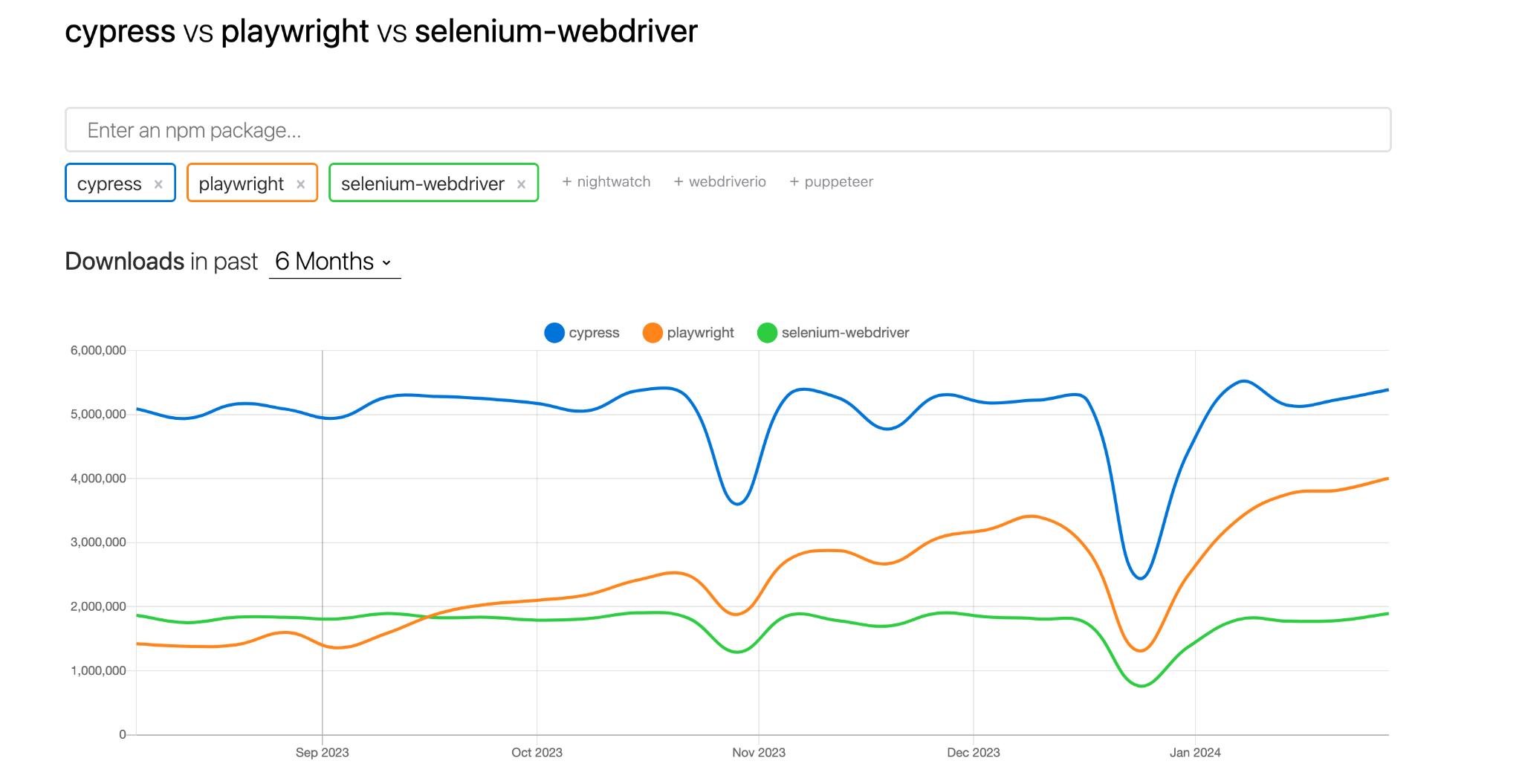
## **Playwright JS - TheTestingAcademy**

## 🟩 **What is a Playwright?**

1. Playwright was created for end-to-end testing with reliable **auto-wait capability** for modern web apps.
2. Playwright supports all modern rendering engines. including Chromium, WebKit, and Firefox.  
   **Explain** - [What's the difference between a browser engine and rendering engine?](https://stackoverflow.com/questions/46169376/whats-the-difference-between-a-browser-engine-and-rendering-engine)
3. You can test on Windows, Linux, and macOS, locally or on CI, headless or headed with native mobile emulation.
4. **Traces** which can take automatic Screenshots, test video recording, Flaky test retry & Logging mechanism.
5. PW provides Context isolation, running it on multiple browser configurations out of the box.
6. **Inspector tool** helps monitor and debug execution, check click points, and verify page locators at runtime.
7. Codegen tool which generates test code by recording your actions. Save them into any language.

[cypress vs playwright vs selenium-webdriver | npm trends](https://npmtrends.com/cypress-vs-playwright-vs-selenium-webdriver)



## 🟩 **Feature of Playwright**

* Support for all browsers: Chrome, Edge, Firefox and Safari (on both Mac and Windows!)
* Emulate different devices, Ex. web apps and mobile web browsers
* Ability to reuse browser storage, Ex. test with reusable "logged-in" session
* Browser contexts. Reuse a single browser instance for multiple
* Tests and use different browser contexts in one test
* Test simultaneously in multiple domains and pages.
* Headed and headless mode, auto-waits, network request mocking
* Text/CSS/XPath/layout/nth-match selectors, ability to chain selectors
* Support for file downloads, uploads and iframes , You can specify geolocation, permissions and even dark mode.
* Playwright is free! Developed and open-sourced by Microsoft.

## 🟩 **Install Playwright with VS Code**

Sure, let's break down the step-by-step instructions for installing Node.js, Playwright, Visual Studio Code, and adding Node.js to the environment variables for Windows, macOS, and Linux.

### **Install Node.js**

**Windows:**

1. Download the Windows installer from the [Node.js website](https://nodejs.org/).

2. Run the installer.

3. Follow the prompts in the installer (Accept the license agreement, click the NEXT button a bunch of times and finally install).

4. Restart your computer to ensure changes take effect.

**macOS**:

1. Download the macOS installer from the [Node.js website](https://nodejs.org/).

2. Open the `.pkg` file you downloaded and follow the instructions.

3. You may need to enter your admin password.

4. To verify installation, open a terminal and type `node -v`.

**Linux**

1. Use a package manager to install Node.js.

- For Ubuntu/Debian, use `sudo apt-get install nodejs npm`.

- For Fedora, use `sudo dnf install nodejs`.

- For other distributions, check the package manager or Node.js website.

2. Verify the installation with `node -v` and `npm -v`.

### **Install Playwright**

1. Ensure Node.js and npm (Node Package Manager) are installed.

2. Open your terminal or command prompt.

3. Run `npm init -y` to create a `package.json` file if you don't have one.

4. Install Playwright by running `npm install playwright`.

5. This command installs Playwright and its browser binaries.

### **Install VS Code - Visual Studio Code (**

Windows, macOS, and Linux:

1. Go to the [VS Code website](https://code.visualstudio.com/) and download the installer for your OS.

2. Run the installer.

3. Follow the installation instructions.

4. Launch VS Code after installation.

**Adding Node.js to the Environment Variables**

**Windows:**

1. Search for "Environment Variables" in the Start menu and open "Edit the system environment variables."

2. Click on "Environment Variables."

3. Under "System Variables," find the "Path" variable and click "Edit."

4. Click "New" and add the path to your Node.js installation (usually `C:\Program Files\nodejs`).

5. Click OK to save your changes.

**macOS and Linux:**

1. Open your terminal.

2. Edit your shell profile file (.bash\_profile, .bashrc, .zshrc, etc.).

- For bash, use `nano ~/.bash\_profile` or `nano ~/.bashrc`.

