

1)What is Test Isolation in Playwright?

Answer:

Test isolation in Playwright means every test runs in a fresh browser context with its own cookies, localStorage, and session, ensuring tests do not interfere with each other.

No shared cookies

[**Cookies** = small data sent with every HTTP request, Stored by browser, Automatically sent to server, Used for authentication & sessions, Have expiry]

Use : Login session, Auth tokens]

No shared local storage

[**LocalStorage** = browser-side storage only, Stores data as key-value pairs, Not sent to server, No expiry (until cleared), Accessible via JavaScript]

No shared cache

[**Cache** = stored website resources, Stores static files, Improves page load speed,]

2)How Playwright Achieves Test Isolation

Answer:

By default, Playwright Test:

Launches one Browser

Creates a new BrowserContext per test

Creates a new Page per test

3) What is Playwright?

Answer:

Playwright is an end-to-end automation framework by Microsoft used to test web applications.

It supports Chromium, Firefox, and WebKit with a single API.

4) Why Playwright over Selenium?

Answer:

Auto-waiting (no need for explicit waits), support Video recording.

Faster execution - using web-socket

Supports multiple browsers & tabs

Better handling of modern web apps (SPA)

Built-in test runner

Built-in Parallel Execution

Auto-Retry & Test Isolation

Powerful Locators - getByRole..

Built-in API Testing

Easy Debugging Tools-Headed mode, UI mode, Trace Viewer, Screenshots, Videos.

5) Why TypeScript is preferred with Playwright?

Answer:

Compile-time error checking

Strong typing (Locator, Page, Browser)

Better IDE support (auto-complete)

Fewer runtime errors

6) Difference between page ,context and browser?

Answer:

Browser = actual browser instance [Examples: Chromium, Chrome, Firefox, WebKit]

BrowserContext = isolated user session [Like an Incognito window]

Page = a single tab

7) Real Playwright Test Example

Answer:

```
test('browser vs context vs page', async ({ browser }) => {
  const context = await browser.newContext();
  const page = await context.newPage();
  await page.goto('https://example.com');
});
```

8) Why does Playwright use BrowserContext instead of launching a new browser for every test?

Answer:

Launching a browser is expensive and slow.

BrowserContext provides a lightweight, isolated session (like incognito) that gives test isolation without restarting the browser, making tests faster and more scalable.

9) Can two pages share cookies?

Answer:

- Yes, if they belong to the same BrowserContext
- No, if they belong to different BrowserContexts

10) Can two BrowserContexts share data?

Answer:

- No.

Each BrowserContext is fully isolated — cookies, localStorage, cache are not shared.

11) How do you handle multiple tabs in Playwright?

Answer:

```
const pages = context.pages();
const secondTab = pages[1];
await secondTab.bringToFront();
```

12) Difference between browser.newPage() and context.newPage()?

Answer:

Method	Result
`browser.newPage()`	Creates **new context + page**
`context.newPage()`	Creates **page inside existing context**

13) How does Playwright achieve test isolation?

Answer:

By creating a new BrowserContext per test, Playwright ensures:

Clean cookies

No shared localStorage

Reliable parallel execution

14) Can you run tests in parallel using one browser?

Answer:

- Yes.

One browser → multiple contexts → parallel tests.

15) How do you wait for an element to be visible?

Answer:

```
await page.locator('#login').waitFor({ state: 'visible' });
```

16) How do you handle alerts / dialogs?

Answer:

```
page.on('dialog', async dialog => {
  console.log(dialog.message());
  await dialog.accept(); // or dialog.dismiss()
});
```

17) How do you upload a file?

Answer:

```
await page.setInputFiles('#upload', 'tests/data/file.pdf');
```

18) How do you take a screenshot?

Answer:

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

19) How do you handle frames (iframe)?

Answer:

```
const frame = page.frameLocator('#frameId');
await frame.locator('#btn').click();
```

20) How do you mock API response?

Answer:

```
await page.route('**/api/users', route =>
  route.fulfill({
    status: 200,
    body: JSON.stringify({ name: 'Shrinivas' })
  })
);
```

21) What is test.use()?

Answer:

Used to apply settings per test or per file
test.use({ browserName: 'chromium' });

22) Difference between Locator and ElementHandle?

Answer:

Locator	ElementHandle
Auto-wait	✗ No auto-wait
Recommended	✗ Not recommended
Lazy evaluation	Immediate

23) How do you get text from element?

Answer:

```
const text = await page.locator('#msg').textContent();
```

24) Common assertions for elements?

Answer:

```
await expect(locator).toBeVisible();
await expect(locator).toBeEnabled();
await expect(locator).toBeChecked();
await expect(locator).toHaveText('Login');
```

25) How to take screenshots on failure?

Answer:

```
Configured in playwright.config.ts
use: {
  screenshot: 'only-on-failure',
}
```

26)What are fixtures in Playwright?

Definition:

A fixture in Playwright is a pre-configured resource that is automatically created before a test and cleaned up after the test.

Key points:

Helps reuse setup code.

Ensures test isolation (each test gets its own instance).

Supports automatic cleanup

Playwright comes with several built-in fixtures that you can use directly in your tests:

page,browser,browserContext,request,workInfo

27)What are locators in Playwright?

Answer:

In Playwright, locators are a powerful way to identify and interact with elements on a web page.

```
const element = await page.locator(selector);
```

28)Explain the difference between locator.click() and page.click()?

Answer:

Use locator.click() instead of page.click() because locators provide better auto-waiting, retry logic, and stability for modern dynamic web applications.

locator.click()->first declare locator and then click (auto-waits, retries, and handles dynamic DOM)

page.click()->passing locator while clicking (is a one-time action and can be flaky)

29) How do you launch a browser in Playwright?

Answer:

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

30) What is auto-waiting in Playwright?

Answer:

Playwright automatically waits for:

Element to appear
Visibility
Enablement
Stability

31)What is test.describe()?

Answer:

Used to group related test cases.

```
test.describe('Login Tests', () => {  
  test('valid login', async () => {});  
});
```

32)What is expect in Playwright?

Answer:

expect is an assertion library used to validate test results.

```
await expect(page).toHaveTitle('Home');
```

33)What is beforeEach and afterEach?

Answer:

beforeEach runs before every test

afterEach runs after every test

34)What is BrowserContext? Why is it important?

Answer:

BrowserContext is an isolated browser session similar to an incognito window.

Importance:

Separate cookies & cache

Enables parallel execution

Improves security testing

35)How do you handle file upload?

Answer:

```
await page.setInputFiles('#upload', 'file.pdf');
```

36)What is an ElementHandle in Playwright?

Answer:

In Playwright, an ElementHandle is a reference (handle) to a specific DOM element on the page.

It represents one particular element at a specific point in time.

```
const elementHandle = await page.$('#username'); // page.$('#username') returns an ElementHandle
```

```
await elementHandle?.fill('Shrinivas');
```

40)Difference between Locator and ElementHandle?

Answer:

Locator	ElementHandle
---------	---------------

Auto-waits	✗ No
------------	------

Retry logic	✗ No
-------------	------

DOM-safe	✗ No
----------	------

Recommended	✗ No
-------------	------

41)How do you handle file download?

Answer:

```
const download = await page.waitForEvent('download');
```

```
await download.saveAs('report.pdf');
```

42)How do you handle multiple tabs or windows?

Answer:

```
test('Handle popup window', async ({ page }) => {
  await page.goto('https://gauravkhurana.in/test-automation-play/')
  await page.locator("//button[@id='radix-:r0:-trigger-advanced']").click()
  const context = page.context();
  const [newTab] = await Promise.all([page.waitForEvent('popup'), page.click("//button[normalize-space()='Open New Window']")]);
  console.log(await newTab.title())
})
```

43)How do you debug Playwright tests?

Answer:

```
npx playwright test --debug
```

44)What is Trace Viewer?

Answer:

Trace Viewer records:

DOM snapshots

Network logs

Screenshots

Actions timeline

Helps in root cause analysis.

45)How do you integrate Allure reports?

Answer:

```
npm install -D allure-playwright
```

46)How do you handle environment variables?

Answer:

Using .env file:

```
process.env.BASE_URL
```

47)How do you retry failed tests?

Answer:

retries: 2

48)What is soft assertion in Playwright?

Answer:

Soft assertions allow test to continue even if assertion fails.

```
await expect.soft(locator).toBeVisible();
```

49)How do you handle network mocking?

Answer:

```
await page.route('**/api/**', route =>
```

```
  route.fulfill({ status: 200, body: JSON.stringify(mockData) })
```

```
);
```

50)What is the default timeout in Playwright?

