**TypeScript:**

TypeScript is a strongly typed superset of JavaScript that compiles to plain JavaScript. It provides static typing, interfaces, and modern ES features, which help in building large-scale applications with better maintainability and fewer runtime errors.

**Key Differences:**

* TypeScript has **static typing**, while JavaScript is dynamically typed.
* TypeScript supports **interfaces, generics, and enums**.
* It provides better tooling support like IntelliSense and compile-time error checking.

**Any vs Unknown Type?**

**Any** wont do any type checking at the compilation time but the **Unknown** type will make to check the type before performing any operation on the variable.

**Interfaces vs Type**

Use interface for objects and class structures when you need extension and merging.

Use type for unions, intersections, and function types.

For objects, both works, but interface is more flexible.

**Const vs readonly?**

1. const applies to variables, while readonly applies to object properties and class members.
2. const prevents reassignment of the variable itself, while readonly allows initialization but prevents further modifications.
3. readonly only exists at compile-time, whereas const enforces immutability at runtime.

Class vs Interfaces ?

Union Types ?

**What is the difference between == and === in TypeScript?**

**Answer:**

* == checks **value equality** with type coercion.
* === checks **both value and type**.

Example:

typescript

console.log(5 == "5"); // true (coerced comparison)

console.log(5 === "5"); // false (strict comparison)

**General Playwright Questions**

1. **What is Playwright, and how does it compare to Selenium?**

Playwright is an open-source automation framework developed by Microsoft for end-to-end (E2E) testing of web applications. It allows you to automate browsers such as Chromium, Firefox, and WebKit with a single API, supporting multiple programming languages like JavaScript, TypeScript, Python, C#, and Java.

**Comparison: Playwright vs. Selenium**

| **Feature** | **Playwright** | **Selenium** |
| --- | --- | --- |
| **Architecture** | Modern, built from scratch for parallel execution and speed | Older, designed originally for single-threaded execution |
| **Browser Support** | Chromium, Firefox, WebKit (including mobile emulation) | Chrome, Firefox, Edge, Safari, Internet Explorer (IE support is legacy) |
| **Execution Speed** | Faster, thanks to WebSocket communication and headless mode optimization | Comparatively slower due to HTTP-based WebDriver protocol |
| **Auto-Waiting** | Built-in auto-waiting for elements to be ready before actions | Requires explicit waits or polling |
| **Parallel Execution** | Native parallel execution via multiple browser contexts | Requires Selenium Grid or third-party tools for parallel execution |
| **Test Framework Integration** | Built-in test runner (@playwright/test for JS/TS) | Supports various testing frameworks like JUnit, TestNG, PyTest |
| **Selectors** | Supports advanced selectors like CSS, XPath, text, and locators() | Primarily relies on CSS and XPath |
| **Mobile Testing** | Supports mobile emulation natively | Uses Appium for mobile testing |
| **Headless Mode** | Optimized headless execution with better performance | Headless mode supported but can be slower |
| **Setup Complexity** | Easier setup, single dependency installation | More complex setup with WebDriver requirements |
| **Network Interception** | Supports request/response interception for mocking | Limited support; requires additional tools |
| **Shadow DOM & iFrames** | Better handling of Shadow DOM and nested iFrames | Supports iFrames but can struggle with Shadow DOM |

**When to Use Playwright?**

* When you need fast and reliable web automation.
* If you require parallel execution without additional setup.
* When dealing with modern web applications using Shadow DOM, iFrames, or mobile emulation.
* For advanced network interception and mocking capabilities.

**When to Use Selenium?**

* If you need broad browser support, including legacy browsers (IE).
* When integrating with existing Selenium-based frameworks.
* If you require deep integrations with different test frameworks like JUnit, TestNG, or NUnit.

1. **What are the limitations of playwright?**

**1. Limited Browser Support for Legacy Browsers**

* Playwright does **not** support **Internet Explorer (IE)** and has limited support for older versions of browsers.
* If your application requires testing on legacy browsers, **Selenium is a better choice.**

**2. Higher Resource Usage**

* Playwright runs **multiple browser instances in parallel**, which can consume more CPU and memory compared to Selenium WebDriver.
* This might be an issue when running tests on low-resource machines or cloud environments with limited quotas.