- For zsh, use `nano ~/.zshrc`.

3. Add `export PATH="/usr/local/share/npm/bin:$PATH"` to your profile (adjust the path if your Node.js installation is different).

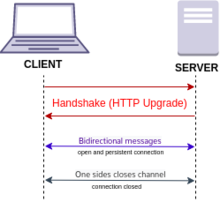
4. Save the file and reload your profile with `source ~/.bash\_profile`, `source ~/.bashrc`, or `source ~/.zshrc`.

## 🟩 **Playwright Architecture**

* Playwright is a Node.js library for browser automation and testing. The core is written in TypeScript.
* It uses a client/server architecture. The Playwright library in Node.js sends commands to a BrowserServer process.
* BrowserServer launches and controls actual browser instances like Chromium, Firefox or WebKit via the Chrome DevTools Protocol.
* BrowserContexts isolate state like cookies, local storage between different contexts. Pages with frames run inside a browser context.
* Under the hood, **the Playwright lib translates API calls into DevTools protocol commands that BrowserServer executes.**
* Results, events and browser logs are propagated back to the Playwright library via WebSocket.

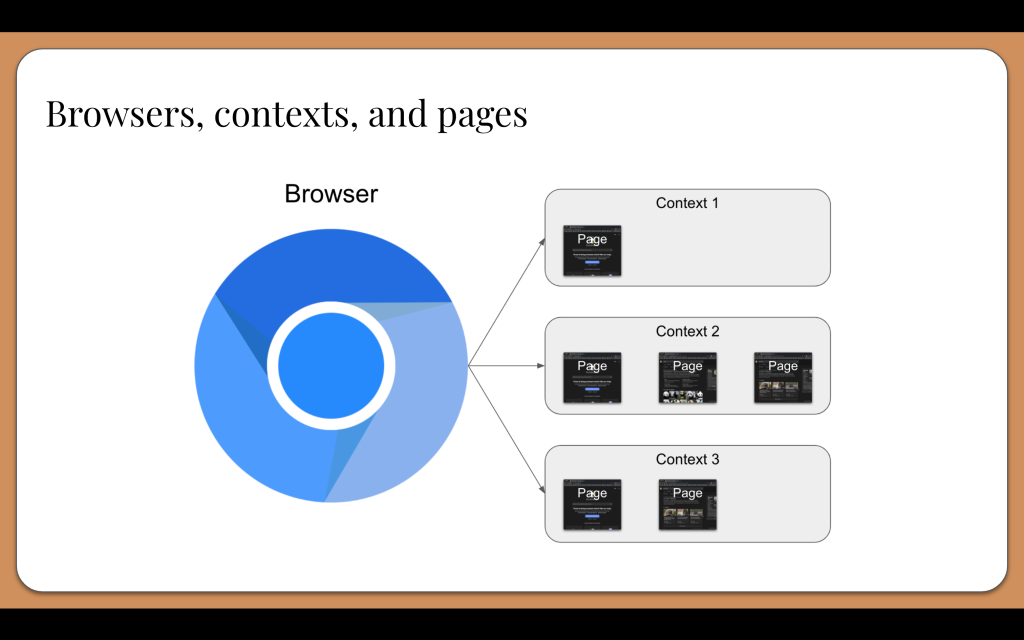
What is Websocket?

* WebSockets provide bidirectional, full-duplex communication between a client and server over a single TCP connection.
* This allows for real-time data transfer between client and server unlike HTTP where the client must request data from the server.
* With WebSockets, the client or server can push data to each other at any time after the initial handshake and connection setup.
* This makes WebSockets well-suited for real-time web applications like chat, notifications, collaborations and multiplayer games.
* The WebSocket protocol upgrades from HTTP but then operates over TCP rather than HTTP request/response.



Overall :

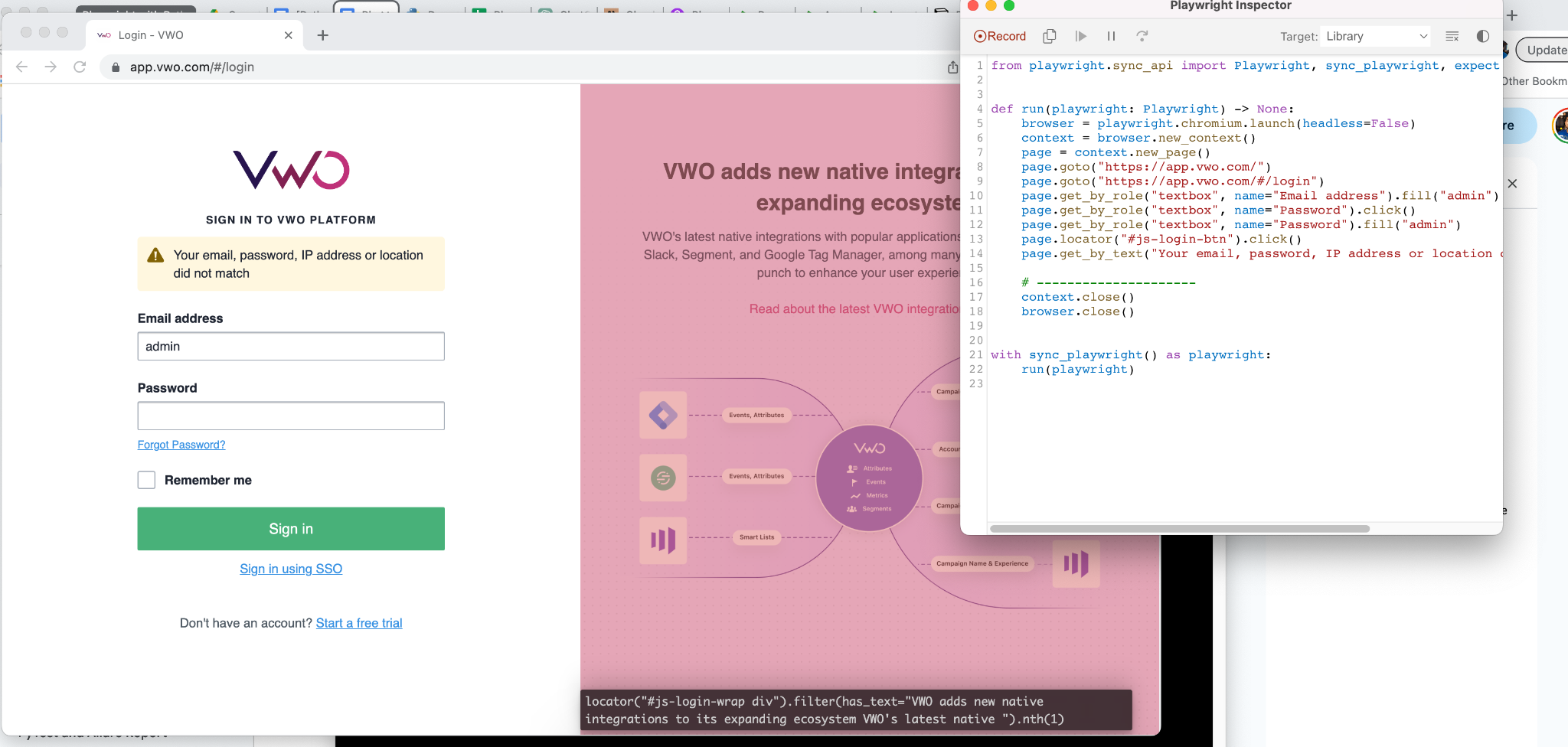
* Playwright lib (Node.js) => BrowserServer => Browser Drivers + Actual Browsers
* Runs browsers headless or headful
* Isolates state with BrowserContexts
* API commands convert to DevTools protocol calls
* Results returned back to Playwright over WebSocket



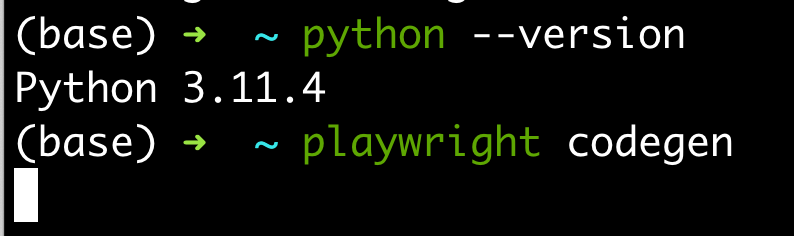
## 

Ref - <https://www.programsbuzz.com/article/playwright-architecture>

## 🟩 **Test Generators & Record and Play**



* Playwright's test generation capabilities with record and playback.
* **playwright codegen** that launches browsers to record interactions and generate Playwright tests.