Answer:

Test timeout: 30 seconds

Action timeout: 0 (relies on auto-wait)

Expect timeout: 5 seconds

51)Difference between timeout and expect.timeout?

Answer:

timeout expect.timeout

Test-level Assertion-level

Stops entire test Only waits for assertion

Configurable globally Configurable per expect

52)What is locator.all()?

Answer:

Returns all matching elements as an array of locators.

```
const items = await page.locator('.row').all();
```

53)What is locator.first() / locator.last()?

Answer:

Used to resolve strict mode when multiple elements exist.

```
locator.first()
```

```
locator.last()
```

54)How do you handle Alerts & Dialogs?

Answer:

Using dialog event.

```
page.on('dialog', dialog => dialog.accept());
```

55)How do you handle frames / iframes?

Answer:

```
Using frameLocator().
```

```
await page.frameLocator('#frame').locator('#btn').click();
```

56)Explain the internal architecture of Playwright

Answer:

Playwright uses a client–server architecture.

Test code runs in Node.js

Playwright communicates with browsers using WebSocket

Each browser runs in a separate process

Actions are sent as protocol commands

Supports Chromium, Firefox, WebKit via a unified API

57)When do you use multiple browser contexts?

Answer:

Multiple browser contexts are used when:

Testing multiple users in parallel

Running isolated sessions

Verifying role-based access

Preventing cookie/session conflicts

Each context behaves like an incognito window.

58)What is Playwright Test Generator?

Answer:

A tool that records user actions and generates Playwright test code.

```
npx playwright codegen
```

59)Difference between Promise and Promise.all()

Answer:

Promise:

Promise represents one asynchronous operation that will complete in the future.

States of Promise:

Pending

Fulfilled

Rejected

Example

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Data loaded');
  }, 1000);
});
```

```
promise.then(result => console.log(result));
```

Promise.all():

Promise.all() is used to run multiple promises in parallel and wait until all of them are resolved.

Example

```
const p1 = Promise.resolve(10);
const p2 = Promise.resolve(20);
const p3 = Promise.resolve(30);
const result = await Promise.all([p1, p2, p3]);
console.log(result);
```

60)Test randomly fails — how do you debug Playwright tests?

Approach:

Enable trace, video, screenshots

Run in headed mode

Use DEBUG=pw:api

Identify missing waits / wrong locator strategy

61)Login test takes time — how do you speed it up?

Answer:

Best Practice: Storage State

```
// global-setup.ts
```

```
await page.goto('/login');
```

```
await page.fill('#user', 'admin');
```

```
await page.fill('#pass', 'pwd');
```

```
await page.click('#login');
```

```
await page.context().storageState({ path: 'auth.json' });
```

62)How do you handle flaky tests?

Answer:

Use auto-waiting locators

Replace waitForTimeout

Add retries only for flaky tests

63)How do you download a file & verify contents?

Answer:

```
const [download] = await Promise.all([
page.waitForEvent('download'),
page.click('#download')
]);
```

```
const path = await download.path();
```

Verify content:

```
import * as fs from 'fs';
```

```
expect(fs.readFileSync(path!).toString()).toContain('Invoice');
```

64)How do you delete test artifacts after test completion?

Answer:

```
test.afterEach(() => {
fs.rmSync('temp.txt', { force: true });
});
```

65)How do you pass test data from Excel / JSON?

JSON

Answer:

```
import data from './data/users.json';
await page.fill('#user', data.username);
import ExcelJS from 'exceljs';
const workbook = new ExcelJS.Workbook();
await workbook.xlsx.readFile('data.xlsx');
```

66)How do you wait for network calls?

Answer:

```
await page.waitForResponse(resp =>
resp.url().includes('/api/order') && resp.status() === 200
);
```

67)How to handle dynamic elements?

Answer:

Text-based locators

Regex

68)Types of waits?

Answer:

Auto wait (default)

Explicit (toBeVisible)

Network waits

Load state

69)playwright.config.ts purpose?

Answer:

- Browser config
- Parallel execution
- Retry
- Reporters
- Env setup

70)What is StorageState?

Answer:

Saves cookies & localStorage to reuse authenticated session.

71)Flaky tests — how do you handle?

Answer:

- Remove hard waits
- Use proper locators
- Retry logic
- Trace viewer

72)OTP automation?

Answer:

handle via API OTP

73)What happens if locator matches multiple elements?

Answer:

Playwright throws strict mode violation

74)How do you disable strict mode?

Answer:

Not recommended, but possible:

```
page.locator('button').first().click();
```

75)Difference between waitForSelector and locator?

Answer:

waitForSelector	locator
Manual wait	Auto-wait
Returns element	Action-oriented

76)When does waitForNavigation fail?

Answer:

When navigation happens before wait is registered

Solution:

```
await Promise.all([
  page.waitForNavigation(),
  page.click('#submit')
]);
```

77)networkidle vs load vs domcontentloaded?

Answer:

State	Meaning
domcontentloaded	HTML loaded
load	CSS + images
networkidle	No network calls

78)How do you handle new tab?

Answer:

```
const [newPage] = await Promise.all([
  context.waitForEvent('page'),
  page.click('#open')
]);
```

79)Can Playwright read DB?

Answer:

Not directly. Use Node DB client.

80)How do you share API context across tests?

Answer:

```
const apiContext = await request.newContext();
```

81)Where do you write reusable logic?

Answer:

```
utils/  
fixtures/  
helpers/
```

82)Playwright commands

Answer:

```
npx playwright test -> Runs all tests in headless mode (default).  
npx playwright test --last-failed -> Run Failed Tests Only  
npx playwright test --ui -> Best for exploring, debugging, watching steps visually  
npx playwright test --debug -> Test pauses automatically on failure  
npx playwright test --headed -> Headed Mode  
npx playwright codegen -> Codegen (Record Tests) or npx playwright codegen https://example.com  
npx playwright test --headed --project=chromium -> Headed + specific browser  
npx playwright test --debug --headed -> Headed + Debug  
npx playwright test login.spec.ts -> Run Specific Test File  
npx playwright test login.spec.ts dashboard.spec.ts -> Multiple files  
npx playwright test -g "should login successfully" -> Run Specific Test (by title)  
npx playwright test -g @smoke -> Run Tests with Tags (@smoke, @regression)  
npx playwright test --workers=4 -> Run Tests in Parallel  
ENV=qa npx playwright test -> Run Tests in Specific Environment  
ENV=uat npx playwright test -> Run Tests in Specific Environment  
npx playwright test --trace on -> npx playwright test --trace on  
npx playwright show-trace trace.zip -> View trace
```

83)what is use of playwright.config.ts

Answer:

- playwright.config.ts is the main configuration file for Playwright Test.
- Central place to manage test settings
- What can we configure
 - Browser configuration (Chromium, Firefox, WebKit)
 - Headed / headless mode
 - Base URL (QA, UAT, Prod)
 - Timeouts
 - Parallel execution
 - Retries
 - Reporters (HTML, Allure)
 - Screenshots, videos, traces
 - Test directory & patterns

84)what is package.json?

Answer:

- package.json is the Node.js project configuration file.
- It manages dependencies, scripts, and project metadata.
use:
 - Project name & version
 - Dependencies
 - Scripts (commands to run tests)

85)What is Synchronous and Asynchronous?

Answer:

Synchronous: In synchronous execution, tasks run one after another. Blocking in nature, Execution happens step-by-step
eg: Standing in a queue at a bank

Asynchronous: In asynchronous execution, tasks do not block each other, Non-blocking, program continues without waiting.

86)Why we use promise and what is async and await?

Answer:

use- To perform sequential task,eg- to wait for database data.

async- Make function to return promise for every step.(Can check whether the function return promise or not by hover over it)

await- pause execution until promise is resolved or rejected.

87)Explain Playwright working

Answer:

Playwright is using web-socket protocol which made single connection and do all operations within than connection not like selenium(selenium use new API for every action on web)

JS/TS---->Web-Socket---->Real Browser

88)What are Playwright Locators?

Answer:

Playwright's inbuilt Locator has an auto-wait mechanism

- 1) getByRole() -> Accessible roles (best practice)
- 2) getByText() -> Visible text
- 3) getByLabel() -> Form labels
- 4) getByPlaceholder() -> Input placeholders
- 5) getByAltText() -> Image alt text
- 6) getTitle() -> Title attribute
- 7) getByTestId() -> Test IDs

89)What is Assertion?

Answer:

An assertion is a validation used in testing to verify expected vs actual result.

Hard Assertion: A Hard Assertion stops the test execution immediately if the assertion fails.

await expect(page).toHaveTitle('Login Page'); // Hard assertion

await page.click('#submit'); // ✗ This line won't run if title check fails

Soft Assertion: A Soft Assertion allows the test to continue execution even if assertion fails.

await expect.soft(page.locator('#password')).toBeVisible();

await expect.soft(page.locator('#loginBtn')).toBeEnabled();

90)Screenshot

Answer:

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