**3. Limited Ecosystem Compared to Selenium**

* Selenium has been around for over a decade, so it has a **richer ecosystem** with many third-party integrations (like Appium for mobile testing).
* Playwright's community and plugins are growing but are still **not as extensive as Selenium’s.**

**4. No Built-in Support for Real Mobile Devices**

* Playwright supports **mobile emulation**, but it does not natively support **real device testing** like Appium.
* If you need to test on actual mobile devices, you’ll have to integrate Playwright with **third-party services like BrowserStack or Sauce Labs**.

**5. Learning Curve for Beginners**

* Playwright has a **modern API with async/await** patterns, which might be **challenging** for testers coming from a Selenium background (which mainly uses synchronous operations).
* The use of **context, browser, and page** fixtures can be confusing at first.

**6. Limited Community Support Compared to Selenium**

* Selenium has a **larger and older community**, meaning you can find **more tutorials, Stack Overflow answers, and troubleshooting guides**.
* Playwright’s documentation is great, but if you run into a unique issue, you might find fewer community solutions.

**7. No Built-in Image Comparison**

* Unlike tools like Cypress or some Selenium-based frameworks (like WebDriverIO), Playwright **does not have built-in visual testing** capabilities.
* You need **third-party tools like Percy, Applitools, or Playwright Test Snapshots** for UI comparisons.

**8. Debugging Can Be Tricky**

* Since Playwright tests are often **executed in parallel**, debugging failed tests can be more complex.
* You might need to **reproduce the issue** using Playwright’s tracing and debugging tools, which require additional setup.

**9. Limited Built-in Reporting**

* Playwright has a basic test runner (@playwright/test), but for **advanced reporting**, you may need external tools like **Allure Report** or third-party integrations.

**10. Not Ideal for Long-Running Tests**

* Playwright is optimized for **fast execution**, but if you need **long-running UI tests**, some stability issues might arise.
* Running **very long sessions** (e.g., overnight regression runs) could lead to browser crashes due to memory leaks.

1. **Playwright Architecture (**[**Playwright Architecture**](https://www.youtube.com/watch?v=eSlxSmCory4)**)**

A diagram of a computer

AI-generated content may be incorrect.

The Playwright is aligned with the architecture of the modern browsers and runs tests out-of-process, and it is free of the typical in-process test runner limitations of Cypress.

Playwright communicates all requests between client and server through a single WebSocket connection which is comparatively better than the Selenium HTTP Connection protocol for automation.

**Playwright Architecture**

At the client end, you have your code written in different programming languages like JavaScript, Java, Python,C#, etc. The Playwright Server communicates with the client and different web browser engines. The Playwright uses the Chrome DevTools Protocol CDP) to communicate with Chromium. For Firefox and WebKit, Playwright implemented their own protocols like CDP.

A WebSocket connection is established by sending a request to the server from a client through a process called the WebSocket handshake.

WebSockets have a much lower latency in terms of messages being sent to their clients due to the open connection, which is contrary to the long polling where the connection must be reestablished with each request.

WebSockets, send the response as soon as it gets it in real time. The WebSocket connection uses WebSocket communication protocol which provides a full-duplex communication channel over a single TCP connection.

A full-duplex system allows communication in both directions. All modern browsers support this protocol. It is a stateful protocol where the connection between the client and server will stay alive until either client or server terminates it.

Once you will trigger your test, the code will be converted into JSON format, and then it will be sent to the server using Web Socket Protocol. The Playwright communicates all requests through a single web socket connection, which stays in place until all test execution is completed. Since commands are sent on a single connection, chances of test failure or flakiness are less, and commands are executed quickly.

This architecture is contrary to Selenium which uses HTTP connection protocol and sends each command like browser opening, clicking, sending keys, or closing the browser as a separate HTTP request. Also, in Selenium, the connection between the server and the client will be terminated after each request and re-established for the next request. This is the reason why Playwright is faster than Selenium.