* Use the --navigation flag to auto-insert navigation like clicks and type actions. Omit this to only insert assertions for element state.
* Record your interactions across various pages and scenarios to generate a robust test suite.
* The generated tests will use Page Object Model with a base Page class and subclasses for each page.
* Tests will use async/await and have descriptive names based on your actions. Tests are fully runnable.
* Review and parameterize the locators, replace fixed values with variables, extract common steps into reusable functions.
* Add any missing assertions or logic that may not have been captured during recording.
* Use playwright test to run the generated tests and integrate them into your CI/CD pipelines.

## 🟩 **Write Playwright Test**

* Verify the Title of the Page.
* Verify the Heading of the Page.
* Click on a button
* Explain - Test Isolation (via Page)
* Test Hooks

| <Code here> |
| --- |

## 

## 🟩 **Locators in Python**

A mechanism used to locate and interact with web elements on a web page

### HTML Explained

* HTML is the standard markup language for Web pages.
* <tagname>Content goes here...</tagname>

| <!DOCTYPE html> <html> <head> <title>Page Title</title> </head> <body>  <h1>This is a Heading</h1> <p>This is a paragraph.</p>  </body> </html> |
| --- |



### 

A locator is a way of identifying an element on a web page so that it can be interacted with.

There are several different types of locators that can be used, including:

* ID: This locator type uses the unique ID attribute of an element to locate it on the page.
* Name: This locator type uses the name attribute of an element to locate it on the page.
* Class name: This locator type uses the class attribute of an element to locate it on the page.
* Tag name: This locator type uses the HTML tag name of an element to locate it on the page.
* **CSS selector**: This locator type uses a CSS selector to locate an element on the page.
* **Xpath:** This locator type uses an XPath expression to locate an element on the page.
* When writing test scripts with Selenium, you can use a combination of these locator types to accurately and reliably locate elements on the page.

It uses "locators" to identify and manipulate elements on a web page. There are several types of locators that can be used in Selenium, including:

<a id="btn-make-appointment" href="./index.php#appointment" class="btn btn-dark btn-lg">Make Appointment</a>

### 

### Locators and HTML Forms



Tag

Attribute = Value

**<input data-qa="hocewoqisi" type="email" class="text-input W(100%)" name="username" id="login-username" >**

data-qa="hocewoqisi"

type="email"

class="text-input W(100%)"

name="username"

id="login-username"

### 

### 

### **HTML Forms**

What is an HTML Form?

A HTML form is a section of a web page that contains form elements, such as text fields, checkboxes, and buttons. Forms allow users to enter data and interact with a website.

Forms are created using the <form> HTML tag. This tag defines the start and end of a form, and it can have several attributes that determine how the form behaves.

For example, the action attribute specifies the URL of the server-side script that will process the form data, and the method attribute specifies whether the form data will be sent to the server using the GET or POST method.

| <form action="http://www.example.com/form-handler.php" method="POST">  <label for="username">Username:</label>  <input type="text" id="username" name="username">  <br>  <label for="password">Password:</label>  <input type="password" id="password" name="password">  <br>  <input type="submit" value="Log In"> </form> |
| --- |

## 🟩 **Mastering XPath**

What is XPath?

XPath is a query language for selecting nodes from an XML/HTML document.

XPath was defined by the World Wide Web Consortium

Core Logic - **//tagName[@attribute='value']**

| **page.locator("css=button").click() page.locator("xpath=//button").click()** |
| --- |

**TAG** - h1, p, input, a, form, img, video, audio,button, table, ul, li, tr, div, select, span, -> Html Tags

Attrbute - id, class, name, alt, href, src, data-qa, ….srcset ..

* **Relative XPath**
* Absolute XPath
* XPath Functions
* Xpath Axes

**Absolute XPath**

/html/body/div[2]/div[1]/div[2]/div/div[1]/div/div/div[3]/form[1]/ul/li[1]/div/input

Why do we need to MASTER Locators?

Probably the first question asked by the interviewer.

You should always find small and efficient Locators.

UI Automation is all about finding locators.

Don't use tools at first.



##### Absolute XPath

Complete path from the Root Element.

If any element is added or deleted, Xpath fails.

/html/body/div[2]/div[1]/div[2]/div/div[1]/div/div/div[3]/form[1]/ul/li[1]/div/input

##### Relative Xpath

You can simply start by referencing the element you want and go from there

Based on searching an element in DOM. //\*[@id="login-username"]

* //a[@id="btn-make-appointment"]/..
* //a[contains(@id,"make")]
* //a[starts-with(@id,"btn")]
* //a[text()="Make Appointment"]
* //a[text()="Make Appointment" or @id="btn-make-appointment"]
* //a[text()="Make Appointment" and @id="btn-make-appointment"]

//input[@id="login-username"]

Xpath -> //input[@id="txt-username"]

Css - > #txt-username

##### XPath Functions

Contains()

//tag\_name[contains(@attribute,'value\_of\_attribute')]

Starts-with()

//tag\_name[starts-with(@attribute,'Part\_of\_Attribute\_value')]

Text()

//tag\_name[text()='Text of the element']

**String functions**

concat(string, ...): XPath concat function concatenated number of arguments and return to a concatenated string.

starts-with(string, string): XPath start-with function return True/False. Return True if second argument string is start with first argument.

contains(string, string) - XPath contains function return True/False. Return True if second argument string is a contain of first argument.

string-length(string): XPath string-length function return the length of string.

substring-after(string, string): XPath substring-after function return the substring of the first argument string base on first occurrence of the second argument string after all character.

substring-before(string, string): XPath substring-before function return the substring of the first argument string base on first occurrence of the second argument string before all character.

normalize-space(string): XPath normalize-space function sequence of whitespace combine into single normalize space and removing leading and trailing whitespace.