```
await page.locator('#loginBtn').screenshot({ path: 'button.png' }); //locator
```

options-

```
use: {  
  screenshot: 'only-on-failure',  
  screenshot: 'off',  
  screenshot: 'on',  
  screenshot: 'only-on-failure'  
}
```

91)Record Video

Answer: use: {

```
video: 'off', // No video recording
```

```
video: 'on', // Record video for every test
```

```
video: 'retain-on-failure', // Record only when test fails ✅ (most used)
```

```
video: 'on-first-retry' // Record video only on first retry
```

```
}
```

92)Trace Options

Answer: use: {

```
trace: 'off', // Do not record trace
```

```
trace: 'on', // Record trace for every test
```

```
trace: 'retain-on-failure', // Record trace only when test fails ✅ (most used)
```

```
trace: 'on-first-retry' // Record trace only on first retry
```

```
}
```

93)Grouping test cases

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

```
test.describe('group1', () => {
```

```
  test('Valid Login', async ({ page }) => {
```

```
    // test steps
```

```
  });
```

```
  test('Invalid Login', async ({ page }) => {
```

```
// test steps
});
});
To run : npx playwright test --grep group1 –headed
```

94)Hooks

Answer:

```
// Runs once before all tests in this describe
test.beforeAll(async () => {
  console.log('Browser setup or global init here');
});

// Runs before every test
test.beforeEach(async ({ page }) => {
  await page.goto('https://example.com/login'); // precondition
});

// Runs after every test
test.afterEach(async ({ page }, testInfo) => {
  // Take screenshot on failure
  if (testInfo.status !== testInfo.expectedStatus) {
    await page.screenshot({ path: `failure-${testInfo.title}.png` });
  }
});

// Runs once after all tests
test.afterAll(async () => {
  console.log('Global cleanup here');
});
```

95)Annotations

Answer:

```
only - test.only('Run only this test', async ({ page }) => {});
skip - test.skip('Skip this test', async ({ page }) => {});
fixme-test.fixme('Fix this later', async ({ page }) => {});
slow - test.slow('This test is slow', async ({ page }) => {});
```

96)Parallel execution

Answer: Set fullyParallel as true and workers:4 or whatever

```
import { defineConfig } from '@playwright/test';
export default defineConfig({
  fullyParallel: true, // enables full parallel execution
  workers: 4, // number of parallel threads
  use: [
    {
      headless: true,
      screenshot: 'only-on-failure',
      video: 'retain-on-failure',
      trace: 'retain-on-failure',
    },
  ],
});
```

97)Reporter

Answer:

Reporters in Playwright are used to generate test execution results, which can be viewed in terminal, HTML, or external tools like Allure. They help analyze test results, failures, and logs.

inbuilt- list, dot, line, html.

98)Visual Testing in Playwright

Answer:

Visual testing compares the current UI against a baseline screenshot to detect unintended visual changes.

```
import { test, expect } from '@playwright/test';
test('Homepage visual test', async ({ page }) => {
  await page.goto('https://example.com');
  // Take a screenshot and compare with baseline
  await expect(page).toHaveScreenshot('homepage.png');
});
```

99)Accessibility Testing in Playwright

Answer:

Accessibility (a11y) testing ensures that a web application is usable by people with disabilities and follows accessibility standards like WCAG.

```
import { test, expect } from '@playwright/test';
test('Check accessibility of homepage', async ({ page }) => {
  await page.goto('https://example.com');
  // Run built-in accessibility snapshot
  const accessibilitySnapshot = await page.accessibility.snapshot();
  console.log(accessibilitySnapshot); // Analyze the snapshot for issues
});
```

100)Add a Custom Script in package.json

```
{
  "scripts": {
    "test-login": "npx playwright test tests/login --project=chromium",
    "test-dashboard": "npx playwright test tests/dashboard --project=firefox",
    "test-all": "npx playwright test"
  }
}
```

To run : npm run test-login, npm run test-all

101)Use of ?? (Nullish Coalescing Operator)

Answer: ?? returns the right-hand value only when the left-hand value is null or undefined.

```
const name = undefined;
const userName = name ?? 'Guest';
console.log(userName); // print Guest
```

102)What is a GitHub Workflow?

Answer:

A GitHub workflow is an automated CI/CD process defined using GitHub Actions that runs jobs like build, test, or deploy when specific events occur (push, pull request, schedule, etc.).

103)How Playwright Handles Shadow DOM?

Answer: Shadow DOM is used to encapsulate elements inside web components so their HTML/CSS does not interfere with the rest of the page.

Playwright automatically handles open Shadow DOM, so you can locate elements just like normal DOM elements.
Can't handle with xpath, can directly handle by css selector as normal element in playwright and in selenium this is handled by javascript

104)What is Viewport in Playwright?

Answer: Viewport in Playwright defines the visible area of the browser page where the web application is rendered, simulating different screen sizes like desktop, tablet, or mobile.

```
use: {
  viewport: { width: 1920, height: 1080 }, }
```

105)Handling SSL Certificates in Playwright?

Answer: An SSL certificate is a digital certificate that encrypts communication between a user's browser and a web server, ensuring data security and trust.

```
use: {
  ignoreHTTPSErrors: true,
}
```

106)How to handle cookie?

Answer: Cookies are managed at the BrowserContext level, not directly on page.

```
const context = page.context();
const cookies = await context.cookies(); //get all cookies
const cookies = await context.cookies('https://example.com'); //Get Cookies for a Specific URL
await context.addCookies([{name: 'session_id',value: '12345',domain: 'example.com',path: '/'},]); // add cookies
await context.clearCookies(); //clear all cookies
```

107)Keyboard actions

Answer: await page.keyboard.press('Enter'); // Press a Key
await page.keyboard.type('Hello Playwright'); // Type Text
await page.keyboard.down('Shift'); // Key Down (Hold Key)

```
await page.keyboard.up('Shift'); // Key Up (Release Key)
await page.keyboard.press('Control+A');
```

108) Mouse Events

```
Answer: await page.mouse.move(100, 200); // to move
await page.mouse.down(); //Mouse Down (Press Button)
await page.mouse.up(); //Mouse Up (Release Button)
await page.mouse.click(200, 300); //Mouse Click
await page.locator('#menu').hover(); //Hover
await page.mouse.wheel(0, 500); // Scroll Using Mouse Wheel
await page.mouse.click(300, 400, { button: 'right' }); //Right Click (Context Click)
```

109) Does Playwright need explicit waits?

Answer:

No, Playwright does not usually need explicit waits.

Playwright has a built-in auto-wait mechanism. It automatically waits for:

- Elements to appear in the DOM
- Elements to be visible, enabled, and stable
- Navigation and network activity (when required)

Because of this, using `waitForSelector()` or `waitForTimeout()` is rarely needed

110) Difference between `waitForSelector`, `waitForLoadState`, and `timeout`

Answer:

`waitForSelector()` - Waits for a **specific element** to appear in the DOM eg. `await page.waitForSelector('#submit');`

`waitForLoadState()` - Waits for a **page-level load event**. eg. `await page.waitForLoadState('networkidle');`

- `load` → page fully loaded
- `domcontentloaded` → DOM ready

`Timeout` - Defines **how long Playwright should wait before failing**. Eg. `await page.click('#save', { timeout: 5000 });`

111) Why is `page.waitForTimeout()` not recommended?

Answer:

`page.waitForTimeout()` uses fixed waiting time (hard wait).

It makes tests slow and flaky because it does not depend

on application state. eg. `await page.waitForTimeout(5000); X`

113) Why are `getByRole` / `getByText` preferred?

Answer:

`getByRole()` and `getByText()` are preferred because they are user-centric, accessibility friendly, and less flaky.

They automatically wait for elements to be ready.

114) How does Playwright handle dynamic DOM updates?

Answer:

Playwright uses live locators to handle dynamic DOM updates.

Locators re-evaluate the DOM every time an action is performed and automatically wait for expected conditions.

115) What happens if a locator matches multiple elements?

Answer:

By default, Playwright runs in strict mode.

If a locator matches more than one element,

Playwright throws a strict mode violation error.

116) How do you force strict mode off?

Answer:

Strict mode can be bypassed by explicitly selecting a single element using `first()`, `last()`, or `nth()`.

eg.