[Test Script] → [Playwright API] → [Playwright Server] → [Browser Engine]

(You write) (Translates) (Sends commands) (Executes actions)

1. **What browsers does Playwright support?**

Playwright supports **four major browser engines**:

1. **Chromium** (Google Chrome, Microsoft Edge)
2. **Firefox**
3. **WebKit** (Safari engine)
4. **Microsoft Edge** (via Chromium)

**Detailed Browser Support:**

| **Browser** | **Playwright Support** | **Notes** |
| --- | --- | --- |
| **Google Chrome** | ✅ Supported | Uses **Chromium** by default. Can also run full Chrome if installed. |
| **Microsoft Edge** | ✅ Supported | Based on **Chromium** engine. |
| **Firefox** | ✅ Supported | Works with Playwright’s custom Firefox build. Some features may be missing compared to the official Firefox. |
| **WebKit (Safari engine)** | ✅ Supported | Enables testing for Safari-based rendering, but not the full Safari browser itself. |
| **Opera, Brave, Vivaldi** | ⚠️ Partially Supported | Can work if they are Chromium-based but not officially tested. |
| **Internet Explorer (IE)** | ❌ Not Supported | No support for legacy browsers. Selenium is required for IE testing. |

**Headless & Headed Modes**

* **Headless Mode:** Playwright runs all supported browsers in headless mode (no UI), which improves speed and efficiency.
* **Headed Mode:** You can also run tests in a visible browser window for debugging.

1. **What are the key features of Playwright?**

Playwright offers several powerful features that make it a great choice for modern web automation. Here are the key features:

**1. Cross-Browser Support**

✅ Supports **Chromium (Chrome, Edge), WebKit (Safari engine), and Firefox** with a single API.  
✅ Ensures compatibility testing across different browsers.

**2. Multi-Language Support**

✅ Playwright can be used with:

* **JavaScript/TypeScript**
* **Python**
* **C# (.NET)**
* **Java**

**3. Auto-Waiting & Smart Assertions**

✅ **Auto-waits** for elements to be visible, clickable, or ready before performing actions.

✅ Playwright keeps retrying the assertion **until the expected condition is satisfied** or **the timeout expires**.  
✅ Reduces flakiness by retrying failed actions instead of using explicit waits.

await page.locator('button#submit').click(); // No need for explicit waits

**4. Parallel Test Execution & Multi-Browser Contexts**

✅ Allows running **multiple tests in parallel** for faster execution.  
✅ **Isolated browser contexts** let tests run independently without affecting each other.

const context1 = await browser.newContext();

const context2 = await browser.newContext(); // Each context acts like a separate user session

**5. Headless & Headed Execution**

✅ Supports **headless mode** for faster automation.  
✅ Can run in **headed mode** for debugging or UI validation.

const browser = await chromium.launch({ headless: false }); // Run in visible mode

**6. Network Interception & API Testing**

✅ Allows intercepting network requests and modifying responses.  
✅ Supports mocking, modifying headers, and tracking network activity.

await page.route('\*\*/api/data', route => route.fulfill({ json: { message: "Mocked Response" } }));

✅ Can be used for **API testing** without opening a browser.

const response = await request.get('https://api.example.com/data');

console.log(await response.json());

**7. Mobile Emulation & Geolocation Testing**

✅ Supports testing on **mobile device emulators** with different screen sizes and touch capabilities.  
✅ Allows **geolocation testing** for location-based applications.

const context = await browser.newContext({

viewport: { width: 375, height: 667 },

geolocation: { latitude: 37.7749, longitude: -122.4194 },

userAgent: 'Mozilla/5.0 (iPhone; CPU iPhone OS 14\_0 like Mac OS X)',

});

**8. Built-in Screenshot & Video Recording**

✅ Supports **screenshots and video recording** for debugging failed tests.

await page.screenshot({ path: 'screenshot.png' });

await context.tracing.stop({ path: 'trace.zip' }); // Record execution trace

**9. Better Handling of Shadow DOM & iFrames**

✅ Can interact with elements inside **Shadow DOM** and nested **iFrames**, which is challenging in Selenium.

const frame = await page.frame({ name: 'iframe\_name' });

await frame.locator('#element').click();

**10. Built-in Test Runner (@playwright/test)**

✅ Includes a **Playwright Test Runner** with:

* **Parallel test execution**
* **Assertions**
* **Automatic retries**
* **Test fixtures** for setup/teardown

**Why Choose Playwright?**

✅ **Faster & more reliable** than Selenium  
✅ **Modern architecture** with better debugging tools  
✅ **Great for UI, API, and network testing**

