PLAYWRIGHT FOR QA AUTOMATION

#ALEXUSADAYS

PLAYWRIGHT QA AUTOMATION

WHY PLAYWRIGHT & ENV SETUP

#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1 WHY PLAYWRIGHT & ENV SETUP

WHY PLAYWRIGHT

- Playwright is rapidly growing in popularity, with a strong community, extensive documentation, and continuous updates, ensuring it stays relevant for modern web testing.
- **Ease of Setup:** Simple setup process enables testers to quickly begin writing and executing tests.
- **Cross-Browser Testing:** Allows for writing tests once and running them across all modern browsers.
- **Parallel Test Execution**: Reduces testing time by running tests in parallel.
- **Automated Media Capture:** Automatically captures screenshots and videos for easier debugging.
- **Rich API**: Provides a comprehensive API to automate complex user interactions.
- **Built-in Reporting**: Playwright provides built-in tools for generating test reports.
- **Automate Script Creation**: codegen records browser actions (clicks, navigations, inputs, assertions) and generates corresponding Playwright scripts, reducing the effort needed to write test cases.
- **Async Testing**: Playwright handles the dynamic nature of modern web browsing by running tests that adapt to asynchronous events, like loading or data fetching.

WHY PLAYWRIGHT

- The goal of this course is to equip you with the skills required to build Playwright framework and effectively use Playwright documentation to identify the right solutions for your needs. By the end of this course, you'll be able to automate your products using online resources as your guide. This will not only enhance your testing efficiency but also empower you to deliver high-quality software more quickly and with confidence.
- To get started with the playwright we need to install:
  1. IDE (integrated development environment). For IDE we will use VS Code
  2. And we will need to install node js.

- **Visual Studio Code: https://code.visualstudio.com/**
- **Node: https://nodejs.org/en/download**
- **Playwright: https://playwright.dev/docs/intro**
  IN VS Code go to a folder where you want your playwright and run command: npm init playwright@latest

PLAYWRIGHT QA AUTOMATION

CODEGEN WITH PLAYWRIGHT

#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT

PLAYWRIGHT CODEGEN

- Playwright provides a command-line tool to start the code generator. When you run this tool, it launches a browser in a special mode designed for recording interactions. The command typically looks like npx playwright codegen followed by the URL of the web application you want to test.
- **Interactive Learning:** By using Playwright's codegen tool (npx playwright codegen), new testers can immediately start generating test scripts by simply interacting with the web application as they normally would. This hands-on approach is highly intuitive and reduces the initial learning curve.
- **Understanding Through Observation**: Observing how manual interactions are translated into Playwright commands in real-time helps testers understand the relationship between UI actions and the corresponding automation code.
- **Fast Prototyping:** Code generation allows for rapid prototyping of tests. Testers can generate the skeleton of a test case quickly and then refine and expand upon it, which is significantly faster than writing tests from scratch.

PLAYWRIGHT QA AUTOMATION

UNDERSTAND PLAYWRIGHT CODEGEN CODE

#ALEXUSADAYS

UNDERSTAND PLAYWRIGHT CODEGEN CODE

- In our previous video, we looked at how to automate the login process for the website "https://the-internet.herokuapp.com/login" using Playwright's powerful code generation capabilities.
- Code generation allows for rapid prototyping of tests. Testers can generate the skeleton of a test case quickly and then refine and expand upon it, which is significantly faster than writing tests from scratch.
- Codegen is a great way to learn coding. First, you can generate code automatically and then practice by rewriting it yourself. This helps you get better at understanding how code works.
- Today, we're going to build the same test case together, and we'll go through the code line by line explaining what each part does as we proceed.