```
await page.getText('Submit').first().click();
await page.getText('Submit').nth(1).click();
```

117) Difference between CSS selector and Playwright locator

Answer:

CSS selector directly finds elements in the DOM
but does not have auto-wait capability.
Playwright locator is a smart object that
automatically waits for elements to be visible,
enabled, and stable before performing actions.
eg.
CSS selector: await page.click('.btn-primary');
Locator: await page.getByRole('button', { name: 'Save' }).click();

118) When should you use nth(), first(), last()?

Answer:
Use nth(), first(), or last() when multiple elements
match the same locator and you want to target
a specific one.
eg.
await page.locator('.item').first().click();
await page.locator('.item').last().click();
await page.locator('.item').nth(2).click();

119) How do you handle elements that appear only on hover?

Answer:
Use hover() action before interacting with
elements that appear only on mouse hover.
eg.
await page.getText('Settings').hover();
await page.getText('Logout').click();
Playwright automatically waits for the hovered element to become visible.

120) How do you verify element count?

Answer:
Use toHaveCount() assertion to verify the number
of matching elements.
await expect(page.locator('.product')).toHaveLength(5);

121) How do you wait for an element to disappear?

Answer:
Use waitFor() with state 'hidden' or 'detached'
to wait for an element to disappear.
eg.
await page.getText('Loading...').waitFor({ state: 'hidden' });
hidden → element is invisible but present in DOM
detached → element is removed from DOM

122) What actions trigger Playwright auto-wait?

Answer:
Playwright auto-wait is triggered by actions
like click, fill, type, check, uncheck, and selectOption.
It waits for the element to be visible, enabled,
and stable before performing the action.
eg.
await page.getByRole('button', { name: 'Submit' }).click();

123) How does Playwright execute tests in parallel?

Answer:
Playwright runs tests in parallel using worker processes.
Each worker runs tests in an isolated browser context.
Parallel execution is controlled using workers in playwright.config.ts eg. workers: 4

124) Difference between workers and projects?

Answer:
Workers – Define how many tests run in parallel
at the same time.
Projects – Define different test environments like browsers, devices, or configurations.

eg.

Projects → Chromium, Firefox, WebKit

Workers → Number of parallel threads

125) What happens if tests share the same test data?

Answer:

If multiple tests share the same test data

during parallel execution, it can cause

data collision and flaky test failures.

eg. One test may modify or delete data used by another test.

126) How do you avoid data collision in parallel runs?

Answer:

Use unique test data for each test or worker.

Generate dynamic data using timestamp or faker.

Use separate users or accounts per test.

Use isolated browser contexts and storage state.

eg. const email = `user_\${Date.now()}@test.com`;

127) Can you run tests sequentially in Playwright?

Answer:

Yes, Playwright allows running tests sequentially.

You can disable parallel execution using `describe.serial()`

or by setting workers to 1.

eg.

```
test.describe.serial('Sequential tests', () => {
  test('Test 1', async ({ page }) => {});
  test('Test 2', async ({ page }) => {});
});
```

128) How do you limit workers for CI?

Answer:

In CI, workers should be limited to avoid

resource issues and flaky tests.

Workers can be limited in `playwright.config.ts`.

eg.

```
workers: process.env.CI ? 1 : undefined;
```

129) Does Playwright retry failed assertions?

Answer:

Yes, Playwright automatically retries assertions

until they pass or the timeout is reached.

eg.

```
await expect(page.getText('Success')).toBeVisible(); // The assertion keeps retrying until the condition is met.
```

130) Difference between auto-wait and explicit wait

Answer:

Auto-wait is built into Playwright actions and assertions

and waits for conditions automatically.

Explicit wait requires manually waiting using

`waitForSelector()` or `waitForTimeout()`.

Auto-wait makes tests faster and reliable,

while explicit waits can make tests flaky

131) Why does `expect(locator).toBeVisible()` retry automatically?

Answer:

Playwright assertions are designed to be resilient.

`expect(locator).toBeVisible()` keeps retrying

until the element becomes visible or timeout occurs. This supports dynamic and async web applications.

132) How does Playwright know when the page is “ready”?

Answer:

Playwright tracks browser lifecycle events

like `DOMContentLoaded`, `load`, and `network activity`.

It waits for navigation and auto-wait conditions based on the action being performed.

eg. `await page.waitForLoadState('networkidle');`

133) What is the default timeout hierarchy?

Answer:

Playwright follows a timeout hierarchy:

1) Assertion timeout (default 5s)

2) Action timeout

3) Navigation timeout

4) Global test timeout (default 30s)

Timeouts can be overridden at test,action, or config level.

134) How do you pass environment variables in Playwright?

Answer:

Environment variables can be passed using command line, .env file, or CI configuration.

eg. (Command line)

```
npx playwright test --env=QA
```

eg. (Using process.env)

```
const env = process.env.ENV;
```

135) Difference between .env and playwright.config.ts

Answer:

.env file is used to store environment-specific values like URLs, usernames, and passwords.

playwright.config.ts is used to configure Playwright behavior like browsers, workers, baseURL, timeouts, and reporters.

.env → Data configuration

config → Framework configuration

136) How do you manage different base URLs?

Answer:

Different base URLs are managed using environment variables in playwright.config.ts.

eg.

```
baseURL: process.env.ENV === 'QA'
```

```
? 'https://qa.example.com'
```

```
: 'https://prod.example.com';
```

137) Difference between frameLocator and contentFrame

Answer:

frameLocator is used to locate elements

inside an iframe directly using Playwright locators.

contentFrame returns a Frame object from an iframe element.

eg.

```
frameLocator: page.frameLocator('#frame').getByText('Submit');
```

```
contentFrame: const frame = await iframe.contentFrame();
```

138) How do you handle popup windows?

Answer:

Popup windows are handled using the

```
page.on('popup') or context.waitForEvent('page').
```

```
eg. const popup = await page.waitForEvent('popup');
```

139) How do you wait for a new tab?

Answer:

Use context.waitForEvent('page') combined with the action that opens the new tab.

eg.

```
const [newTab] = await Promise.all([
  page.context().waitForEvent('page'),
  page.click('#openTab')
]);
```

140) How do you switch between tabs?

Answer:

Use browser context to get all open pages and switch using page reference.

```
eg. const pages = page.context().pages();
```

```
await pages[1].bringToFront();
```

141) How do you close child windows only?

Answer:

Identify the parent page first, then
close all other pages in the context.

```
eg. const pages = page.context().pages();  
for (const p of pages) {  
if (p !== page) {  
await p.close(); } };
```

142) How do you test browser back/forward navigation?

Answer:

Use page.goBack() and page.goForward()
to test browser navigation.

```
eg. await page.goBack();  
await page.goForward();
```

143) How do you override config per project?

Answer:

Config can be overridden per project
inside playwright.config.ts using use property.

eg.

```
projects: [  
{  
name: 'Chromium',  
use: { browserName: 'chromium' }  
},  
{  
name: 'Firefox',  
use: { browserName: 'firefox' }  
}  
];
```

144) How do you run tests conditionally?

Answer:

Tests can be run conditionally using
if conditions or environment variables.

```
eg. if (process.env.ENV === 'QA') {  
test('Run only in QA', async () => {});  
}
```

145) How do you skip tests dynamically?

Answer:

Use test.skip() with a condition
to skip tests dynamically.

```
eg. test.skip(process.env.ENV === 'PROD', 'Skipping in PROD');
```

146) How do you intercept network requests?

Answer:

Network requests are intercepted using page.route().

eg.

```
await page.route('**/api/users', route => route.continue());
```

147) How do you modify request headers?

Answer:

Modify request headers inside page.route()
before continuing the request.

eg.

```
await page.route('**/*', route => {  
const headers = {  
...route.request().headers(),  
Authorization: 'Bearer token'  
};
```

```
route.continue({ headers });
});
```

148) How do you mock a failed API response?

Answer:

Mock failed API response using route.fulfill()
with error status code.

eg.

```
await page.route('**/api/login', route => {
  route.fulfill({
    status: 500,
    body: 'Internal Server Error'
  });
});
```

149) How do you debug a test running in CI?

Answer:

Enable tracing, screenshots, and videos to debug CI failures.

eg.

