Intermediate Level (Configuration, Execution & Debugging)

1. What is the structure of the playwright.config.js file?

The playwright.config.js file defines how Playwright tests are configured and executed. Its basic structure looks like this:

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


export default defineConfig({
  testDir: './tests',              // Folder containing tests
  timeout: 30000,                  // Global test timeout
  retries: 1,                      // Retry failed tests
  reporter: 'html',                // Reporting format
  use: {
    browserName: 'chromium',       // Default browser
    headless: true,                // Run in headless mode
    baseURL: 'https://example.com',  // Base URL for tests
    screenshot: 'on',              // Capture screenshots
    video: 'retain-on-failure',    // Record video on failure
  },
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox',  use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit',   use: { ...devices['Desktop Safari'] } },
  ],
});
```

✅ Explanation:

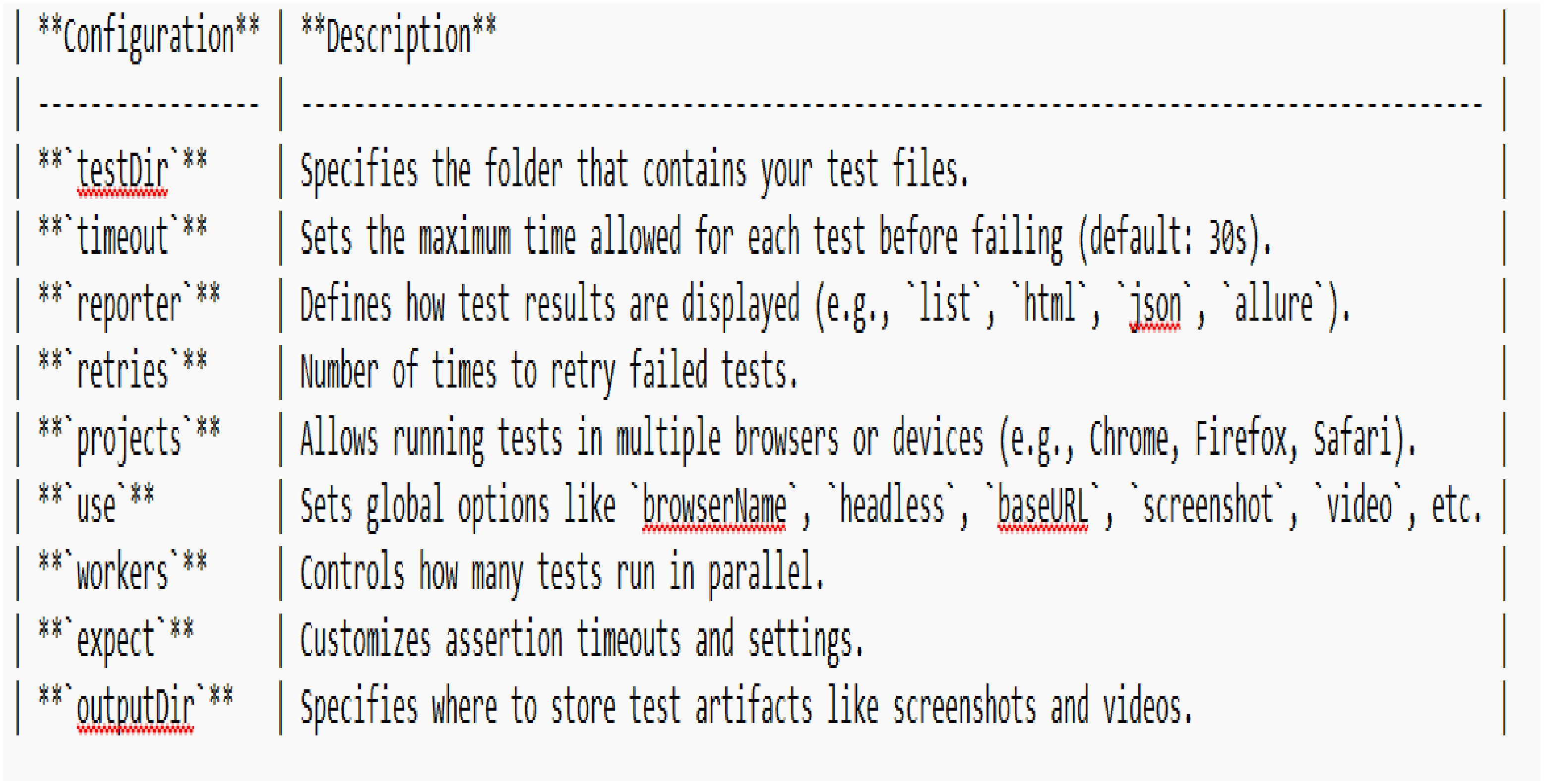
testDir → Location of test files.

use → Global browser and context settings.

projects → Run tests across multiple browsers/devices.

reporter → Controls the type of report generated.

2. What are the key configurations available (testDir, timeout, reporter, projects, use, etc.)?



Configuration	**Description**
`testDir`	Specifies the folder that contains your test files.
`timeout`	Sets the maximum time allowed for each test before failing (default: 30s).
`reporter`	Defines how test results are displayed (e.g., `list`, `html`, `json`, `allure`).
`retries`	Number of times to retry failed tests.
`projects`	Allows running tests in multiple browsers or devices (e.g., Chrome, Firefox, Safari).
`use`	Sets global options like `browserName`, `headless`, `baseURL`, `screenshot`, `video`, etc.
`workers`	Controls how many tests run in parallel.
`expect`	Customizes assertion timeouts and settings.
`outputDir`	Specifies where to store test artifacts like screenshots and videos.

These configurations help you control how tests run, report, and debug across environments and browsers.

3. How do you set browser type in the config file?

You can set the browser type in the playwright.config.js file using the use property.

