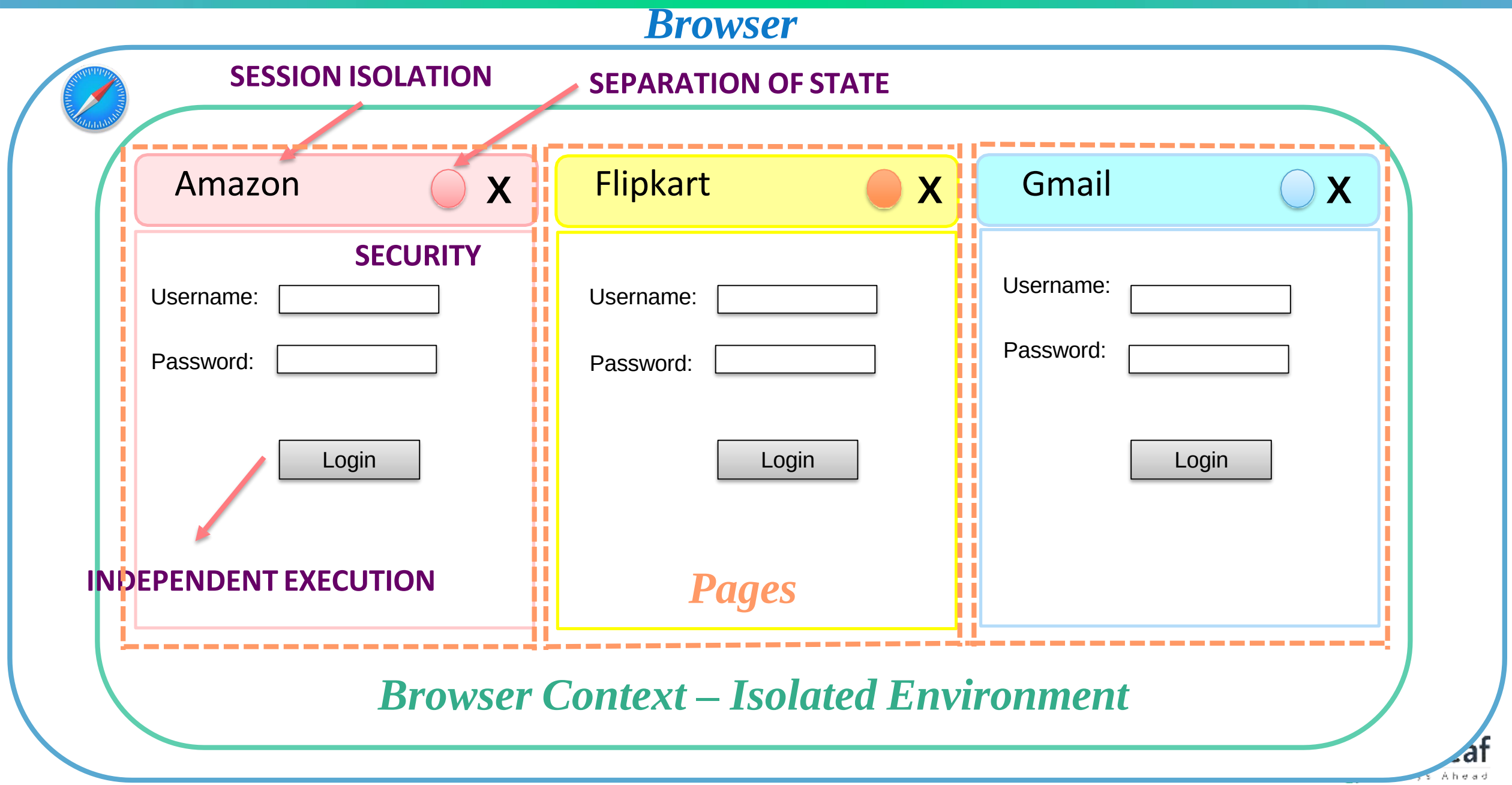


Link to doc: https://playwright.dev/

Agenda – Part 2 (Playwright Key Concepts)