PLAYWRIGHT QA AUTOMATION

BUILD CLICK ME WEBPAGE

#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT
3. UNDERSTAND CODEGEN CODE
4. BUILD CLICK ME WEBPAGE

BUILD CLICK ME WEBPAGE

- Let's build a simple HTML page to understand HTML structure. Later we will use this page to practice finding locators.
- We will need to install Live Server Extension in Visual Studio Code to be able to run html page locally.
- Basic HTML structure representation for selecting locator looks something like this:
  <openTag attributeName="attributeValue"> Content (some text) <closingTag>
- On a real page it will look like this:

```
<button id="clickButton">Click me</button>
```

PLAYWRIGHT QA AUTOMATION

HOW TO FIND SELECTORS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT
3. UNDERSTAND CODEGEN CODE
4. BUILD CLICK ME WEBPAGE
5. HOW TO FIND SELECTORS

HOW TO FIND SELECTORS

- Let's understand how we can find elements and add specific selectors to our code. Remember the basic structure is: <tag attribute="value"> content</tag>

We will focus on:

- Selecting by ID - targets the element by its unique ID
- Selecting by Class - for elements with specific class
- By Tag Name and Class – combining tag and class for unique id
- By Attribute Value - for elements with specific attribute value
- Partial Attribute Value - elements whose attribute contains specific text
- By Text Content - finds elements by their visible text
- By Playwright Specific Locators – https://playwright.dev/docs/locators
- Use Ranorex Selocity Chrome Plugin to help with selecting elements.

PLAYWRIGHT QA AUTOMATION

PLAYWRIGHT ASSERTIONS

#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT
3. UNDERSTAND CODEGEN CODE
4. BUILD CLICK ME WEBPAGE
5. HOW TO FIND SELECTORS
6. PLAYWRIGHT ASSERTIONS

PLAYWRIGHT ASSERTIONS

- Playwright assertions are statements that validate the state of the application under test. They are used to compare the actual output of a part of the application with its expected output. If the actual output matches the expected output, the assertion passes; if not, it fails, indicating a potential issue with the application.

Here are some common types of Playwright assertions:

- Visibility: Asserting that an element is visible or hidden.
- Text Content: Checking that an element contains certain text.
- Attribute Values: Ensuring that an element has a specific attribute value.
- Count: Verifying the number of elements that match a specific selector.
- State: Such as ensuring a checkbox is checked or an element is enabled or disabled.
- Navigation: Confirming that the application has navigated to the correct URL after an action.

More on assertions here – https://playwright.dev/docs/test-assertions

PLAYWRIGHT QA AUTOMATION

RUNNING TESTS & CONFIG

#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT
3. UNDERSTAND CODEGEN CODE
4. BUILD CLICK ME WEBPAGE
5. HOW TO FIND SELECTORS
6. PLAYWRIGHT ASSERTIONS
7. RUNNING TESTS & CONFIG

RUNNING TESTS & CONFIG

- Running tests in Playwright can be done in various ways depending on your needs.

Let's look at some popular example of how to run playwright tests:

- Using CLI (command line interface):
  - To run all tests: npx playwright test
  - To run a specific test file: npx playwright test tests/example.spec.js
  - Tests with a specific browser: npx playwright test --project=chromium
  - Tests with a specific tag: npx playwright test --grep @smoke
  - Tests with a specific name: npx playwright test tests/example.spec.js --grep "test name"
- Using VSC app: Playwright Test for VSCode
- Using build in UI runner: npx playwright test –ui
- Adding to package.json specific execution sripts: "test:smoke": "npx playwright test --grep @smoke"

RUNNING TESTS & CONFIG

- How your tests will run will also depend greatly on you playwright.config

Let's get familiar with some of the configuration parameters we have:

- Run test by default in headless more or not - headless: true
- Set default url to start with for your tests - baseURL: 'https://example.com'
- Run tests in parallel mode or not - fullyParallel: true
- Global use setting and project level settings.

I added public repository with config here:
https://github.com/alexusadays/Playwright-for-QA-Automation

Let's look into actual code!