1. **What are playwright’s inbuilt assertions**

Playwright provides several **smart assertions** that automatically retry:

| **Assertion** | **Description** |
| --- | --- |
| toBeVisible() | Checks if an element is visible |
| toBeHidden() | Checks if an element is hidden |
| toHaveText(text) | Checks if an element contains the expected text |
| toHaveValue(value) | Verifies the value of an input field |
| toBeChecked() | Ensures a checkbox or radio button is checked |
| toBeDisabled() | Checks if an element is disabled |
| toBeEnabled() | Checks if an element is enabled |
| toHaveAttribute(attr, value) | Verifies an element’s attribute |

1. **How do you install Playwright?**

Pre-requisite: Node.js

Run the install command and select the following to get started:

npm init playwright@latest

* 1. Choose between TypeScript or JavaScript (default is TypeScript)
  2. Name of your Tests folder (default is tests or e2e if you already have a tests folder in your project)
  3. Add a GitHub Actions workflow to easily run tests on CI
  4. Install Playwright browsers (default is true)

1. **How do you launch a browser in Playwright?**

import { chromium } from '@playwright/test'

(async () => {

const browser = await chromium.launch(); // Launches Chromium

const page = await browser.newPage(); // Opens a new page (tab)

await page.goto('https://example.com'); // Navigate to a webpage

console.log(await page.title()); // Print page title

await browser.close(); // Close the browser

})();

Launching Browser with Playwright Test Runner

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

test('Open Google', async ({ page }) => {

await page.goto('https://google.com');

await expect(page).toHaveTitle(/Google/);

});

1. **What are the different browser contexts in Playwright?**

**Default Context (Single Context)**

* When you open a new browser, it has a **single default context**.
* If you use browser.newPage(), all pages **share** the same session.

**Multiple Contexts (Isolated Sessions)**

* Each browser.newContext() creates a completely new, isolated session.

const browser = await chromium.launch();

const context1 = await browser.newContext();

const page1 = await context1.newPage();

await page1.goto('https://example.com');

const context2 = await browser.newContext();

const page2 = await context2.newPage();

await page2.goto('https://example.com'); // Opens in a separate session

await browser.close();

**Incognito Context (Private Mode)**

* Every new context is like an **Incognito Window** in a browser.
* **Automatically clears storage** when closed.

1. **What is the difference between Page and Browser Context in Playwright?**

A **Page** in Playwright represents a single browser tab, while a **Browser Context** is an isolated session (like an incognito window) that can have multiple pages. Pages within the same context share cookies and storage, but different contexts do not. This allows testing multiple users or scenarios in parallel without data interference.

1. **How do you handle multiple browser tabs in Playwright?**

In Playwright, you can handle multiple browser tabs using context.newPage(), which opens a new tab within the same session.

import { chromium } from '@playwright/test'

(async () => {

const browser = await chromium.launch({ headless: false });

const context = await browser.newContext(); // Creates a session

const page1 = await context.newPage(); // Opens first tab

await page1.goto('https://example.com');

const page2 = await context.newPage(); // Opens second tab

await page2.goto('https://playwright.dev');

await browser.close();

})();

1. **What are the different ways to locate elements in Playwright?**

Playwright provides multiple ways to locate elements on a webpage:

* 1. **Text Selector** → page.getByText('Login') (Finds an element by visible text)
  2. **Role-Based Selector** → page.getByRole('button', { name: 'Submit' })
  3. **Placeholder Selector** → page.getByPlaceholder('Enter name') (For input fields)
  4. **Label Selector** → page.getByLabel('Username') (For form labels)
  5. **Test ID Selector** → page.getByTestId('login-button') (Useful for testing)
  6. **CSS Selector** → page.locator('.btn-primary') (Finds by class, ID, tag, etc.)
  7. **XPath Selector** → page.locator('//button[text()="Submit"]')
  8. **Chained Locators** → page.locator('.form').locator('button') (Find inside a parent)

1. **How do you handle dynamic elements in Playwright?**

Dynamic elements (elements that load asynchronously or change states) can be handled using wait strategies in Playwright:

**Use locator.waitFor() (Wait for Element to Appear)**

await page.locator('.dynamic-button').waitFor();

await page.locator('.dynamic-button').click();

✅ **Waits until the element is visible before interacting.**

**Use await page.waitForSelector()**

await page.waitForSelector('.dynamic-button', { state: 'visible' });

await page.click('.dynamic-button');

✅ **Waits for the element to appear in the DOM before clicking**.