- Browser
- BrowserContext
- Page
- DOM
- WebElement

Browser, Browser Context & Page

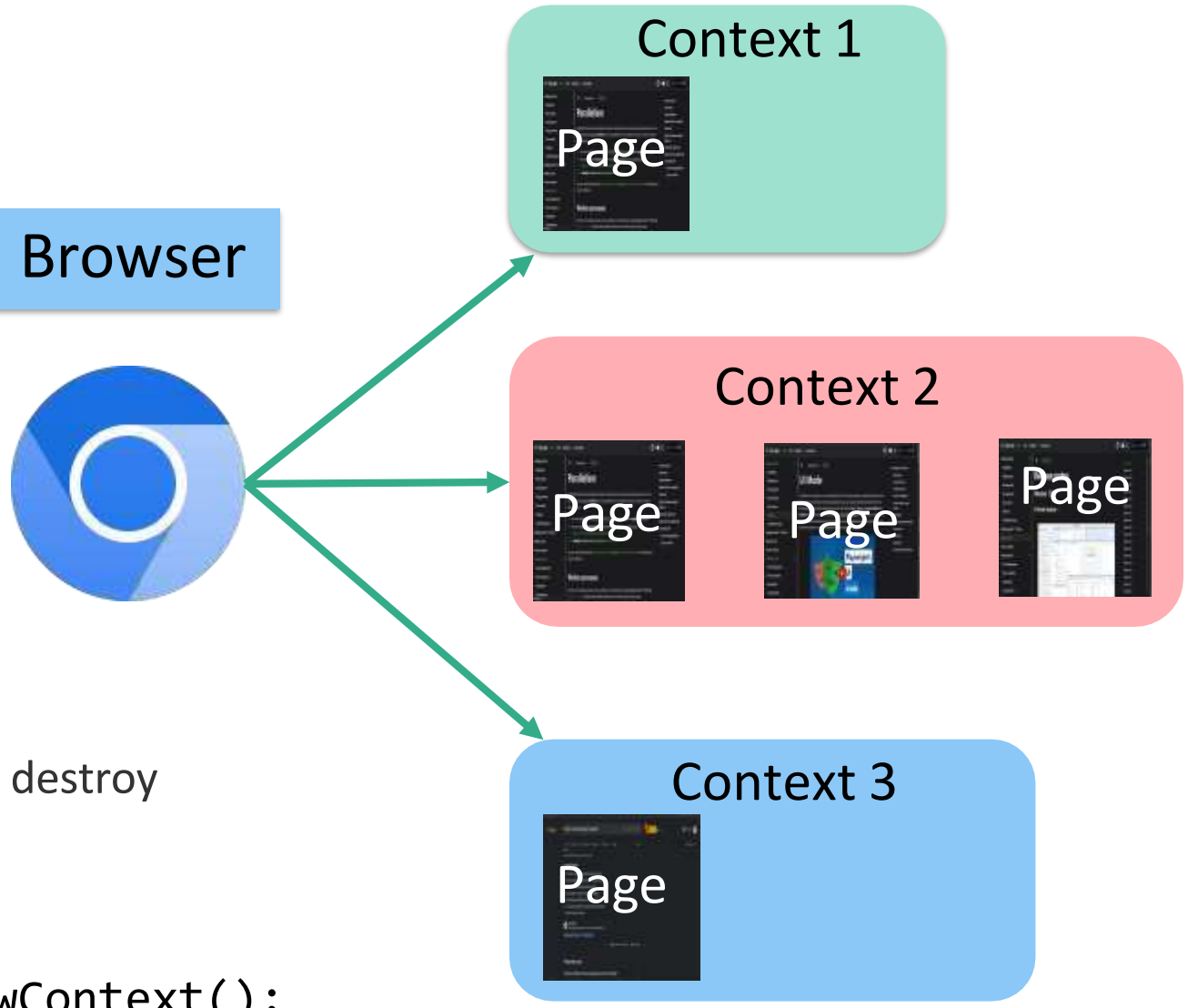


Browser & Browser Context

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
- Code Snippet

```
const context = await browser.newContext();
```

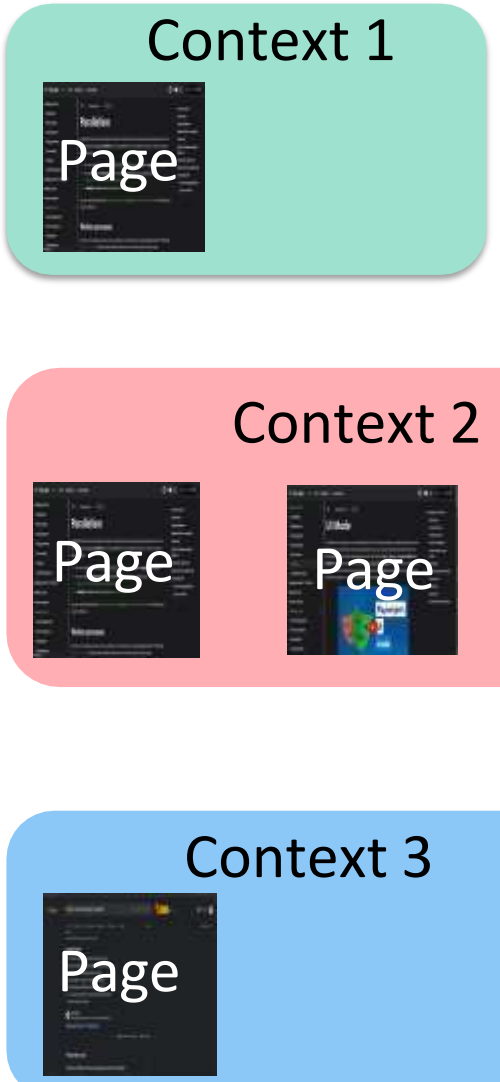


Browser Context & Pages

Page

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

```
const page = await context.newPage();
```



Implementation of Browser Context and Page

```
import { chromium, test } from "@playwright/test";

test(`Test to launch the browser`, async () => {

    // Create a browser instance
    const browser = await chromium.launch({headless: false, channel: 'chrome'});

    // Create the browser context
    const context = await browser.newContext();

    // Create a new page
    const page = await context.newPage();


    // Load the url
    await page.goto("http://leaftaps.com/opentaps/control/main");

    // Get the title of the page
    console.log(`The title of the page is ${ await page.title()}`);

    await page.waitForTimeout(5000);
})
```

Playwright Test execution – Analogy to RELAY RACE.