PLAYWRIGHT QA AUTOMATION

SETTING PAGE & HOOKS

#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT
3. UNDERSTAND CODEGEN CODE
4. BUILD CLICK ME WEBPAGE
5. HOW TO FIND SELECTORS
6. PLAYWRIGHT ASSERTIONS
7. RUNNING TESTS & CONFIG
8. SETTING PAGE & HOOKS

SETTING PAGE & HOOKS

- In Playwright, the built-in {page} is a fixture. Fixtures automatically set up and tear down resources needed for tests. Built in {page} provides a fresh page tab in a specific browser within a browser context for each test. It ensures that each test runs in isolation.
- However, sometimes you may need to set up the browser and context yourself to customize the configuration or manage multiple contexts. You can do this manually and organize the process using hooks such as beforeAll, afterAll, beforeEach, and afterEach:

- beforeAll - hook is used to execute code before any tests in the test suite run.
- afterAll - hook is used to execute code after all tests in the test suite have finished
- beforeEach - hook is used to execute code before each individual test runs.
- afterEach - hook is used to execute code after each individual test has finished.

- Hooks can be used for many things, like initialize and clean up test data before and after tests or perform login operations and store session tokens or cookies. In this lesson we will look into using hooks specifically for manual page setup and tear down.

PLAYWRIGHT QA AUTOMATION

PAGE OBJECT MODEL

#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT
3. UNDERSTAND CODEGEN CODE
4. BUILD CLICK ME WEBPAGE
5. HOW TO FIND SELECTORS
6. PLAYWRIGHT ASSERTIONS
7. RUNNING TESTS & CONFIG
8. SETTING PAGE & HOOKS
9. PAGE OBJECT MODEL
10. PLAYWRIGHT API TESTING

POM - PAGE OBJECT MODEL

- The Page Object Model (POM) is a way to organize your test code. It helps you manage the interactions with your web page in an organized and reusable way. Instead of writing all your test steps directly in your test files, you create a separate class that represents each page of your application.
- When creating POM framework try to balance 2 principles:

- DRY (Don't Repeat Yourself) concept in software development that aims to reduce repetition of code.
- KISS (Keep It Simple, Stupid) philosophy that emphasizes simplicity in design and implementation. The idea is to avoid unnecessary complexity and aim for easy-to-understand solutions.

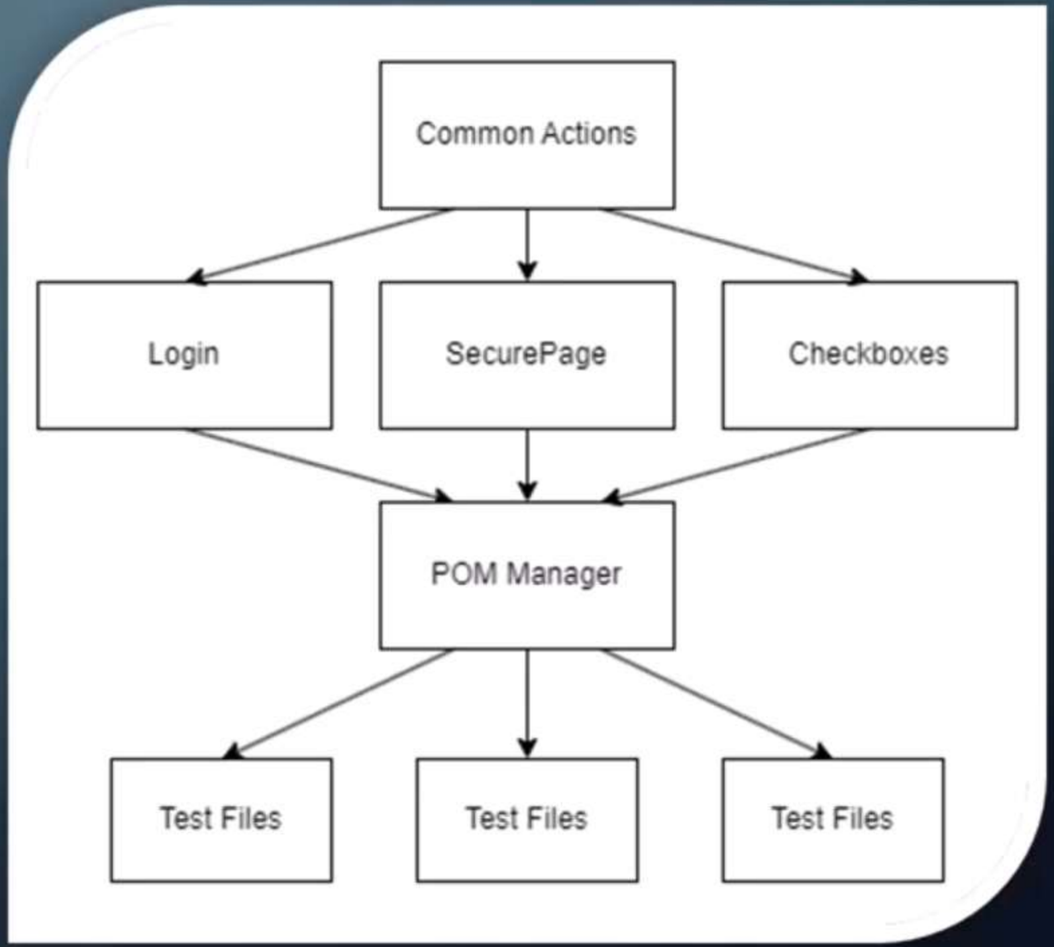
- Here are the main benefits of project using POM:

- **Easier to Read**: It keeps your test code clean and easy to understand.
- **Reusability:** You can reuse code for interacting with the same page across multiple tests.
- **Maintainability:** If something changes on the page, you only need to update it in one place.

POM - PAGE OBJECT MODEL

Project Structure:

```
project-name-root-dir/
├── pages/
│   ├── LoginPage.js
│   ├── SecurePage.js
│   ├── CheckboxesPage.js
│   └── PomManager.js
├── tests/
│   └── test.spec.js
└── utils/utils
    └── CommonActions.js
```



PLAYWRIGHT QA AUTOMATION

API VERIFICATION

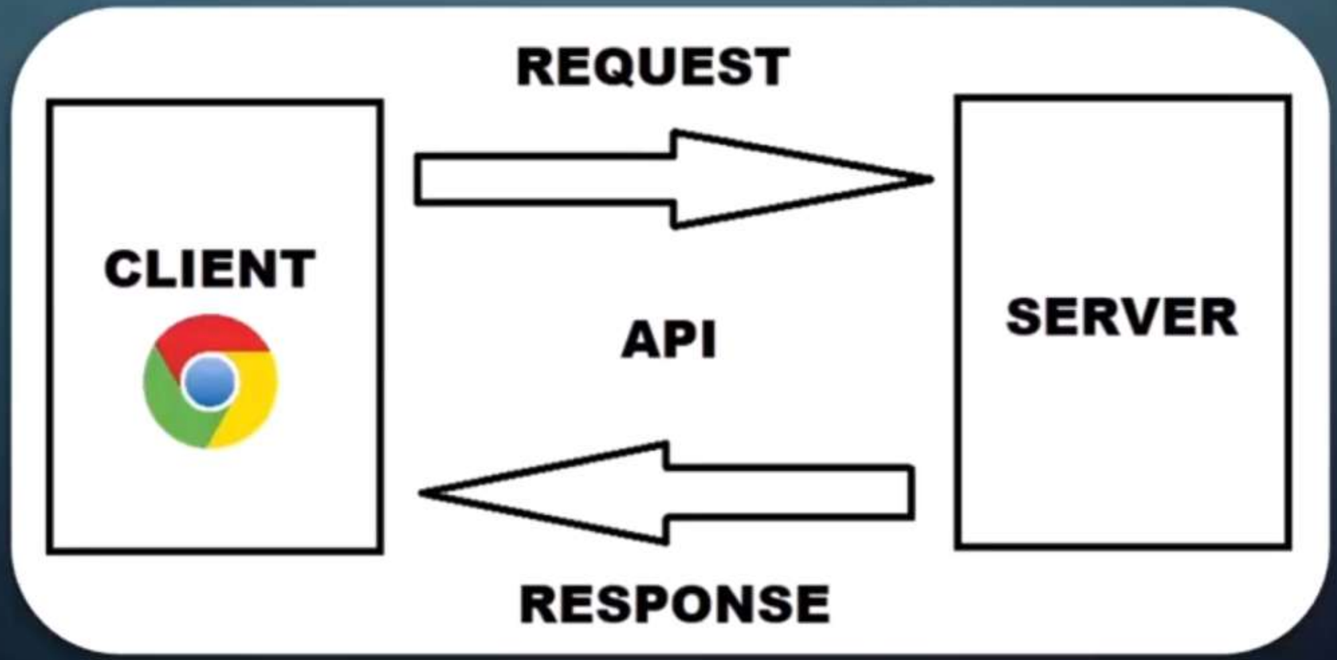
#ALEXUSADAYS

PLAYWRIGHT ROADMAP

1. WHY PLAYWRIGHT & ENV SETUP
2. CODEGEN WITH PLAYWRIGHT
3. UNDERSTAND CODEGEN CODE
4. BUILD CLICK ME WEBPAGE
5. HOW TO FIND SELECTORS
6. PLAYWRIGHT ASSERTIONS
7. RUNNING TESTS & CONFIG
8. SETTING PAGE & HOOKS
9. PAGE OBJECT MODEL
10. PLAYWRIGHT API TESTING

API VERIFICATION

- API stands for Application Programming Interface.
  In web API is a set of rules and protocols that allows different software applications to communicate.
  API allows web browser (client) to communicate with server.



API VERIFICATION

- One of the most popular API approaches is REST.
  REST API - Representational State Transfer is a type of web API that follows a set of architectural principles and constraints. It is designed to create a simple and standardized way for software to communicate.
- REST APIs use standard HTTP methods to perform actions on resources.
  HTTP - Hypertext Transfer Protocol, is the language that web browsers and web servers use to talk to each other on the internet.
- The primary HTTP methods used are:

- **GET**: Retrieve data from a resource.
- **POST**: Create a new resource.
- **PUT**: Update or replace an existing resource.
- **DELETE**: Remove a resource.
- **PATCH**: Partially update a resource.

API VERIFICATION