```
use: {
  trace: 'on-first-retry',
  screenshot: 'only-on-failure',
  video: 'retain-on-failure'
};
```

Download trace and open using:npx playwright show-trace trace.zip

150) What is the difference between trace and video?

Answer:

Trace is a detailed execution record that includes DOM snapshots, network requests, console logs, and Playwright actions.

Video is a screen recording of the test execution showing only visual flow.

→Deep debugging Video → Visual playback

151) What is visual regression testing?

Answer:

Visual regression testing compares screenshots to detect unintended UI changes.

It ensures UI consistency across builds and environments.

eg. await expect(page).toHaveScreenshot();

152) How does toHaveScreenshot() work?

Answer:

toHaveScreenshot() captures a screenshot and compares it with a baseline image.

If the difference exceeds the threshold, the test fails.

eg. await expect(page.locator('.header')).toHaveScreenshot();

153) How do you handle permissions (camera, location)?

Answer:

Permissions are handled using browser context configuration.

eg.

```
const context = await browser.newContext({
  permissions: ['geolocation', 'camera'],
  geolocation: { latitude: 18.52, longitude: 73.85 }
});
```

154) How do you clear cache/session?

Answer:

Cache and session are cleared by creating a new browser context.

eg.

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

Each context is isolated and has fresh session data.

155) How do you integrate Playwright with GitHub Actions?

Answer:

Use Playwright GitHub Action workflow with Node.js setup and Playwright install.

eg. - npm ci

- npx playwright install

- npx playwright test

156) What challenges occur when running Playwright in CI?

Answer:

Common CI challenges include:

- Limited resources
- Network latency
- Timing issues
- Environment differences

These can cause flaky tests if not handled properly.

157) How do you generate reports in CI?

Answer:

Reports are generated using Playwright reporters like HTML or Allure.

eg.

```
reporter: [['html', { open: 'never' }]];
```

CI uploads the report as an artifact.

158) How do you store test artifacts?

Answer:

Test artifacts like traces, screenshots, videos, and reports are stored using CI artifact storage.

eg. Upload folders:

- playwright-report
- test-results
- traces

159) How do you design a scalable Playwright framework?

Answer:

A scalable Playwright framework uses:

- Page Object Model (POM)
- Reusable fixtures
- Environment-based config
- Utility and helper layers
- Parallel execution support

This improves maintainability, readability, and supports large test suites.

160) How do you block third-party scripts?

Answer:

Third-party scripts can be blocked using page.route() and aborting matching requests.

eg. await page.route('**/analytics/**', route => route.abort()); This improves test speed and stability.

161) How do you test HTTPS & insecure content?

Answer:

Use ignoreHTTPSErrors option in browser context to handle HTTPS and insecure certificates.

```
eg. const context = await browser.newContext({  
  ignoreHTTPSErrors: true  
});
```

162) How do you simulate offline mode?

Answer:

Offline mode can be simulated using browser context network settings.

```
eg. await context.setOffline(true);
```

This helps test offline and network failure scenarios.

163) What problems does visual testing solve?

Answer:

Visual testing detects unintended UI changes

that functional tests cannot catch. It helps identify layout issues, CSS regressions, and broken UI components.

164) What is accessibility testing?

Answer:

Accessibility testing ensures applications are usable by people with disabilities.
It checks keyboard navigation, screen reader support, color contrast, and ARIA roles.

165) How do you test accessibility in Playwright?

Answer:

Accessibility testing is done using Playwright with axe-core integration.
eg. await expect(page).toPassA11y();
This detects accessibility violations automatically.

166) What is ARIA role?

Answer:

ARIA roles define the purpose of UI elements for assistive technologies. Playwright uses ARIA roles in getByRole() locators.
eg. button, textbox, link, checkbox

168) What logs are available in Playwright?

Answer:

Playwright provides multiple logs such as:
- Console logs
- Network logs
- Trace logs
- Browser logs

These help in debugging test failures.

169) How do you enable verbose logging?

Answer:

Enable verbose logging using DEBUG environment variable.

170) How do you wait for a specific API call?

Answer:

Use page.waitForResponse() to wait for a specific API call.
eg.
await page.waitForResponse(response =>
response.url().includes('/api/users') && response.status() === 200
);

171) Difference between route.fulfill() and route.continue()

Answer:

route.continue() allows the request to go to the real server (optionally with modified data).
route.fulfill() mocks the request and sends a custom response without hitting the server.
eg.

route.continue(); → real API call
route.fulfill(); → mocked API response

172) How do you test slow network scenarios?

Answer:

Slow network scenarios are tested by delaying network responses using route.fulfill()
or route.continue() with setTimeout.

eg.

```
await page.route('**/api/data', async route => {  
  await new Promise(r => setTimeout(r, 3000));  
  route.continue();  
});
```

173) Test passes locally but fails in CI — why?

Answer:

Common reasons include:

- Slower execution in CI
- Resource limitations
- Network instability
- Hard waits or timing issues
- Environment differences

Solution:

Use auto-waiting, proper assertions, retries, traces, and screenshots.

174) Application UI changes frequently — how do you stabilize tests?

Answer:

Use user-facing locators like `getByRole()` and `getByText()` instead of CSS or XPath.

Apply Page Object Model and avoid brittle selectors like dynamic IDs.

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

175) Tests are slow — how do you improve execution time?

Answer:

Improve execution time by:

- Running tests in parallel
- Blocking third-party scripts
- Reusing authentication with `storageState`
- Avoiding hard waits
- Using headless mode

176) Dropdown loads values from API — how do you validate?

Answer:

Intercept the API call and validate both

API response and UI dropdown values.

eg.

```
const response = await page.waitForResponse('**/api/options');
const data = await response.json();
await expect(page.locator('option')).toHaveLength(data.length);
```

177) CAPTCHA appears — how do you handle it?

Answer:

CAPTCHA should not be automated.

In test environments:

- Disable CAPTCHA
- Use test keys
- Bypass CAPTCHA via backend or config

Automation focuses on functionality,

not CAPTCHA validation.

178) Same element has different locators in environments — solution?

Answer:

Use environment-based locators or `data-testid` attributes.

Best practice is to use stable `data-testid` across all environments.

eg.