❑ Organizer Starts the Race (Test Runner Initiates the Test):

The **race organizer** starts the event.
Similarly, when you run `npx playwright test`, the **Playwright Test runner** starts looking for test cases (`test()` functions) to execute.

❑ First Runner (Launch Browser):

The **first runner** holds the baton and starts running. In your test, this is like **launching the browser**:

```
const browser = await chromium.launch({headless: false, channel: 'chrome'});
```

❑ Second Runner (Create Context & New Page):

The second runner grabs the baton only when the first runner reaches them.
This is like creating the browser context and opening a new page:

```
const context = await browser.newContext();

const page = await context.newPage();
```

Note : Without waiting, the test could fail—just like dropping the baton in a relay race!

Testleaf
Always Ahead

Playwright Test execution – Analogy to RELAY RACE.

❑ Third Runner (Navigate to URL):

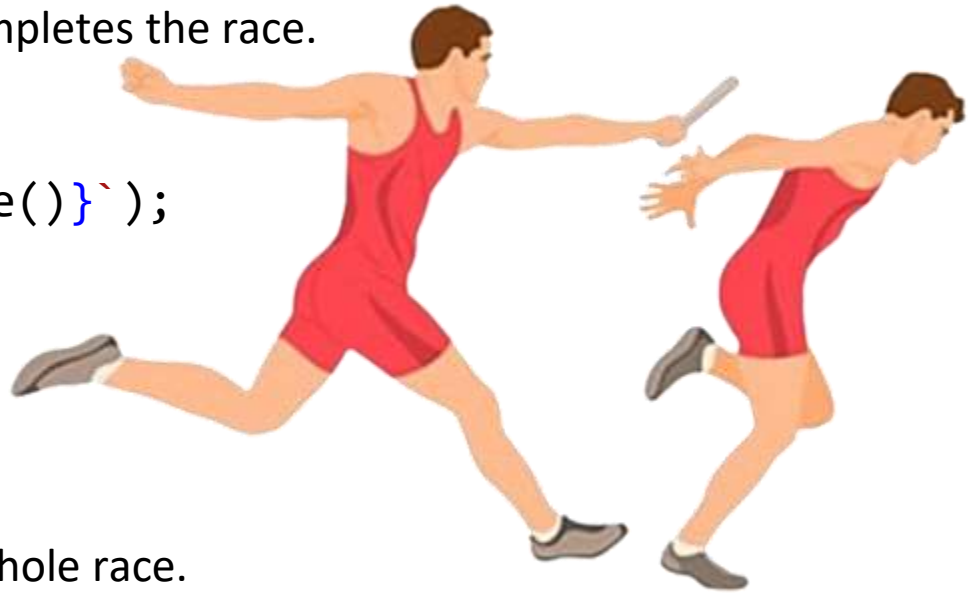
The **third runner** runs after getting the baton. This corresponds to **navigating to the URL**:

```
// Load the url
await page.goto("http://leaftaps.com/opentaps/control/main");
```

❑ Fourth Runner (Get Title & Finish the Race)):

The **final runner** grabs the baton, sprints to the finish line, and the team completes the race.
This is like **fetching the page title** and **waiting for a few seconds**:

```
console.log(`The title of the page is ${ await page.title()}`);
await page.waitForTimeout(5000);
```