##### Operators - AND & OR

And Example

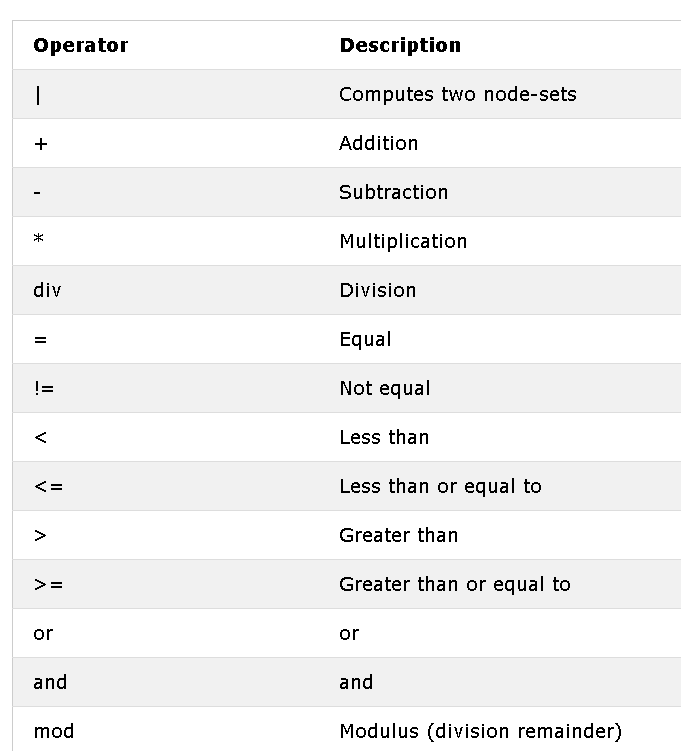
//tag\_name[@name = 'Name value' and @id = ‘ID value’]

<https://katalon-demo-cura.herokuapp.com/>

//a[text()="Make Appointment" and contains(@id,"btn-make-appointment")]

OR Example

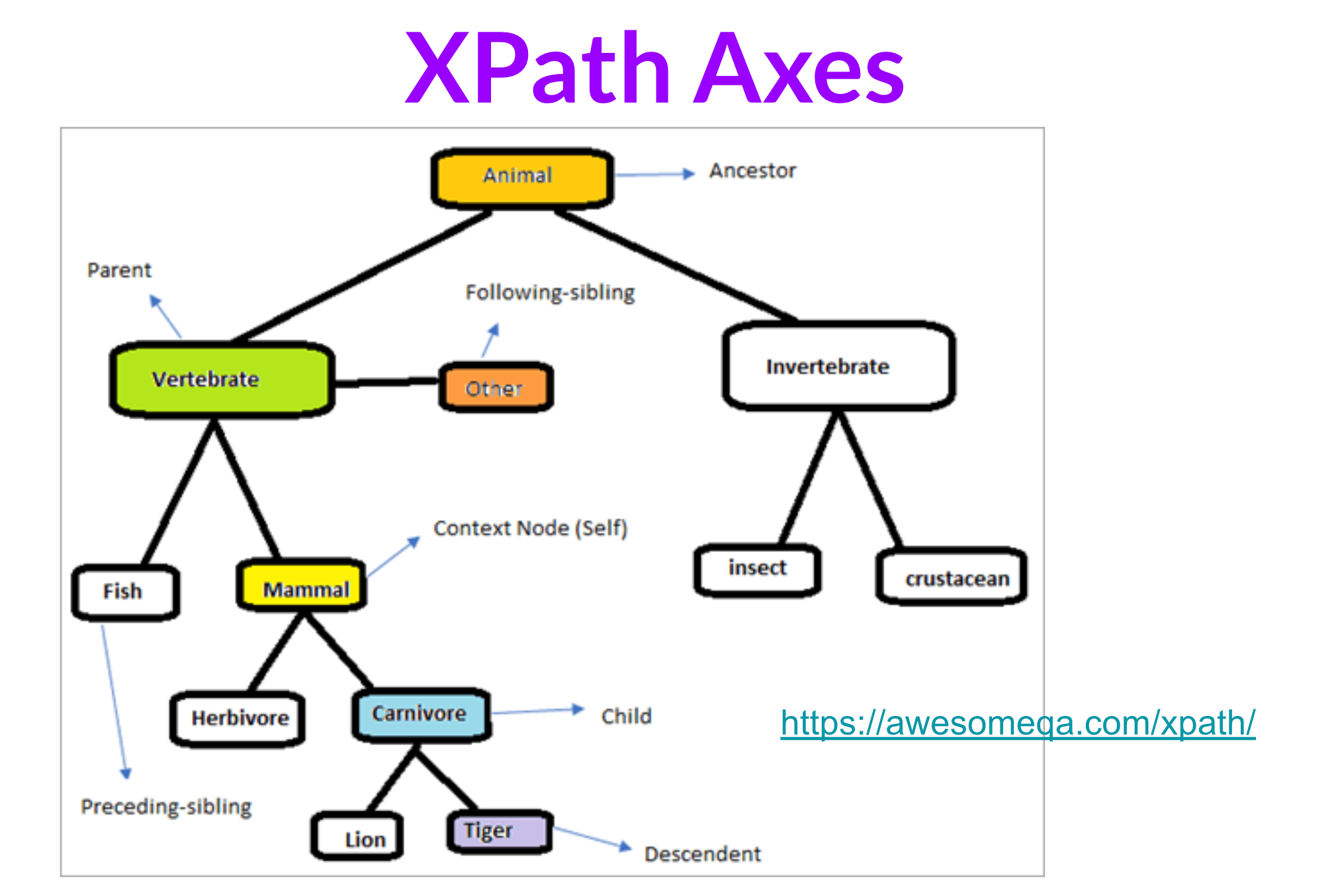
//input[@placeholder ='Full Name' or @type = 'text']