```
page.getByTestId('login-button');
```

179) How do you test responsive design?

Answer:

Responsive design is tested using multiple viewports and devices.

eg. `await page.setViewportSize({ width: 375, height: 812 });`

Or using projects: Desktop, Tablet, Mobile

180) Playwright Internals (Advanced)

Answer:

Playwright works by communicating directly with browser engines using WebSocket-based protocol. It controls Chromium, Firefox, and WebKit with the same API.

Key internals include:

- Browser → Context → Page hierarchy
- Auto-waiting mechanism
- Live locators
- Isolated browser contexts

This architecture makes Playwright fast, reliable, and cross-browser.

181) How does Playwright achieve auto-wait internally?

Answer:

Playwright achieves auto-wait by continuously polling the browser state before performing actions.

Before any action, it checks:

- Element is attached to DOM
 - Element is visible
 - Element is enabled
 - Element is stable (not animating)
- The action executes only when all conditions are met or timeout is reached.

182) What is the role of injected scripts in Playwright?

Answer:

Playwright injects helper scripts into the browser to observe DOM changes and user interactions.

These scripts help with:

- Auto-waiting
- Event listening
- Stable element detection
- Handling locators efficiently

This is key to Playwright's reliability.

183) Difference between DOM snapshot and live DOM

Answer:

DOM snapshot is a static capture of the DOM

at a specific point in time.

Live DOM is the actual, continuously changing DOM during application execution.

Playwright locators work with the live DOM, not static snapshots.

184) Why does Playwright not need WebDriver?

Answer:

Playwright communicates directly with browser engines using native browser protocols.

It does not rely on WebDriver, which adds latency and flakiness.

Direct communication makes Playwright faster and more stable than Selenium.

185) How does Playwright handle cross-origin iframes?

Answer:

Playwright can interact with cross-origin iframes using frameLocator(). It bypasses same-origin limitations by controlling the browser internally.

eg. page.frameLocator('#iframe').getByText('Submit');

186) What happens internally when you call locator.click()?

Answer:

Internally, Playwright:

- Resolves the locator to a live element
- Auto-waits for actionability checks
- Scrolls element into view
- Performs the click via browser protocol
- Waits for any resulting navigation or events

187) How does Playwright detect network idle?

Answer:

Playwright tracks active network requests in the browser.

Network idle is detected when no network requests are in flight for 500 milliseconds.

eg. await page.waitForLoadState('networkidle');

188) Why are ElementHandles discouraged?

Answer:

ElementHandles capture a reference to an element at a specific time.

If DOM changes, the handle may become stale.

Locators are preferred because they:

- Re-evaluate DOM automatically
- Support auto-waiting
- Are more reliable

189) How does Playwright isolate tests at process level?

Answer:

Playwright uses worker processes to isolate tests. Each worker runs in a separate Node.js process with its own browser instances.

Inside each worker, browser contexts provide further isolation for cookies, cache, and storage. This ensures true parallelism and test isolation.

190) What is strict mode?

Answer:

Strict mode is a Playwright behavior where a locator must resolve to exactly one element. If a locator matches zero or multiple elements, Playwright throws an error. This prevents ambiguous actions and flaky tests.

191) Why does Playwright fail if locator matches multiple elements?

Answer:

Playwright fails to avoid acting on the wrong element. Multiple matches create ambiguity. Strict mode forces testers to write precise and deterministic locators. This improves test reliability and stability.

192) How do you debug strict mode violations?

Answer:

Use .count() to check how many elements match the locator. Use Playwright Inspector or highlight matched elements. eg. await page.locator('text=Submit').count(); Then refine the locator using role, name, or filters.

193) How do you write resilient locators for dynamic UI?

Answer:

Use user-facing locators like getByRole(), getByLabel(), getByText(). Prefer stable attributes like data-testid. Avoid brittle selectors like dynamic IDs or indexes. eg. page.getByRole('button', { name: 'Save' });

194) Can you disable strict mode globally?

Answer:

No, strict mode cannot be disabled globally. Playwright enforces strict mode by design to ensure reliable and maintainable tests. However, you can bypass it explicitly using first(), last(), or nth().

195) Difference between .filter() and .nth()

Answer:

.filter() narrows down elements based on conditions like text, attributes, or locators. .nth() selects an element based on index position. eg. page.locator('button').filter({ hasText: 'Submit' }); page.locator('button').nth(0); .filter() is more resilient than .nth() for dynamic UIs.

196) How do you wait for animation to finish?

Answer:

Playwright waits for element stability automatically before performing actions. For explicit waiting, wait for element to become stable or for CSS property change. eg. await page.locator('.modal').waitFor({ state: 'visible' }); Playwright ensures the element is not animating before interaction.

197) How do you wait for page to be fully interactive?

Answer:

Use waitForLoadState('networkidle') to ensure page is fully loaded and interactive. eg. await page.waitForLoadState('networkidle'); This ensures DOM, scripts, and API calls are completed.

198) Difference between load, domcontentloaded, networkidle

Answer:

domcontentloaded → DOM is ready, HTML parsed but sub-resources may still load.
load → Page and all resources like images and stylesheets are fully loaded.
networkidle → No network requests for 500ms, ideal for SPA applications.

199) How do you wait for backend processing completion?

Answer:

Wait for a specific API response or UI state that confirms backend completion.

eg.

```
await page.waitForResponse(response =>
  response.url().includes('/api/process') && response.status() === 200
);
```

Or wait for success message on UI.

200) How do you wait for WebSocket events?

Answer:

Listen to WebSocket events using `page.on('websocket')` and wait for specific messages.

eg.

```
page.on('websocket', ws => {
  ws.on('framereceived', frame => {
    console.log(frame.payload);
  });
});
```

This helps validate real-time updates.

201) What data is isolated inside BrowserContext?

Answer:

BrowserContext provides complete isolation

between tests, similar to an incognito window.

The following data is isolated per BrowserContext:

- Cookies
- LocalStorage
- SessionStorage
- Cache
- IndexedDB
- Permissions (camera, location, notifications)
- Authentication state
- Service workers

This ensures tests do not affect each other even when running in parallel.

eg. Each test gets a fresh browser context by default in Playwright Test.

202) How does Playwright emulate mobile devices?

Answer:

Playwright emulates mobile devices using

predefined device descriptors.

Device emulation includes:

- Viewport size
- User agent
- Device scale factor
- Touch support

eg. `const { devices } = require('@playwright/test');`

`use: { ...devices['iPhone 13'] };`

203) Difference between emulation and real device testing

Answer:

Emulation simulates device behavior in desktop browsers.

Real device testing runs tests on physical devices.

Emulation → fast, cost-effective, CI-friendly

Real devices → accurate hardware behavior

Emulation cannot fully replicate real device performance and sensors.

204) How do you test responsive layouts?

Answer:

Responsive layouts are tested using multiple viewports or device projects.

eg. `await page.setViewportSize({ width: 375, height: 812 });`

Or define projects for desktop, tablet, and mobile in `playwright.config.ts`.

205) What are touch actions in Playwright?

Answer:

Touch actions simulate user gestures on touch-enabled devices.

Common touch actions include:

- tap
 - swipe
 - long press
- eg. await page.tap('#submit');

206) How do you test orientation change?

Answer:

Orientation change is tested by changing viewport dimensions dynamically.

eg.

```
// Portrait  
await page.setViewportSize({ width: 375, height: 812 });  
// Landscape  
await page.setViewportSize({ width: 812, height: 375 });
```

This validates layout behavior on orientation changes.

207) How do you mock geolocation?

Answer:

Geolocation is mocked using browser context configuration with latitude and longitude.

eg.

```
const context = await browser.newContext({  
geolocation: { latitude: 18.5204, longitude: 73.8567 },  
permissions: ['geolocation']  
});
```

This simulates user location for testing.

208) How do you test camera/microphone permission flows?

Answer:

Camera and microphone permissions are tested by granting permissions in browser context.

eg.

```
const context = await browser.newContext({  
permissions: ['camera', 'microphone']  
});
```

Then validate UI behavior when access is granted.

209) How do you deny permissions intentionally?

Answer:

Permissions can be denied by not granting them or explicitly clearing permissions.

eg.

```
const context = await browser.newContext({  
permissions: []  
});
```

This helps test permission-denied scenarios and error handling.

210) How do you test clipboard actions?

Answer:

Clipboard actions are tested using page.evaluate() and Playwright clipboard APIs.

eg.

```
await page.click('#copy');  
const text = await page.evaluate(() => navigator.clipboard.readText());
```

This validates copy-paste functionality.

211) How do you simulate time/date?

Answer:

Time and date are simulated using browser context option setFixedTime.

eg.

```
const context = await browser.newContext({  
timezoneld: 'Asia/Kolkata'  
});
```

Or mock Date object using page.addInitScript() for fixed timestamps.

This helps test date-dependent logic.

212) Difference between toHaveRole() and getByRole()

Answer:

getByRole() → Locator method used to find an element by its ARIA role and optionally its name or other attributes.
It returns a Locator that you can interact with (click, fill, etc.).

`toHaveRole()` → Assertion method used to **verify** that an element has a specific ARIA role.

213) How do you capture network timing metrics?

Answer:

Use `page.on('request')` and `page.on('response')` to record timestamps and calculate request duration.
eg.

```
page.on('request', req => console.log('Request started:', req.url()));
page.on('response', res => console.log('Response received:', res.url(), res.timing()));
```

214) How do you assert API response time?

Answer:

Use `page.waitForResponse()` combined with Date timestamps
or `performance.timing` metrics.

eg.

```
const start = Date.now();
await page.waitForResponse('**/api/data');
const duration = Date.now() - start;
expect(duration).toBeLessThan(2000); // assert < 2 seconds
```

215) How do you throttle network speed?

Answer:

Network speed is throttled using browser context settings.

eg.

```
const context = await browser.newContext({
  networkConditions: {
    offline: false,
    download: 50 * 1024, // 50 KB/s
    upload: 20 * 1024, // 20 KB/s
    latency: 100 // 100ms
  }
});
```

216) How do you mock backend downtime?

Answer:

Intercept API requests using `page.route()`
and respond with failure or timeout.

eg.

```
await page.route('**/api/data', route => {
  route.fulfill({ status: 500, body: 'Server Down' });
});
```

219) Difference between parallelism and sharding

Answer:

Parallelism → Multiple tests run simultaneously in separate workers on the same node.

Sharding → Test suite is **split across multiple nodes** to run parts of the suite on different machines.

Summary:

- Parallelism → speed up tests on single machine
- Sharding → distribute tests across CI machines

220) How do you handle flaky tests in CI?

Answer:

Strategies to handle flaky tests:

- Use Playwright retries (retries in config)
- Add proper auto-waiting and assertions
- Avoid hard-coded sleeps
- Use isolated test data per run
- Capture traces, screenshots, and logs for debugging

221) How do you secure secrets in CI?

Answer:

- Use CI secret management (GitHub Actions Secrets, GitLab Variables)
- Never hardcode API keys or passwords in test code
- Access secrets via environment variables

eg. `process.env.API_KEY`

222) How do you reduce CI execution cost?

Answer:

- Run tests in parallel only where needed
- Use headless browsers
- Block unnecessary third-party scripts
- Reuse authentication states (storageState)
- Run only impacted tests using --grep or tags
- Optimize test suite by removing redundant tests

223) How do you structure Playwright framework?

Answer: A scalable Playwright framework typically includes:

- tests/ → all test files organized by feature
- pages/ → Page Object Model classes for UI pages
- fixtures/ → reusable setup/teardown logic
- utils/ → helper functions (API, DB, data)
- config/ → playwright.config.ts and environment config
- reports/ → test reports, screenshots, traces

224) Where do you place locators?

Answer:

Locators are placed inside Page Object classes
for reusability and maintainability.

eg.