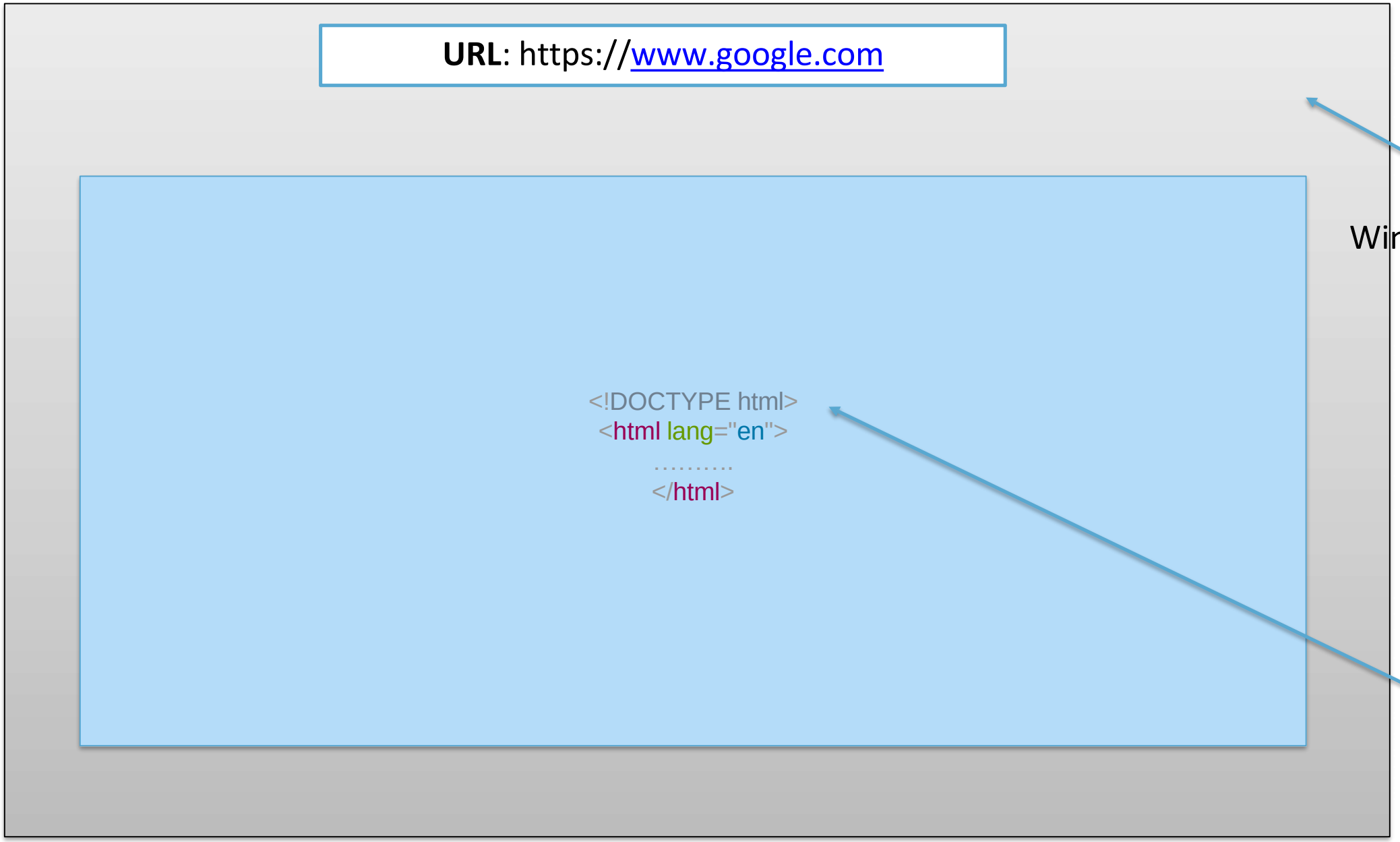


❑ Why `async/await` is Like a Smooth Baton Pass

- In a relay race, if one runner starts too soon or too late, it messes up the whole race.
- In Playwright, if you don't use `await`, the next step might start before the previous one is done, causing errors (like trying to get a page title before the page is even loaded!).