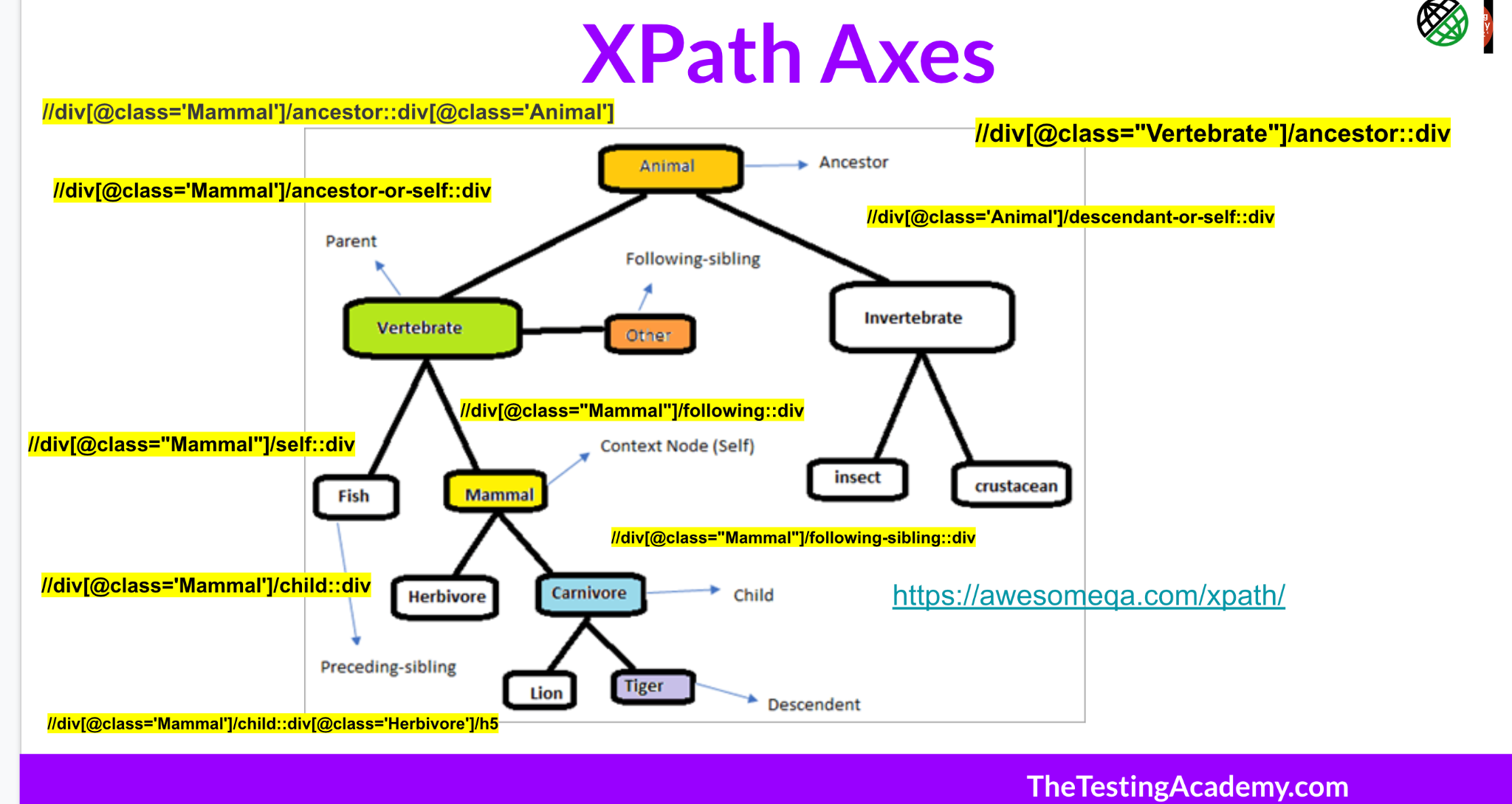


##### XPath Axes

In the XML documents, we have relationships between various nodes to locate those nodes in the DOM structure.

* Ancestor
* Child, parent
* Descendant
* Following, following-sibling
* Self.





<https://www.softwaretestinghelp.com/xpath-axes-tutorial/>

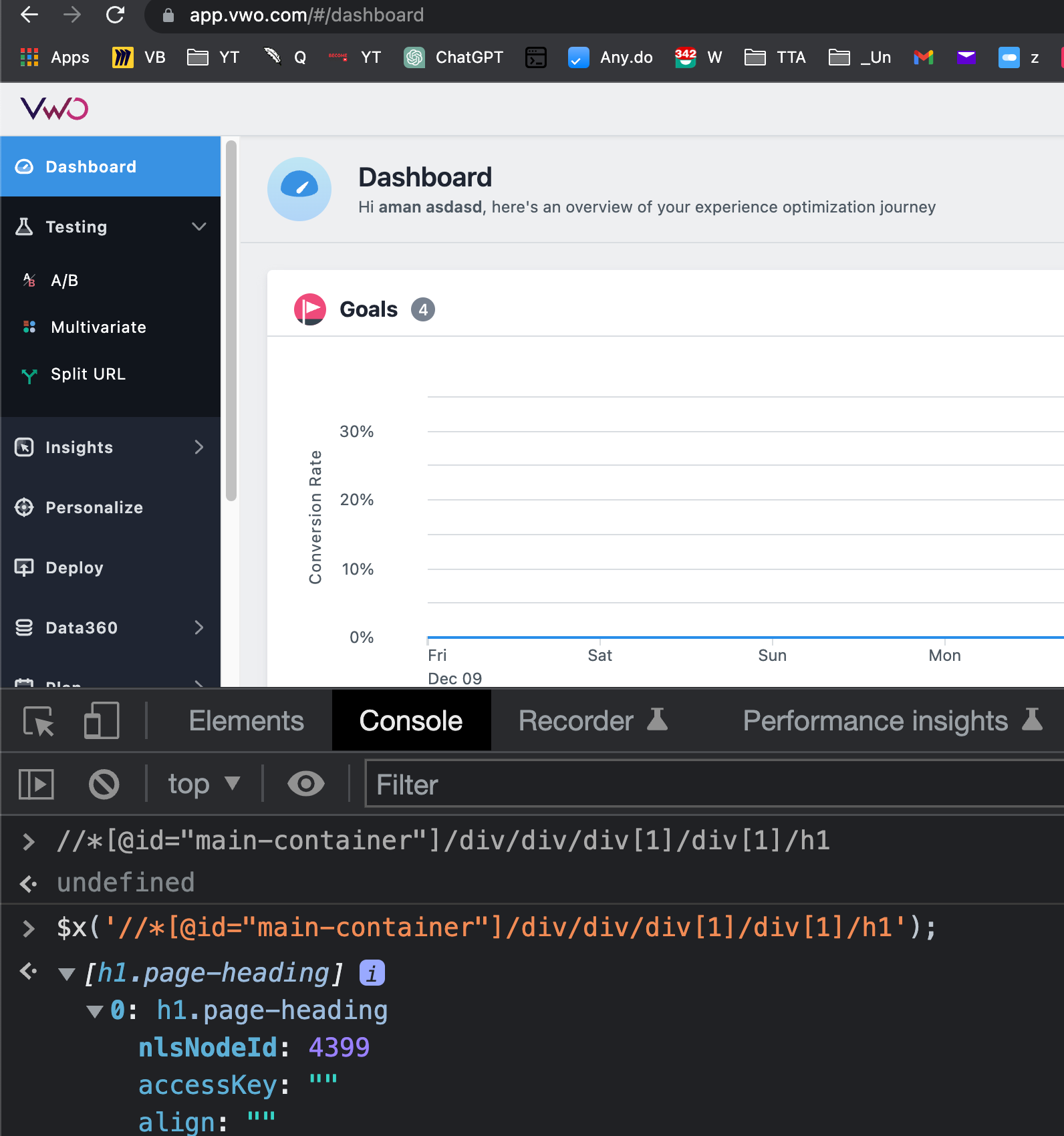
//span[text()='Invalid Email']/ancestor::div