```
```ts
class LoginPage {
 readonly username = page.getByLabel('Username');
 readonly password = page.getByLabel('Password');
 readonly loginButton = page.getByRole('button', { name: 'Login' });
}
```

**225) How do you manage test data?**

Answer:

Test data is managed using:

- JSON, CSV, or Excel files
- Environment-specific config files
- Faker library for dynamic data
- Database seed scripts for known state

**226) Page Object Model vs Fixtures – which is better?**

Answer:

- Page Object Model (POM) → Organizes locators and actions per page
- Fixtures → Handle setup/teardown, login, context reuse

**227) How do you version control test data?**

Answer:

- Store static test data in the repo (JSON, CSV, YAML)
- Use branch-specific or environment-specific test data files
- For dynamic/generated data, log seed scripts and track versions in Git
- Avoid committing sensitive data; use environment variables instead

**229) How do you capture application state on failure?**

Answer:

Playwright can capture screenshots, videos, and traces automatically:

```
ts use: { screenshot: 'only-on-failure', video: 'retain-on-failure', trace: 'on-first-retry' }
```

**230) How do you handle unexpected popups?**

Answer:

Use event listeners to detect and handle popups:

```
ts page.on('dialog', async dialog => { console.log(dialog.message()); await dialog.dismiss(); // or dialog.accept() });
```

**231) How do you recover from browser crash?**

Answer:

- Use isolated BrowserContext per test
- Wrap critical tests in try/catch and relaunch browser if it crashes
- Enable retries to rerun tests after a crash

- Capture trace and logs to debug the cause

Example:

```
```ts
try {
  await page.goto(url);
} catch (error) {
  await browser.close();
  browser = await chromium.launch();
}
```

232) How do you continue test execution after failure?

Answer:

- Use test.step() to isolate steps and prevent complete failure
- Use retries for failed tests instead of halting suite
- Handle exceptions gracefully using try/catch inside tests
- Run independent tests in parallel so one failure does not block others

233) Describe a complex Playwright issue you solved.

Answer:

One complex issue I solved was flaky failures caused by dynamic DOM updates in a SPA application.

Elements were re-rendered after API responses, causing ElementHandle-based logic to fail.

Solution:

- Replaced ElementHandles with Playwright locators
- Used auto-waiting assertions like toBeVisible()
- Waited for specific API responses instead of timeouts

Result:

Test stability improved significantly
and flaky failures were eliminated.

234) Biggest challenge while migrating from Selenium to Playwright?

Answer:

The biggest challenge was changing the mindset from explicit waits to auto-waiting.

In Selenium, we relied heavily on:

- WebDriverWait
- Thread.sleep

In Playwright, we had to:

- Trust auto-waiting
- Use user-centric locators (getByRole)
- Remove unnecessary waits

235) How did Playwright improve your test stability?

Answer:

Playwright improved stability through:

- Built-in auto-waiting
- Strict mode enforcement
- Live locators instead of stale elements
- Isolated browser contexts

236) What feature of Playwright impressed you most?

Answer:

The most impressive feature is auto-waiting combined with strict locators.

It eliminates most synchronization issues without writing extra code.

Additionally, trace viewer is extremely powerful for debugging CI failures.

237) What would you improve in Playwright?

Answer:

If I had to improve something:

- Better built-in support for CAPTCHA handling in test environments
- More native network throttling controls
- Easier visual diff configuration for dynamic UIs

238) Can Playwright tests run without UI?

Answer:

Yes, Playwright tests can run without UI using headless mode.

Headless mode runs the browser in background without opening a visible window.

eg. use: { headless: true }

This is commonly used in CI pipelines for faster and resource-efficient execution.

240) When NOT to use Playwright?

Answer:

Playwright should not be used when:

- Only backend or API testing is required
- Testing legacy browsers not supported by Playwright
- Heavy real-device hardware validation is needed

241) How does Playwright handle memory leaks?

Answer:

Playwright minimizes memory leaks by:

- Using isolated browser contexts per test
- Automatically closing pages and contexts
- Releasing handles and listeners after test completion

242) What happens if Playwright server crashes?

Answer:

If the Playwright browser server crashes:

- The current test fails
- Playwright reports the error
- Retries may rerun the test (if configured)

243) Difference between textContent and innerText

Answer:

textContent returns all text inside an element, including hidden elements and line breaks. Faster

It does NOT consider CSS styles like display:none or visibility:hidden.

innerText returns only the visible text that is rendered on the UI. Slower

244) Difference between page.on() and page.once()

Answer:

page.on() registers an event listener that listens to the event every time it occurs.

eg. page.on('dialog', dialog => dialog.accept()); The handler will be executed for all future dialog events.

page.once() registers an event listener that is executed only one time

eg. page.once('popup', popup => console.log(popup.url())); After the first event is handled, the listener is automatically removed

245) How to get current test name in Playwright?

Answer: Use testInfo.title to get the test name.

246) How to improve test performance and optimize login functionality?

Answer:

* Optimize Login Tests in Automation

- Avoid unnecessary UI logins for every test

- Instead of logging in before every test, reuse authentication tokens or session cookies.

In Playwright

```
// Save login state
const storage = await page.context().storageState({ path: 'state.json' });
// Reuse login state in subsequent tests
const context = await browser.newContext({ storageState: 'state.json' });
const page = await context.newPage();
```

- Use API login instead of UI login

Perform login via API call and set authentication in localStorage/sessionStorage.

This avoids slow UI interactions.

```
await page.request.post('/login', {
  data: { email, password }
});
await page.goto('/dashboard');
```

* Login once per test suite

- Use beforeEach hooks to login once and share the session across multiple tests.

```
test.beforeEach(async ({ browser }) => {
  const context = await browser.newContext();
```

```
const page = await context.newPage();
await page.goto('/login');
await page.fill('#email', 'user@test.com');
await page.fill('#password', 'password');
await page.click('#submit');
await context.storageState({ path: 'state.json' });
});
```

2 Reduce Test Execution Time

- Run tests in parallel
- Playwright supports test.describe.parallel or --workers to execute multiple tests concurrently.
- Use headless mode for CI
- Headless browser execution is faster than full browser mode.
- Use smart waits instead of fixed waitForTimeout
- Use locators with await page.locator(...).waitFor() or Playwright built-in auto-wait.
- Avoid repetitive setup/teardown
- Reuse browser contexts or global setup to avoid relaunching browser unnecessarily.

3 Reduce Flaky Tests and Improve Stability

- Use reliable selectors
- Prefer data-test-id over dynamic CSS/XPath.
- Avoid unnecessary page reloads
- Use network stubbing/mocking for API calls when possible, reducing dependency on server response time.

4 Optimize Backend/API During Login

- Minimize backend API calls during login.
- Cache static resources if possible.
- Reduce heavy redirects or unnecessary page loads.

247) WaitFor methods

Playwright Wait Methods – Complete List with Examples (TypeScript)

1 waitForTimeout

Description: Pause execution for a fixed amount of time (ms).

Example:

```
await page.waitForTimeout(3000); // wait for 3 seconds
```

❖ **Use case:** Debugging, animations (not recommended for real tests).

2 waitForSelector

Description: Wait until an element meets a specific state.

Syntax:

```
await page.waitForSelector(selector, { state?: 'attached' | 'detached' | 'visible' | 'hidden',
timeout?: number });
```

Example:

```
await page.waitForSelector('#login-button'); // attached in DOM
await page.waitForSelector('#login-button', { state: 'visible', timeout: 5000 });
await page.waitForSelector('#spinner', { state: 'hidden' }); // wait until spinner disappears
```

States:

- 'attached' → exists in DOM
- 'detached' → removed from DOM
- 'visible' → visible on UI
- 'hidden' → hidden or removed

3 waitForLoadState

Description: Wait for page navigation/load events.

Syntax:

```
await page.waitForLoadState(state?: 'load' | 'domcontentloaded' | 'networkidle', options?: {
timeout?: number });
```

Example:

```
await page.goto('https://example.com');
await page.waitForLoadState('load'); // full load
await page.waitForLoadState('domcontentloaded'); // DOM ready
await page.waitForLoadState('networkidle'); // no network requests for 500ms
```

4 waitForResponse

Description: Wait for a network response that matches a condition.

Example:

```
const response = await page.waitForResponse(response =>
  response.url().includes('/api/login') && response.status() === 200
);
console.log(await response.json());
```

✓ **Use case:** Wait for API completion before asserting UI.

5 waitForRequest

Description: Wait for a network request that matches a condition.

Example:

```
const request = await page.waitForRequest(req => req.url().includes('/api/login'));
console.log(request.method());
```

6 waitForEvent

Description: Wait for a specific event on page, context, or browser.

Example:

```
// Wait for a new popup/page
const [newPage] = await Promise.all([
  page.waitForEvent('popup'),
  page.click('#open-new-tab')
]);

// Wait for console log
page.on('console', msg => console.log(msg.text()));
await page.waitForEvent('console');
```

Common events: 'popup', 'close', 'console', 'dialog', 'request', 'response'

7 waitForFunction

Description: Wait until a custom JS function evaluates to true.

Example:

```
await page.waitForFunction(() => window['isDataLoaded'] === true);
```

✓ **Use case:** Dynamic conditions not covered by selectors, like JS variables.

8 waitForNavigation

Description: Wait for a page to navigate. Often used with clicks that trigger navigation.

Example:

```
await Promise.all([
  page.waitForNavigation({ waitUntil: 'networkidle' }),
  page.click('#submit')
]);
```

waitUntil options:

- 'load' → full load
 - 'domcontentloaded' → DOM ready
 - 'networkidle' → network idle
-

9 waitForTimeout vs waitForSelector vs auto-waiting

- `waitForTimeout(ms)` → Fixed delay (slow + flaky)
- `waitForSelector` → Wait for element state (recommended)
- Auto-waiting → Most Playwright actions (`click`, `fill`, `check`) automatically wait for the element to be visible, enabled, and stable

Example:

```
await page.click('#submit'); // Auto-waits for visibility & enabled state
```

10 Combined Example – Optimized Login Wait

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

// Wait for login form visible
await page.waitForSelector('#email', { state: 'visible' });

// Fill login
await page.fill('#email', 'user@test.com');
await page.fill('#password', 'Password@123');

// Wait for navigation and network idle after submit
await Promise.all([
  page.waitForNavigation({ waitUntil: 'networkidle' }),
  page.click('#login-button')
]);

// Wait for dashboard to load completely
await page.waitForSelector('#dashboard', { state: 'visible' });
```

Method	What it waits for	TypeScript Example Usage
waitForTimeout	Fixed delay (Pause)	<pre>await page.waitForTimeout(3000);</pre> <i>(Note: Use sparingly; prefer web-first assertions)</i>
waitForSelector	Element state (visible, attached, hidden, detached)	<pre>await page.waitForSelector('#btn', { state: 'visible' }); await page.waitForSelector(selector, { state?: 'attached' 'detached' 'visible' 'hidden', timeout?: number });</pre>
waitForLoadState	Page loading status	<pre>await page.goto('https://example.com'); await page.waitForLoadState('load'); // Wait for "load" event await page.waitForLoadState('domcontentloaded'); // DOMContentLoaded fired await page.waitForLoadState('networkidle'); // No network requests for 500ms</pre>
waitForResponse	Specific network response	<pre>const response = await page.waitForResponse(r => r.url().includes('/api') && r.status() === 200);</pre>
waitForRequest	Specific network request	<pre>const request = await page.waitForRequest(r => r.method() === 'POST');</pre>
waitForEvent	Specific browser/page event	<pre>const popup = await page.waitForEvent('popup');</pre>
waitForFunction	Custom JS condition in browser	<pre>await page.waitForFunction(() => window.innerWidth > 1024);</pre>
waitForNavigation	URL change or navigation	<pre>await page.waitForNavigation({ waitUntil: 'networkidle' });</pre>

248) All Events for `waitForEvent()` in Playwright (with Examples)

#	Event	Level	When it is Triggered	Use Case	TypeScript Code Example
1	popup	Page	New tab/window opens	Handle new tabs	<pre>typescript const [newPage] = await Promise.all([page.waitForEvent('popup'), page.click('#open-window')]); await newPage.waitForLoadState();</pre>
2	dialog	Page	Alert / Confirm / Prompt	Handle browser dialogs	<pre>typescript page.on('dialog', async dialog => { console.log(dialog.message()); await dialog.accept();});</pre>
3	download	Page	File download starts	Validate file downloads	<pre>typescript const downloadPromise = page.waitForEvent('download'); await page.getText('Download').click(); const download = await downloadPromise; await download.saveAs('file.pdf');</pre>
4	console	Page	console.log() runs	Debug browser logs	<pre>typescript page.on('console', msg => { console.log(`Browser log: \${msg.text()}`);});</pre>
5	pageerror	Page	Uncaught JS error	Detect frontend crashes	<pre>typescript page.on('pageerror', error => { console.log(`Uncaught error: \${error.message}`);});</pre>
6	request	Page	API request sent	Track API calls	<pre>typescript const req = await page.waitForEvent('request'); console.log(req.url());</pre>
7	response	Page	API response received	Validate API responses	<pre>typescript const res = await page.waitForEvent('response'); console.log(res.status());</pre>
8	close	Page	Page is closed	Cleanup validation	<pre>typescript page.on('close', () => console.log('Page closed!'));</pre>
#	Event	Level	When it is Triggered	Use Case	TypeScript Code Example
9	page	Context	A new page/tab is created within the context	Handling multiple tabs globally	<pre>typescript const [newPage] = await Promise.all([context.waitForEvent('page'), page.click('#new-tab-btn')]);</pre>
10	request	Context	Any request is sent by any page in this context	Tracking all background API calls	<pre>typescript context.on('request', request => { console.log('>>', request.method(), request.url());});</pre>
11	response	Context	Any response is received in this context	Global API monitoring/logging	<pre>typescript context.on('response', response => { console.log('<<', response.status(), response.url());});</pre>
12	requestfailed	Context	A network request fails (timeout, DNS, etc.)	Network error detection / Monitoring	<pre>typescript context.on('requestfailed', request => { console.log(`Failed: \${request.url()} - \${request.failure()?.errorText}`);});</pre>
13	requestfinished	Context	A request completes successfully	Ensuring all assets/APIs finished loading	<pre>typescript const finishedReq = await context.waitForEvent('requestfinished'); console.log('Finished:', finishedReq.url());</pre>
14	close	Context	The entire browser context is closed	Cleanup or teardown validation	<pre>typescript context.on('close', () => { console.log('Context has been destroyed');});</pre>

#	Event	Level	When it is Triggered	Use Case	TypeScript Code Example
15	disconnected	Browser	Playwright loses connection to the browser (crash or manual close)	Stability monitoring & unexpected crash handling	<pre>typescript browser.on('disconnected', () => { console.log('Browser process terminated'); }); // Or wait for it: await browser.waitForEvent('disconnected');</pre>

249) Read Specific Sheet & Write to Specific Sheet (ExcelJS + TypeScript)

```
import ExcelJS from 'exceljs';

async function readSpecificSheet() {
  const workbook = new ExcelJS.Workbook();
  await workbook.xlsx.readFile('testdata.xlsx');

  const sheet = workbook.getWorksheet('LoginData');
  if (!sheet) {
    throw new Error('Sheet not found');
  }
//Read a Specific Cell from a Specific Sheet
const sheet = workbook.getWorksheet('LoginData');
const username = sheet?.getCell('A2').value;
const password = sheet?.getCell('B2').value;
console.log(username, password);

//Update Data in a Specific Sheet & Cell
sheet.getCell('B2').value = 'PASS';

}
```

250) What is Destructuring in TypeScript?

Destructuring allows you to **extract values from arrays or properties from objects into variables** in a clean and readable way. TypeScript uses **JavaScript destructuring syntax**, but adds **type safety**.

Create an array

```
const data = ['test@mail.com'];
```

Destructure the array

```
const [email] = data;
```

Arrow function with single parameter without type

```
const sendEmail = ({ email }) => {
  console.log(email);
};
```

Call the function

```
sendEmail({ email });
```