Testleaf
Always Ahead

DOM - Document object model

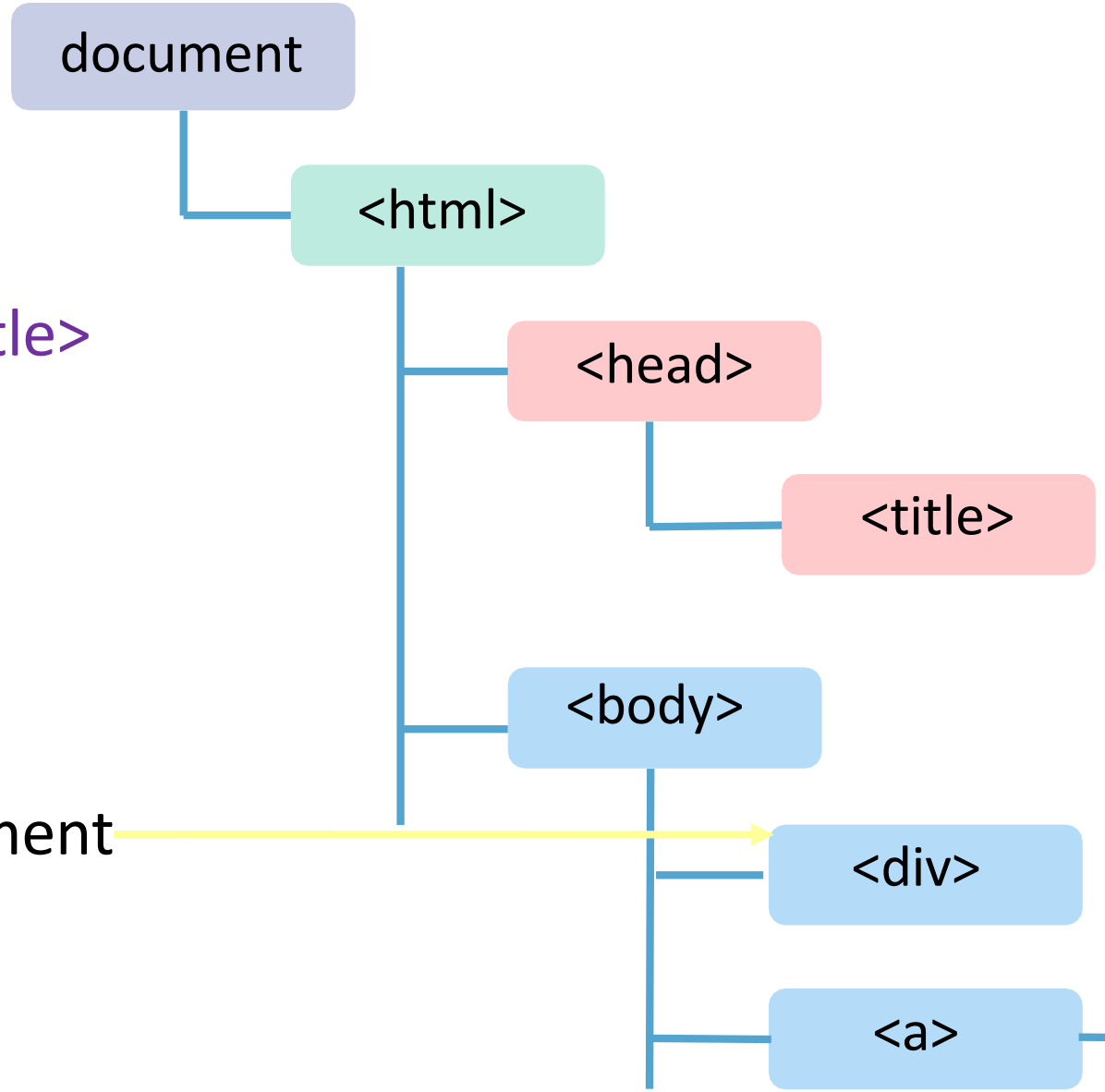


Testleaf
Always Ahead

DOM & WebElement

```
<html>
<head>
   <title> Amazon India </title>
</head>
<body>
<div>
 <span>
   <input>
   </input>
 </span>
</div>
</html>
```

WebElement



Three core components of a webpage

- ## HTML (HyperText Markup Language)

  Defines the structure and content of a webpage.
  Example: Headers, paragraphs, images, and forms.

- ## CSS (Cascading Style Sheets)

  Controls the visual styling of the webpage.
  Example: Fonts, colors, layouts, and animations.

- ## JavaScript

  Makes the webpage dynamic and interactive.
  Example: Form validation, animations, and real-time updates.

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.

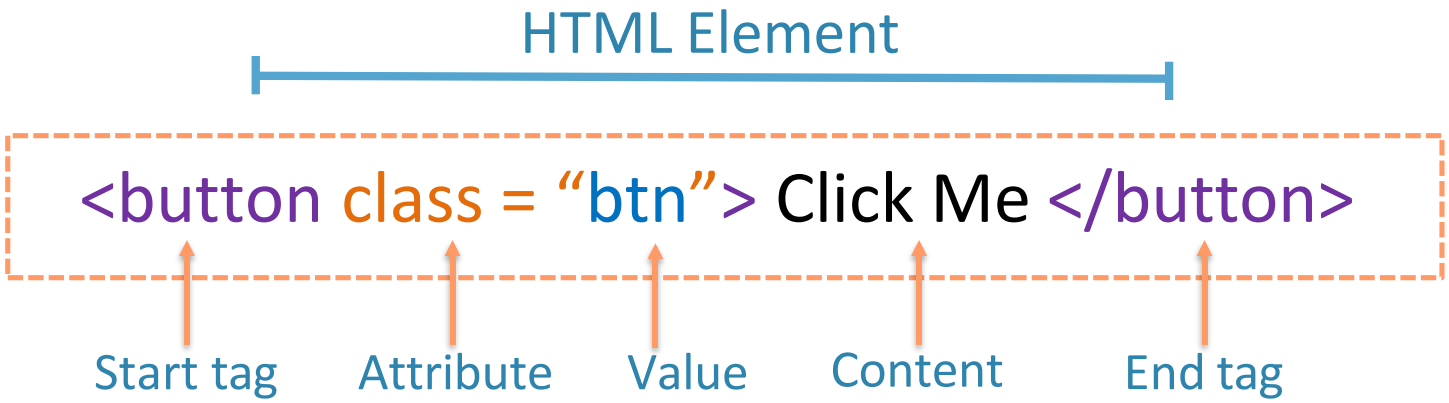
Learn more about locators.

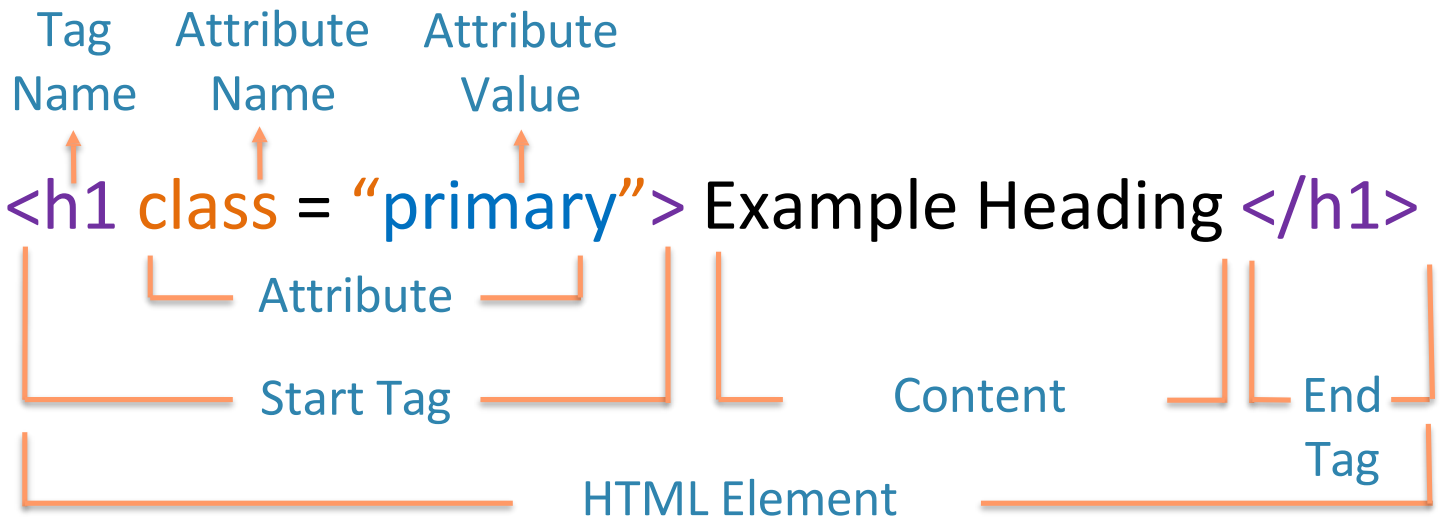
@param `selector` — A selector to use when resolving DOM element. See working with selectors for more details