//\*[@id="main-page"]/div[1]/child::div

//\*[@id="js-main-container-wrap"]/child::div

//\*[@id="js-main-container-wrap"]/following::div

<https://devhints.io/xpath>



#### 

#### 

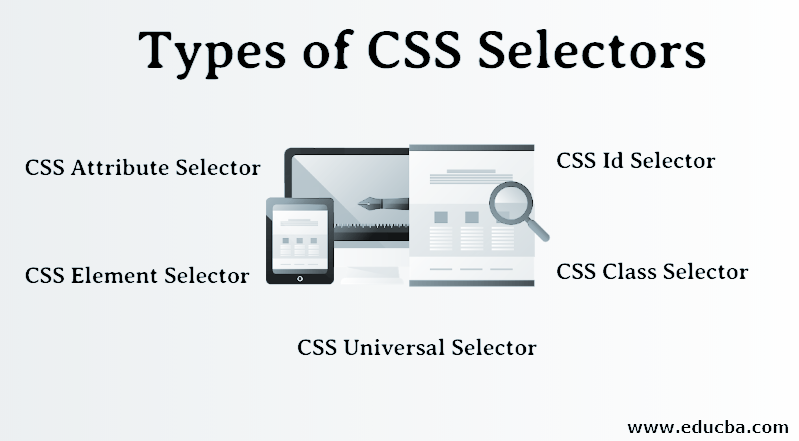
#### 

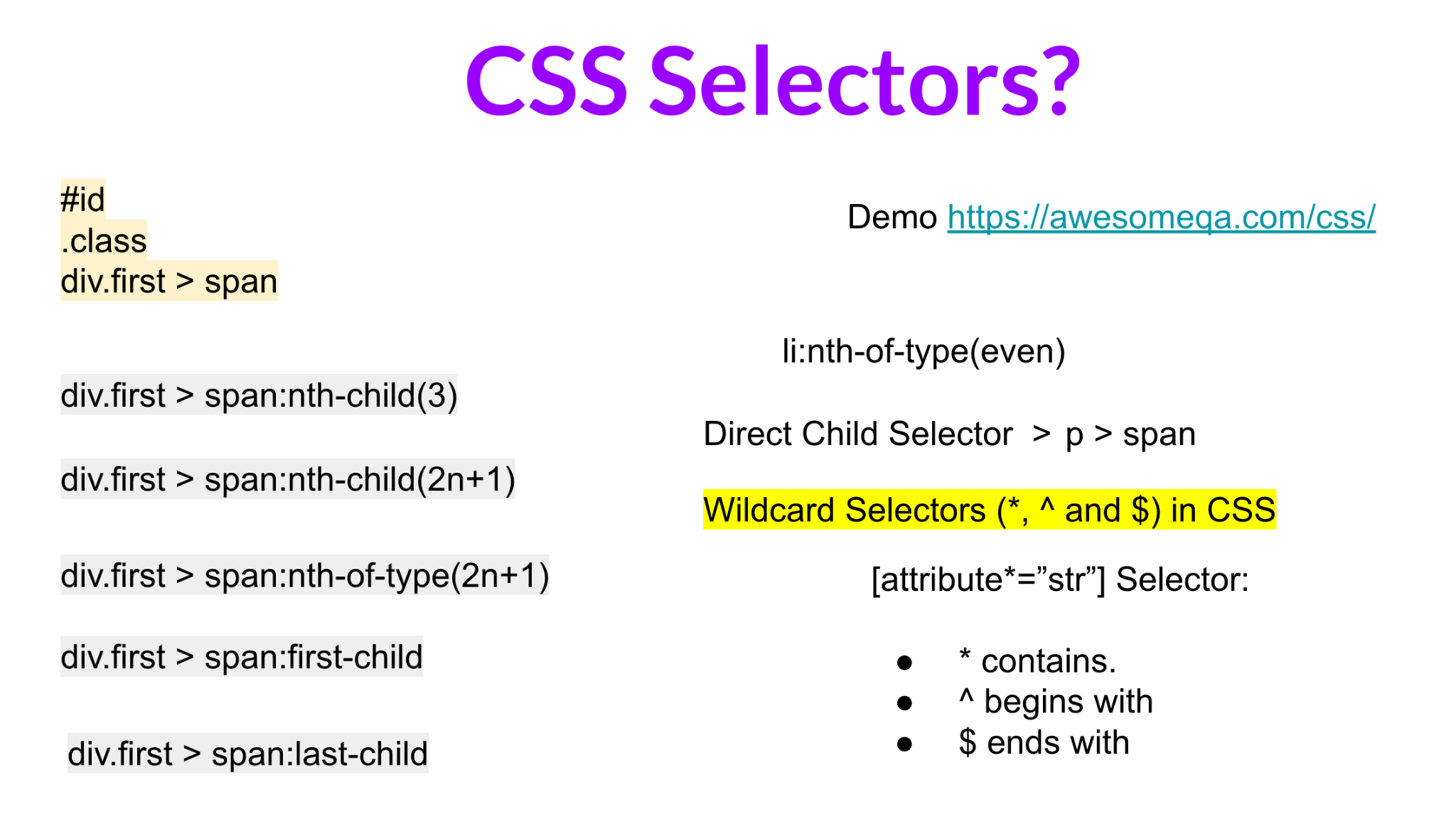
#### 

#### 

## 🟩 Master CSS Selectors

CSS selectors are used to select elements in an HTML or XML document in order to apply styles or other manipulations to those elements.





Css Selector - a#btn-make-appointment, a[id$="appointment"]

Xpath -> //a[@id=”btn-make-appointment”]

# ID > name -> className -> Css SELECTOR -> Xpath (70%)

CSS selectors allow you to select elements based on their tag name, id, class, attribute, and other characteristics.

* To select all elements with the tag "p" (paragraph), you could use the following selector: p
* To select an element with the ID "main-heading", you could use the following selector: #main-heading
* To select all elements with the class "error", you could use the following selector: .error
* To select all elements with the attribute "disabled", you could use the following selector: [disabled]
* To select all "a" elements that are descendants of a "nav" element, you could use the following selector: nav a

form#login-form input[type="radio"]

**CSS [attribute\*=value] Selector**

The [attribute\*=”str”] selector is used to select those elements whose attribute value contains the specified substring str.

**CSS [attribute=value] Selector**

The [attribute=value] selector in CSS is used to select those elements whose attribute value is equal to “value”.

**CSS [attribute$=value] Selector** The [attribute$=”value”] selector is used to select those elements whose attribute value ends with a specified value “value”.

**CSS [attribute|=value] Selector** This is used to select those elements whose attribute value is equal to “value” or whose attribute value started with “value” immediately followed by hyphen (-).

**CSS [attribute~=value] Selector** The [attribute~=”value”] selector is used to select those elements whose attribute value contains a specified word.

**CSS [attribute^=value] Selector** The [attribute^=value] selector is used to select those elements whose attribute value begins with given attribute.

**CSS :first-child Selector** The :first-child selector is used to select those elements which are the first-child elements.

**CSS :last-child Selector** The :last-child Selector is used to target the last child element of it’s parent for styling.

**CSS :nth-child() Selector** The :nth-child() CSS pseudo-class selector is used to match the elements based on their position in a group of siblings.

**CSS :nth-of-type() Selector** The :nth-of-type() in css Selector is used to style only those elements which are the nth number of child of its parent element.

## 🟩 Auto Waiting Concept

🟩 Upload & Download files

## 🟩 Checkbox, Input box, Radio box

## 🟩 Handle Shadow DOM

## 🟩 Handle SVG

## 🟩 Handle Alert & Popups

## ✅ Handle Windows,Frames

## **✅** Handle Javascript

## ✅ Handle WebTables

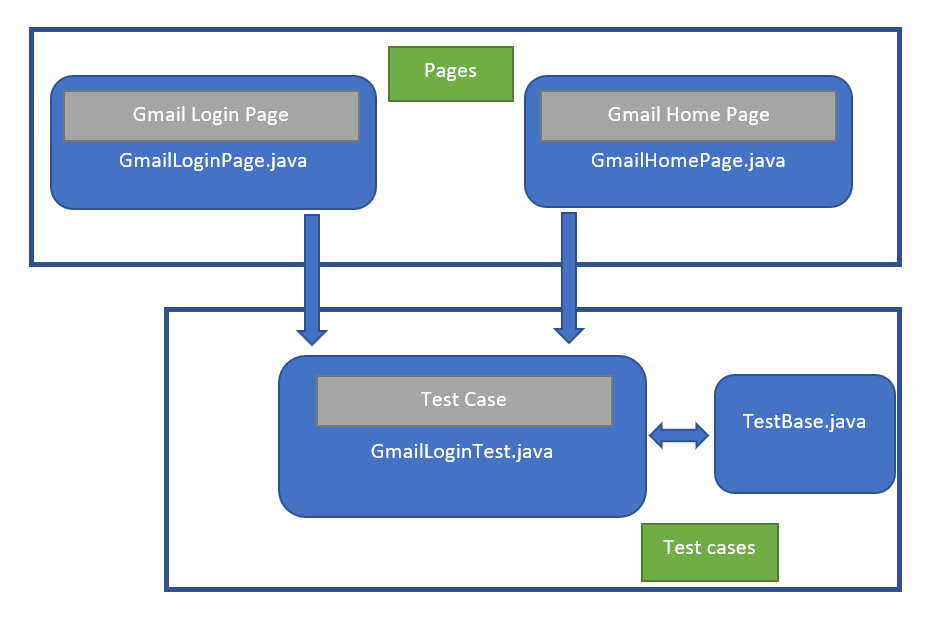
## ✅ Taking ScreenShots (+Failure)

## **✅** Page Object Model

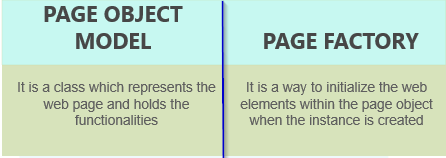
What is a Page Object Model in Selenium?