- With an API request method there is a response with a status code. HTTP status codes are three-digit numbers returned by a web server in response to a client's request made to the server. They provide information about the result of the request and help both the client and server understand how to proceed.
- Response Status Codes and examples (this is not a full list):

- 1xx (Informational) – 100 Continue, 101 Switching
- 2xx (Successful response) – 200 OK, 201 Created
- 3xx (Redirection messages) - 300 Multiple Choices, 301 Moved Permanently
- 4xx (Client error responses) - 401 Unauthorized, 404 Not Found
- 5xx (Server error responses) - 502 Bad Gateway, 504 Gateway Timeout
- See more details at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

API VERIFICATION

- When testing APIs with Playwright, the main objective is to implement and verify various types of requests such as GET, POST, DELETE, or others. The process typically involves the following steps:
- Implementing the Request: Create and send the API request using Playwright, specifying the necessary method (GET, POST, DELETE, etc.), headers, parameters, and body data as required.
- Verifying the Response:
  - Status Code: Ensure that the response returns the correct HTTP status
  - Response Data: If the response includes data, verify that the data is correct and matches the expected values.
- Error Handling and Edge Cases: Test various edge cases and error conditions to ensure the API handles them gracefully. This includes sending invalid data, missing required parameters, and simulating server errors to check if the application responds appropriately.
- Documentation Compliance: Compare the actual API response against the API documentation to ensure that the implementation aligns with what is specified. This includes verifying endpoint paths, required parameters, response formats, and error codes.