Agenda – Part 3

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

Let’s understand structure of an element





- Tags
- Attribute Name
- Attribute Value
- Visible Text

Diving into Selector Strategies

- Playwright recommended selectors
- CSS Selectors
- Xpath Locators

Playwright recommended Locator Strategy

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

Playwright recommended Locator Strategy

- getByRole()

To locate a web element by its role (button, link, checkbox, alert)

Syntax :

```
await page.getByRole('button', { name: 'Login' }).click();
```

- getByText()

To locate a web element by the text content.

Syntax:

```
await page.getByText('Username').click();
```

Playwright recommended Locator Strategy

- ## getByLabel()

To locate a web element by the label's text

Syntax :

```
await page.getByLabel('Username').click();
```

- ## getByPlaceholder()

to locate an input by its placeholder value

Syntax:

```
await page.getByPlaceholder('Search Amazon.in').click();
```

Playwright recommended Locator Strategy

- getByAltText()

  To locate a web element by its images/logos (text alternatives)

  Syntax :

  ```
  await page.getByAltText("Playwright logo").isVisible();
  ```

- getByTitle()

To locate a web element by its title attribute

<title>Home | Salesforce</title>

Syntax :

```
await page.getByTitle("Home | Salesforce").isVisible();
```

Playwright recommended Locator Strategy

- getByTestId()

To locate a web element based on its data-testid attribute.

Syntax :

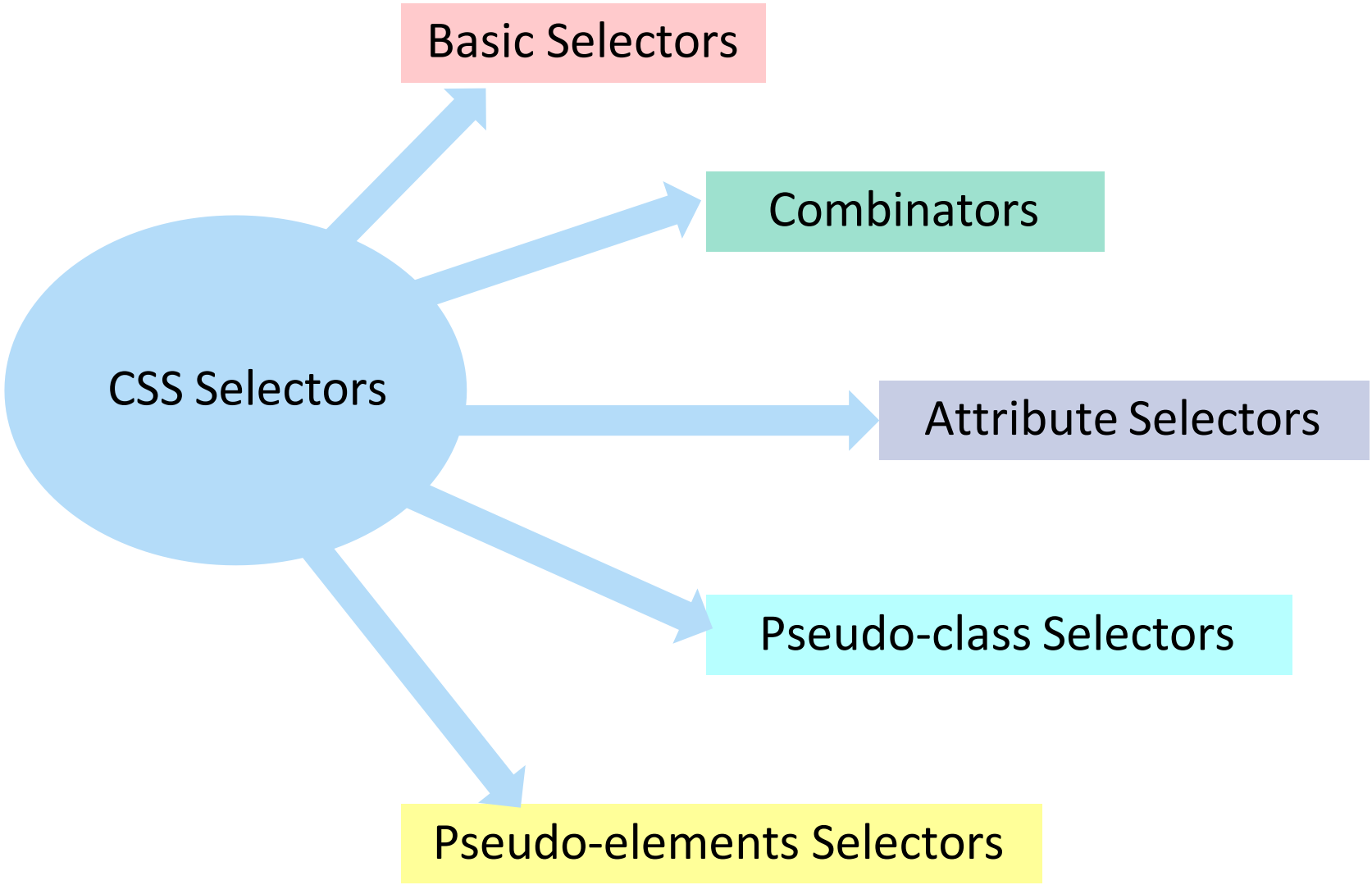
```
page.getByTestId("overlay").click();
```