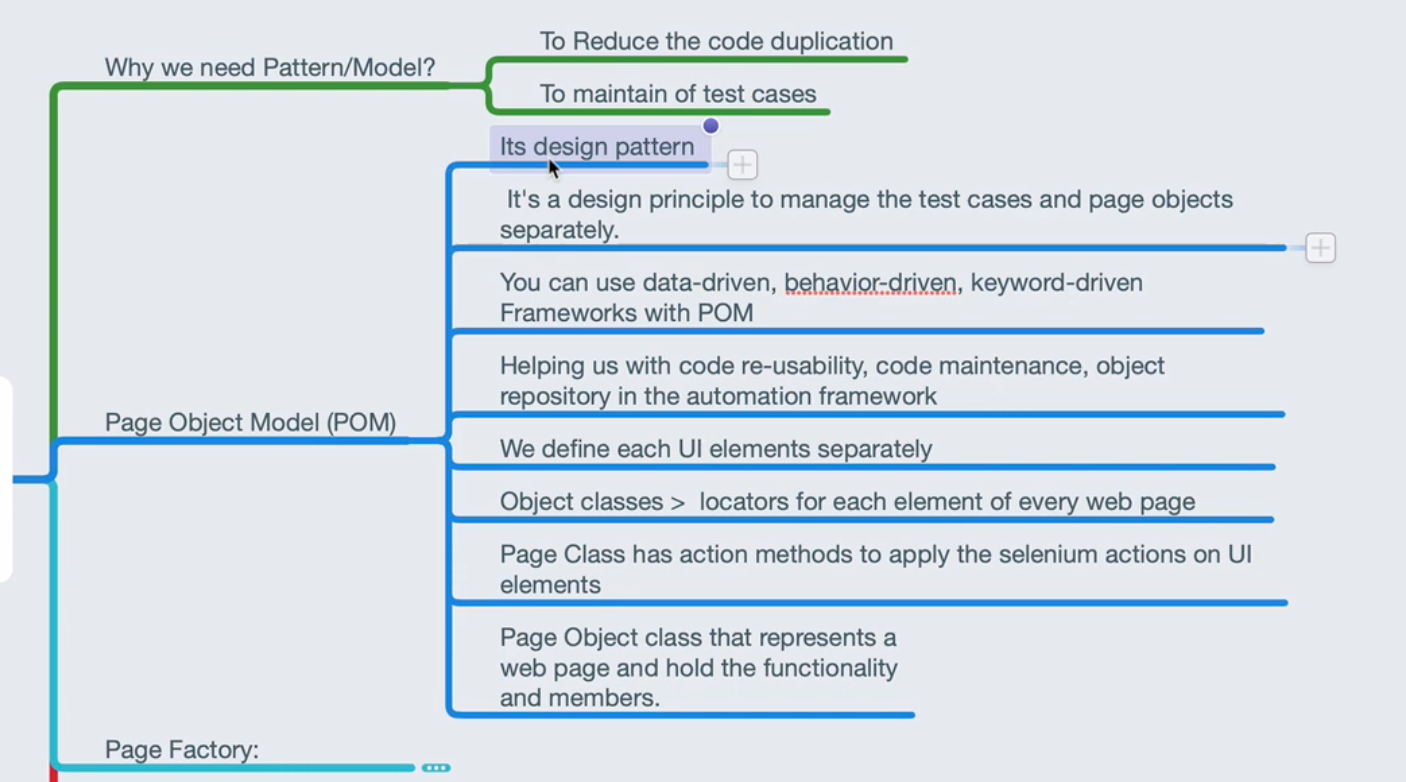
is a design pattern in Selenium that creates an object repository for storing all web elements. It helps reduce code duplication and improves test case maintenance.

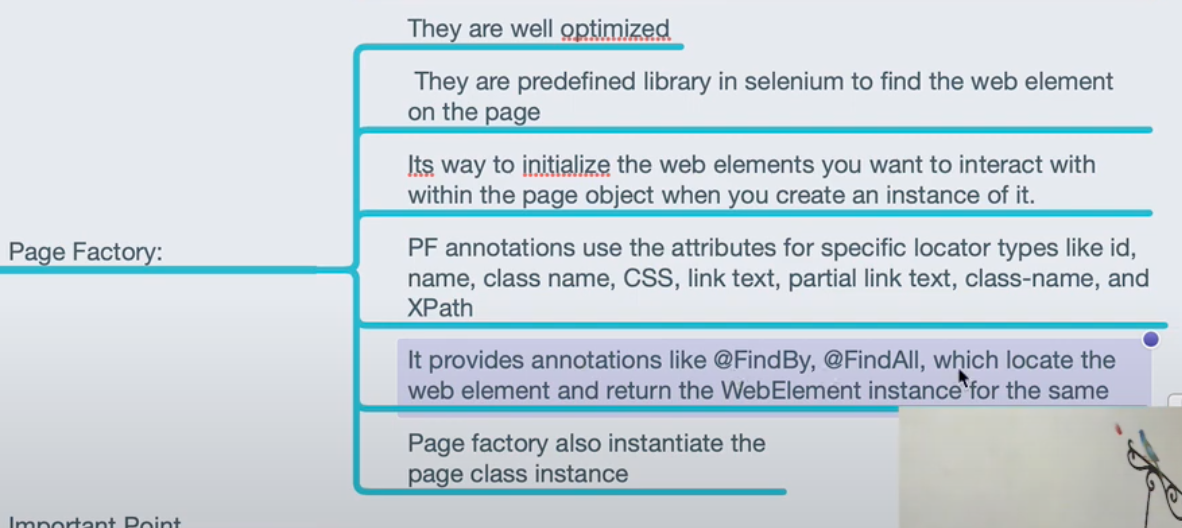
<https://playwright.dev/python/docs/pom>



* Page Object Model in Selenium WebDriver is an Object Repository design pattern.
* Selenium page object model creates our testing code maintainable, reusable.
* Page Factory is an optimized way to create an object repository in the Page Object Model framework concept.
* AjaxElementLocatorFactory is a lazy load concept in Page Factory – page object design pattern to identify WebElements only when they are used in any operation.







#### POM vs Page Factory

[What is the Main difference between Page Object Model and Page Factory in Selenium? (Download Code)](https://www.youtube.com/watch?v=EcjDrADDEfw)