**Use await page.waitForLoadState() (For Page or Network Stability)**

await page.waitForLoadState('networkidle'); // Waits until network requests finish

✅ **Useful for waiting until AJAX calls or API responses complete.**

**Use locator.nth() (Handle Multiple Matching Elements)**

await page.locator('.item').nth(2).click(); // Clicks the 3rd matching element

✅ **Helps when elements load dynamically but you need a specific one.**

**Use expect(locator).toBeVisible() (Assertion with Auto-Retry)**

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

const button = page.locator('.dynamic-button');

await expect(button).toBeVisible(); // Auto-retries until found

await button.click();

✅ **Best for Playwright test runner (auto-retries for stability).**

**Playwright API & Functionalities**

1. **How do you perform click and type operations in Playwright?**

await page.locator('input#email').type('test@example.com');

await page.locator('button#signup').click();

1. **What is the difference between locator() and page.locator()?**

page.locator(selector) → **Finds elements globally** on the page.

locator(selector) → **Finds elements inside another locator** (parent-child relationship).

1. **How do you handle alerts, popups, and modal dialogs in Playwright?**

page.on('dialog', callback) → Handle JavaScript alerts (alert, confirm, prompt).

context.waitForEvent('page') → Handle new popup windows/tabs.

locator('.modal') → Handle custom modals in HTML.

1. **How do you perform assertions in Playwright tests?**

toHaveTitle(value) → Checks if the page title matches the expected value.

toBeVisible() → Verifies if an element is visible on the page.

toHaveText(value) → Ensures an element contains specific text.

toHaveAttribute(name, value) → Checks if an element has a given attribute.

toBeEnabled() / toBeDisabled() → Validates if an element is enabled or disabled.

1. **How do you handle file uploads and downloads in Playwright?**

**// Locate file input and set file to upload**

const fileInput = page.locator('input[type="file"]');

await fileInput.setInputFiles('path/to/local/file.pdf');

**// Wait for the download event**

const [download] = await Promise.all([

page.waitForEvent('download'), // Wait for download event

page.locator('a#download-button').click() // Click to download

]);

**// Save As**

await download.saveAs('downloads/sample-file.pdf');

1. **What is waitForSelector() in Playwright?**

waitForSelector(selector) is a **method used to wait until an element appears in the DOM** before interacting with it. It ensures that Playwright does not attempt actions on elements that are not yet available.

await page.waitForSelector('#submit-button');

await page.click('#submit-button');

**When to Use waitForSelector()?**

1. **Waiting for dynamically loaded elements** (e.g., AJAX content).
2. **Ensuring stability in tests** by avoiding race conditions.
3. **Handling slow-loading pages** where elements may not be immediately available.

Instead of waitForSelector(), Playwright recommends to use waitfor()

await page.locator('#submit-button').waitFor();

1. **How do you take a screenshot in Playwright?**

Full Page Screenshot:

await page.screenshot({ path: 'screenshot.png', fullPage: true });

Screenshot of Specific element:

const element = await page.locator('#my-element');

await element.screenshot({ path: 'element.png' });

1. **What are Tracing and Debugging in Playwright?**

Tracing captures screenshots, network requests, and logs for analysis.

import { test } from '@playwright/test';

test.use({ trace: 'on' }); // 'on', 'off', or 'retain-on-failure'

test('trace example', async ({ page }) => {

await page.goto('https://example.com');

});

npx playwright show-trace trace.zip

$env:PWDEBUG=1; npx playwright test

1. **What are the different ways of doing debugging in Playwright ?**
   1. **Using await page.pause() (Interactive Debugging)**

Just add above statement into the test case you want to debug and run the test case from command line. Playwright will automatically starts playwright inspector for this test case.

* 1. **Using page.pause()**

**3. Using DEBUG=pw:api (Logging detailed API Calls)**

* For Powershell, run: $env:DEBUG="pw:api";npx playwright test
* Or
* For Windows command line run: set DEBUG=pw:api && npx playwright test
* This logs all Playwright API calls in the terminal for better debugging.

**4. Using --debug flag (Full Debug Mode)**

* Run: npx playwright test --debug
* This starts Playwright in a debug UI mode with step-by-step execution and live inspection.

**5. Using Tracing Execution (traceViewer)**

* Enable Tracing

await context.tracing.start({ screenshots: true, snapshots: true });

* After test failure, view trace using npx playwright show-trace trace.zip.