Sample code with getByMethod()

```
import { test, expect } from '@playwright/test';

test('test', async ({ page }) => {
  await page.goto('http://leaftaps.com/opentaps/control/main');
  await page.getByLabel('Username').click();
  await page.getByLabel('Username').fill('democsr');
  await page.getByLabel('Password').click();
  await page.getByLabel('Password').fill('crmsfa');
  await page.getByText('Username').click();
  await page.getByRole('button', { name: 'Login' }).click();
  await page.getByRole('link', { name: 'CRM/SFA' }).click();
});
```

CSS 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

Playwright recommended Locator Strategy

Type Selector

To locate a web element by its images/logos (text alternatives)

Syntax :

tagname <button>

```
page.locator("button").click();
```

ID - #idValue

To locate a web element by its title attribute

<title>Home | Salesforce</title>

Syntax :

```
await page.getByTitle("Home | Salesforce").isVisible();
```

CSS selectors for different situations

CSS Selectors
1. Type Selector
tagname <button>
page.locator("button").click();

2. Id - #idValue
page.locator("#username");

3. class - .classValue
page.locator(".inputLogin").fill("demosalesmanager");

4. Attribute selector - [attributeName = 'attributeValue']
<input type="text" name="USERNAME" size="50">
page.locator("[name='USERNAME']")
[type='text']
[aria-label = 'Amazon.in']

CSS selectors for different situations

CSS Selectors
1. Type Selector
tagname <button>
page.locator("button").click();

2. Id - #idValue
page.locator("#username");

3. class - .classValue
page.locator(".inputLogin").fill("demosalesmanager");

4. Attribute selector - [attributeName = 'attributeValue']
<input type="text" name="USERNAME" size="50">
page.locator("[name='USERNAME']")
[type='text']
[aria-label = 'Amazon.in']

CSS selectors for different situations

Substring
Starts with attribute selector
Select an elememt with an attribute that starts with a specific value
tagname[attribute^='value']

div[class^='inventory']
page.locator("div[class^='inventory']")
id='submit_button_743'

CSS selectors for different situations

Ends with attribute selector
Select an element with an attribute that ends with a specific value
tagname[attribute$='value']
<button class="btn_primary btn_inventory">ADD TO CART</button>
<button class="btn_secondary btn_inventory">ADD TO CART</button>
<button class="btn btn_inventory">ADD TO CART</button>
page.locator("button[class$='btn_inventory']")

Contains attribute selector
Select an element with an attribute containing a specific substring
tagname[attribute*='value']
button[class*='btn']