## 

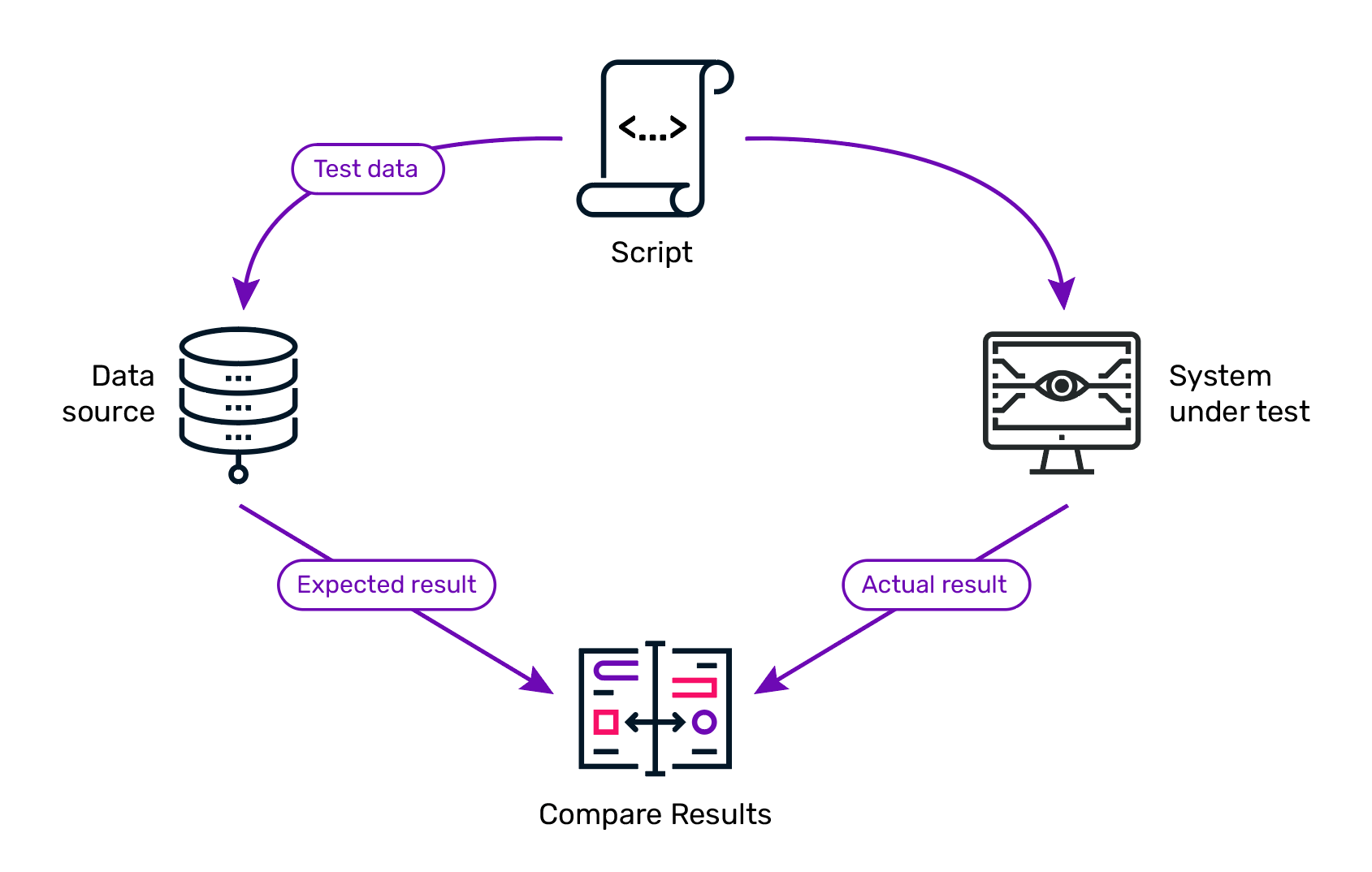
| class SearchPage:  def \_\_init\_\_(self, page):  self.page = page  self.search\_term\_input = page.locator('[aria-label="Enter your search term"]')   def navigate(self):  self.page.goto("https://bing.com")   def search(self, text):  self.search\_term\_input.fill(text)  self.search\_term\_input.press("Enter") |
| --- |

| from models.search import SearchPage  # in the test page = browser.new\_page() search\_page = SearchPage(page) search\_page.navigate() search\_page.search("search query") |
| --- |

## 

## ✅ Data Driven Testing

* Data-driven testing (DDT) with Python and Playwright involves using these tools to automate testing of web applications by feeding them a set of test data.
* The goal is to validate the application's behavior under various conditions.
* Test Data (CSV, Excel) -> Test Scripts -> Verify Expected Result with Actual Result.



##### **Workflow for Data-Driven Testing (DDT):**

**Prepare Test Data**: The first step is to prepare a set of test data. This data can be in various formats, such as CSV, Excel, JSON, or databases. The test data should represent different scenarios that the application might encounter.

**Create Test Scripts:** Using Python, you write test scripts that utilize Playwright to interact with the web application. These scripts are designed to accept data inputs and use them to drive the browser interactions.

**Integrate Test Data with Scripts:** The test scripts are modified to read the test data. This can be achieved through Python's file handling capabilities or database connectivity. The script reads each set of data and feeds it into the test cases.

**Execute Tests**: The Python script, integrated with Playwright, is executed. For each set of test data, the script runs the test case, simulating user interactions on the web application as defined in the test script.

**Verify Results and Report**: After each test case execution, the script checks if the application behaves as expected with the given input data. The results are then logged. This can be a simple console output, a detailed report, or integration with a test management tool.

**Loop Over Data Sets:** The entire process is repeated for each set of data in the test data file. This ensures that the application is tested under various data conditions, improving test coverage.

##### **Advantages of DDT with Python and Playwright:**

* Flexibility: Python's easy-to-read syntax and Playwright's browser automation capabilities make a flexible toolset for web testing.
* Scalability: Tests can easily be scaled by simply adding more data sets.
* Efficiency: Automating tests with different data sets saves time compared to manual testing.
* Comprehensive Testing: Helps in uncovering edge cases and ensuring the application can handle different inputs.

##### **How to perform the Data Driven Tesing via Playwright?**

| pip install playwright pytest allure-pytest openpyxl playwright install |
| --- |

| import pytest import openpyxl from playwright.sync\_api import sync\_playwright |
| --- |

| # Fixture to read data from Excel file @pytest.fixture(scope="module") def get\_test\_data():  file\_path = "path\_to\_your\_excel\_file.xlsx"  workbook = openpyxl.load\_workbook(file\_path)  sheet = workbook.active   data = []  for row in range(2, sheet.max\_row + 1):  username = sheet.cell(row, 1).value  password = sheet.cell(row, 2).value  expected\_text = sheet.cell(row, 3).value  data.append({"username": username, "password": password, "expected\_text": expected\_text})   return data |
| --- |

| # Test case class TestWebApp:  @pytest.mark.parametrize("data", get\_test\_data())  def test\_login(self, data):  with sync\_playwright() as p:  browser = p.chromium.launch()  page = browser.new\_page()   # Navigate to your web application login page  page.goto("https://app.vwo.com/")   # Fill in login credentials and submit  page.fill("input[name='username']", data['username'])  page.fill("input[name='password']", data['password'])  page.click("button[type='submit']")   # Add logic to wait for the page to load or for specific elements to appear  # ...   # Verify the expected text  actual\_text = page.text\_content("selector\_for\_text\_element")  assert actual\_text == data['expected\_text']   browser.close()  # Run the test with pytest # pytest -v --alluredir=/path/to/allure/results this\_script.py |
| --- |

## 

## ✅ CI/CD

1. Install Jenkins: Ensure Jenkins is installed and running on your server.

2. Install Necessary Plugins:

- Install the "Git" plugin (if your code is in a Git repository).

- Install any other required plugins for your specific setup.

3. Create a New Freestyle Project:

- In Jenkins, go to "New Item".

- Select "Freestyle project", give it a name, and click "OK".

4. Configure Source Code Management:

- Under "Source Code Management", select "Git".

- Enter your repository URL and credentials if required.

5. Configure Build Triggers (optional):

- Set up triggers for your build, like on code push, periodically, or manually.

6. Add Build Environment Setup:

- In the "Build Environment" section, configure any necessary environment setup.

7. Add Build Steps:

- Click "Add build step" and choose "Execute shell" or "Execute Windows batch command" based on your OS.

- Enter the commands to set up your environment and run Playwright tests. For example:

- pip install playwright

- playwright install

- python -m playwright test

8. Save and Run the Job:

- Click "Save".

- Click "Build Now" to run the job.

9. View Build Results:

- After the build is complete, check the console output for results and logs.