1. **What are the different PW DEBUG options ?**

Playwright provides multiple ways to debug tests, including PWDEBUG and DEBUG variables.

* 1. PWDEBUG (Interactive Debugging)

Controls whether Playwright Inspector is enabled.

PWDEBUG=1: Opens Playwright Inspector, pauses execution, runs in headed mode

PWDEBUG=console: Prints debug logs to console, does not pause execution

**Usage (Windows Command Prompt)**

set PWDEBUG=1 && npx playwright test # Opens Playwright Inspector

set PWDEBUG=console && npx playwright test # Logs debugging info to console

Best for interactive debugging or log-based debugging.

* 1. DEBUG="PW:API" (Enable API Debug Logs)

Logs detailed API calls made by Playwright, useful for troubleshooting.

**Usage (Windows Command Prompt)**

set DEBUG=PW:API && npx playwright test

Best for \*\*understanding internal Playwright API requests\*\*.

1. **How can you apply your TypeScript basics knowledge in testing?**
   * Using TypeScript, I ensure type safety in test scripts, define custom interfaces for API responses, use async/await for handling asynchronous operations, and leverage generics for reusable utility functions in Playwright tests.
2. **I have a list of objects, but I want to get only one object with a specific condition. What are the different ways to achieve this in TypeScript?**
   * Using .find(): const obj = list.find(item => item.id === targetId);
   * Using .filter()[0]: const obj = list.filter(item => item.id === targetId)[0];
   * Using a for loop or forEach().
3. **How does Playwright interact with web pages? (Architecture of Playwright)**
   * Playwright uses a multi-process architecture where it interacts with browsers via WebSockets or CDP (Chrome DevTools Protocol). It creates a **BrowserContext** for isolated sessions and **Page** objects to interact with elements using APIs that operate outside the DOM, ensuring reliability and automation across Chromium, Firefox, and WebKit.
4. **What is strict mode violation in Playwright?**
   * Strict mode in Playwright ensures unique element selection. If multiple elements match a locator while using locator(), a **strict mode violation** occurs, and Playwright throws an error. This prevents ambiguous selections.
5. **Your webpage has changed drastically, and many elements got updated on the UI. What is the fastest possible way to create new test cases using Playwright?**
   * Use **Codegen (npx playwright codegen <URL>)** to quickly generate new selectors and tests.
   * Utilize **Page Object Model (POM)** for maintainability, so only selectors need updates.
   * Leverage **auto-waiting and resilient locators** for stable tests.
6. **What are the two different ways to debug tests in Playwright?**
   * Using debug mode: DEBUG=pw:api npx playwright test (logs Playwright API calls).
   * Using .pause(): Add await page.pause(); to pause execution and inspect elements.
7. **I want to perform some action (PDF generation/send an email) after generating a report. How can I achieve that?**

We can achieve this using **onEnd hook** in playwright.config.ts to trigger post-test actions.

1. **I have a table with multiple rows, each containing a button. I want to click only one specific button. How can I achieve this in Playwright?**
   * Using nth-child: await page.locator('table tr:nth-child(3) button').click();
   * Using filter: await page.locator('table tr').filter({ hasText: 'Target Row' }).locator('button').click();
2. **How does Playwright work internally to find and click elements?**
   * Playwright first queries elements using **selector engines** (CSS, XPath, Role, Text, etc.).
   * It waits for the element to be attached, visible, and stable before interacting.
   * It then issues a **protocol command** to the browser for clicking, ensuring reliable execution.
3. **How can we take screenshot at the test level and how to add those screenshots in the final report.**

You can take screenshot at the test level by adding either of the below statement

await page.screenshot({ path: pagescreenshot.png})

or

const locatorLogin = await page.getByRole('button', {name:'Login' });

await locatorLogin.screenshot(‘elementscreenshot.png’);

To include this into final report you need to use testInfo. Please make sure that you are passing ‘testInfo’ as parameter to your test case

// Attach it to the Playwright report

        testInfo.attach('Screenshot', {

            path: 'test-results/AccountCreateMessageLabel.png',

            contentType: 'image/png'

        });

**Playwright with Automation Frameworks**

1. **How do you integrate Playwright with Jest or Mocha?**
2. **How do you run Playwright tests in parallel?**
3. **How do you use Playwright with a CI/CD pipeline?**
4. **How do you handle authentication in Playwright tests?**
5. **What is the use of auto-wait in Playwright?**