CSS selectors for different situations

5. Descendant Combinator
ancestor descendat
form input

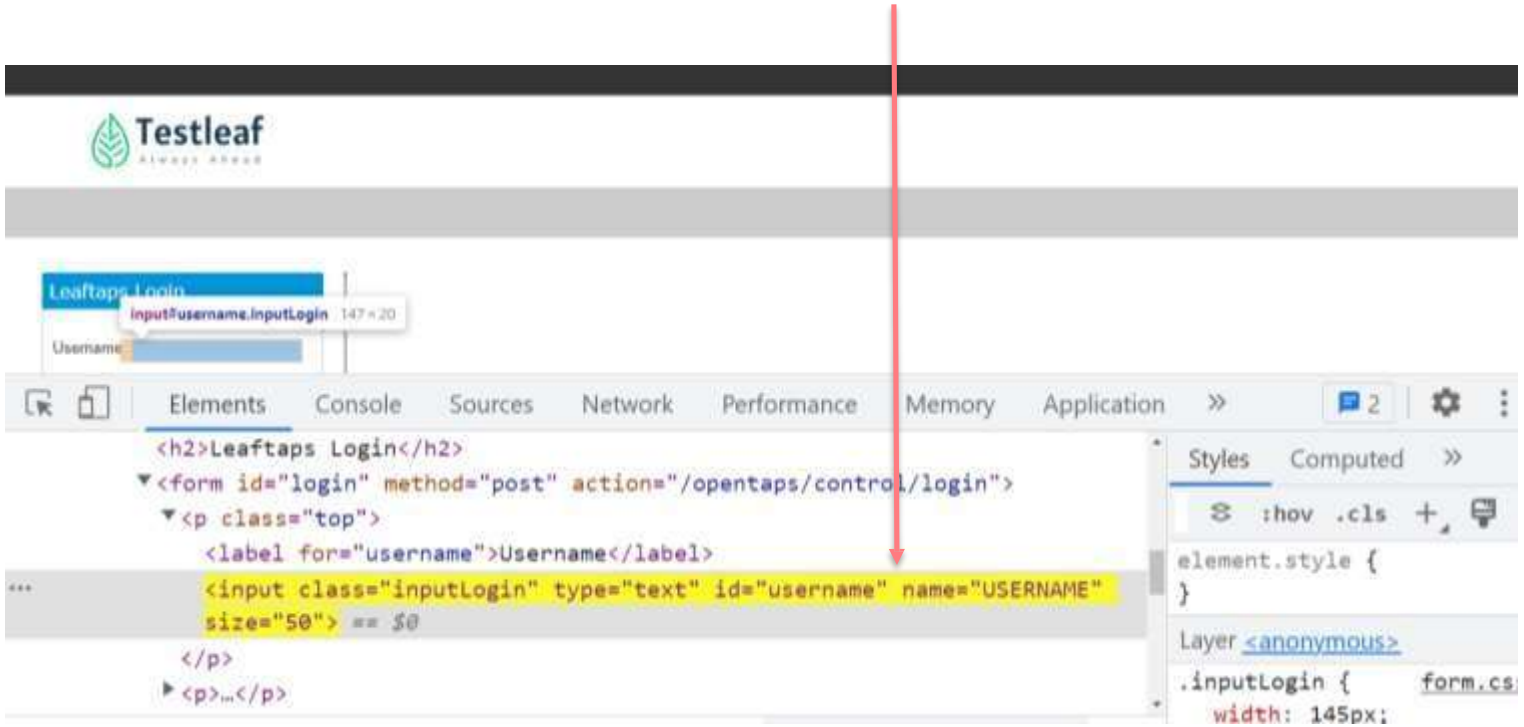
6. Child Combinator
parent>child
p>input

7. Adjacent sibling combinator
element+adjacent

8. General sibling combintor
element~sibling

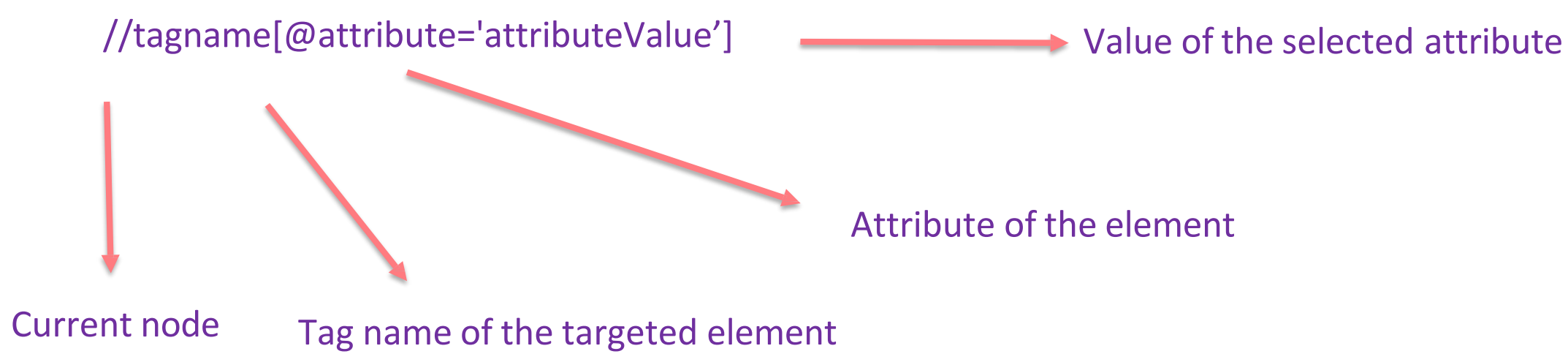
Absolute XPath

- 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:



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.,

Let’s learn few actions & add a new Test

Practice a simple program by creating a lead in the Leaftap application and understand few actions:

- Typing
- Clicking
- Navigating

Playwright *config* file walkthrough

testDir	Contains dir path where all test cases resides
fullyParallel	Decides if to run all files in parallel or all. Files and test cases inside them in parallel
Test timeout	Max time allowed for a test case to run
retries	Number of time a test case to retry on failure
workers	Number of workers to split the test cases on
trace	On what conditions, we should keep the trace
screenshots	On what conditions, we should keep the screenshot
video	On what conditions, we should keep the video
Expect timeout	Max time allowed for assertions we are doing using expect

Auto Wait

Playwright when performing any relevant action first ensures that

- All applied checks for that actions are passed
- Within a set timeout which can be overridden by
- If the criteria is not full filled , it will raise ***TimeoutError***

page.click() ensures

- element is attached
- element is visible
- element is stable
- element is receiving events
- element is enabled