```
use: {
  browserName: 'chromium', // or 'firefox' or 'webkit'
  headless: true,
}
```

alternatively, if you want to run tests across multiple browsers, define them under projects:

```
projects: [
  { name: 'chromium', use: { browserName: 'chromium' } },
  { name: 'firefox',  use: { browserName: 'firefox' } },
  { name: 'webkit',   use: { browserName: 'webkit' } },
],
```

🎯 This setup lets Playwright automatically execute tests on different browsers during a single test run.

4. How do you change the browser for execution without modifying code?

You can change the browser for execution without modifying the code by using the CLI --project option.

```
npx playwright test --project=chromium
```

or

```
npx playwright test --project=firefox
```

The browser projects (like chromium, firefox, webkit) are already defined in your playwright.config.js. Using --project tells Playwright which browser configuration to use, no code changes needed.

5. How do you configure retries in Playwright?

You can configure retries in Playwright to automatically re-run failed tests using the retries option in the playwright.config.js file.

```
export default defineConfig({

  retries: 2, // Retries failed tests up to 2 times

});
```

retries: 0 → No retries (default).

retries: 2 → Playwright will rerun a failed test twice before marking it failed.

You can also override it via CLI:

```
npx playwright test --retries=2
```

“Override it via CLI” means you can change the configuration temporarily from the command line, without editing your playwright.config.js file.

If your config file has:

```
retries: 0
```

but you run this command:

```
npx playwright test --retries=2
```

☞ Playwright will ignore the config file value and use 2 retries just for that run.

It’s useful when you want to test different settings quickly without modifying your main configuration file.

6. What are environment variables, and how can you use them in Playwright?

Environment variables are external values stored outside your code that you can use to manage configurations like URLs, credentials, or API keys, without hardcoding them.

✅ How to use in Playwright:

Set environment variable:

On Windows:

set BASE_URL=https://staging.example.com

On macOS/Linux:

export BASE_URL=https://staging.example.com

Access it in Playwright:

const baseURL = process.env.BASE_URL;

await page.goto(baseURL);

🎯 Why use them:

Keeps sensitive data (like passwords or API tokens) secure.

Makes tests portable across environments (dev, staging, prod) without code changes.

7. How do you manage different environments (DEV, QA, PROD) using multiple config files?

You can manage multiple environments (DEV, QA, PROD) in Playwright by creating separate config files for each environment and switching between them using the CLI.

✅ **Example folder structure:**

playwright.config.dev.js

playwright.config.qa.js

playwright.config.prod.js

Each file can have different settings, like base URLs:

```
// playwright.config.dev.js
export default {
  use: {
    baseURL: 'https://dev.example.com',
  },
};


// playwright.config.qa.js
export default {
  use: {
    baseURL: 'https://qa.example.com',
  },
};
```

✅ **Run tests for a specific environment:**

```
npx playwright test --config=playwright.config.qa.js
```

🎯 Why:

This approach makes it easy to switch environments without editing your code, keeping your tests cleaner, reusable, and environment-independent.

8. What is the purpose of the reporter option in Playwright?

The reporter option in Playwright defines how test results are displayed or stored after execution.

✅ Example:

```
export default {
  reporter: 'html',
};
```

✅ **Common reporter types:**

list – Shows results in the console (default).

html – Generates a detailed HTML report.

json – Exports test results in JSON format.

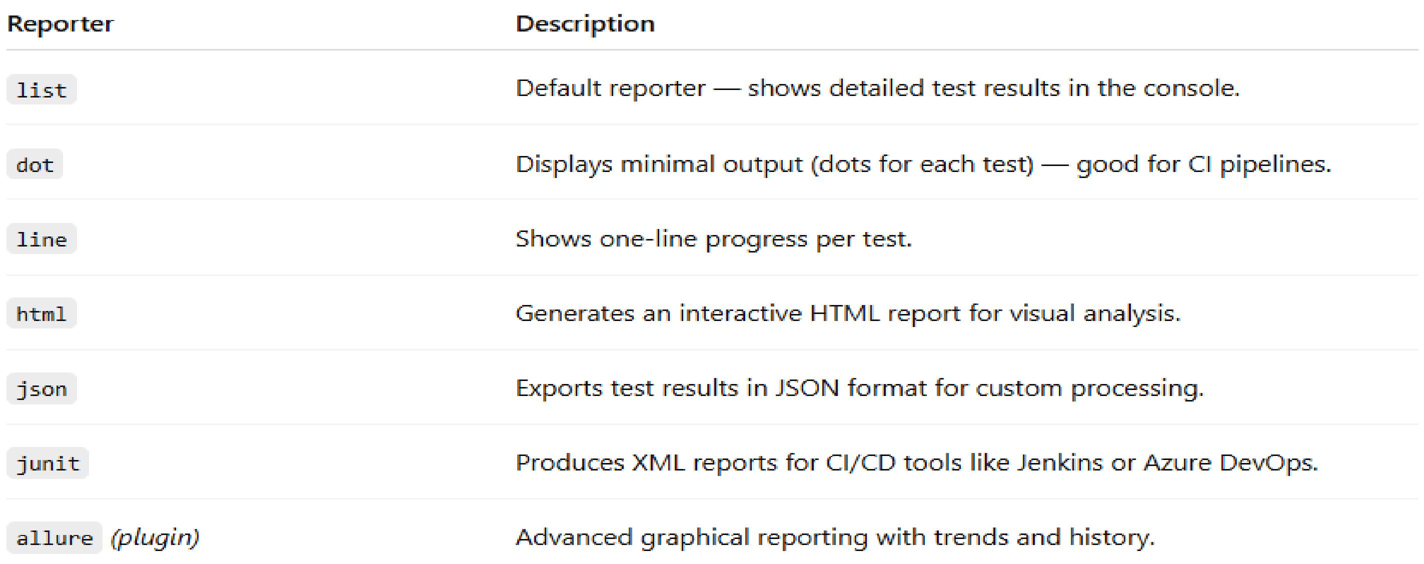
junit – Useful for CI/CD integration.

line, dot, or allure – For compact or advanced reporting styles.

🎯 Purpose: Helps you analyze, visualize, and share test execution results easily.

9. What are the different types of Playwright reporters?

Playwright supports several built-in reporters to display or save test results in different formats.



Reporter	Description
`list`	Default reporter — shows detailed test results in the console.
`dot`	Displays minimal output (dots for each test) — good for CI pipelines.
`line`	Shows one-line progress per test.
`html`	Generates an interactive HTML report for visual analysis.
`json`	Exports test results in JSON format for custom processing.
`junit`	Produces XML reports for CI/CD tools like Jenkins or Azure DevOps.
`allure` *(plugin)*	Advanced graphical reporting with trends and history.

✅ Example usage in config:

```
export default {
  reporter: [['html'], ['json', { outputFile: 'report.json' }]],
};
```

10. How do you enable HTML reports in Playwright?

You can enable HTML reports in Playwright using the reporter option in your configuration file.

✅ Example:

```
export default {
  reporter: 'html',
};
```

After running tests with:

npx playwright test

You can view the HTML report by running:

npx playwright show-report

The report is automatically generated in the playwright-report folder, open index.html to view detailed test results with screenshots, errors, and execution steps.

11.How do you debug a failed Playwright test?

You can debug a failed Playwright test using several powerful options:

✅ **1. Run in headed mode:**

See what’s happening in real time.

```
npx playwright test --headed
```

✅ **2. Use debug mode:**

Pauses execution and opens the inspector.

```
npx playwright test –debug
```

✅ **3. Add debug statements:**

```
await page.pause(); // Opens Playwright Inspector during the test
```

✅ **4. Use traces and screenshots**:

Enable in playwright.config.js:

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

Then view using:

```
npx playwright show-trace trace.zip
```

🎯 Tip: Combine --headed, --debug, and tracing for deep analysis of flaky or failed tests.

12. What is trace viewer in Playwright, and how do you enable it?

The Trace Viewer in Playwright is a powerful tool that lets you visualize every step of your test, including actions, network requests, console logs, and screenshots. It helps you debug failed or flaky tests effectively.

✅ **How to enable tracing:**

In your playwright.config.js file:

```
use: {
  trace: 'on-first-retry', // options: 'on', 'off', 'retain-on-failure', 'on-first-retry'
}
```

✅ **How to open a trace:**

After a test run, open the trace with:

```
npx playwright show-trace trace.zip
```

The Trace Viewer provides a timeline, DOM snapshots, and step-by-step playback, making it one of the best debugging tools in Playwright.

13. How do you capture screenshots in Playwright?

You can capture screenshots in Playwright in two main ways:

✅ **1. Manually in your test:**

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

Captures the current page view and saves it to a file.

You can also use { fullPage: true } to capture the entire page.

✅ **2. Automatically through configuration:**

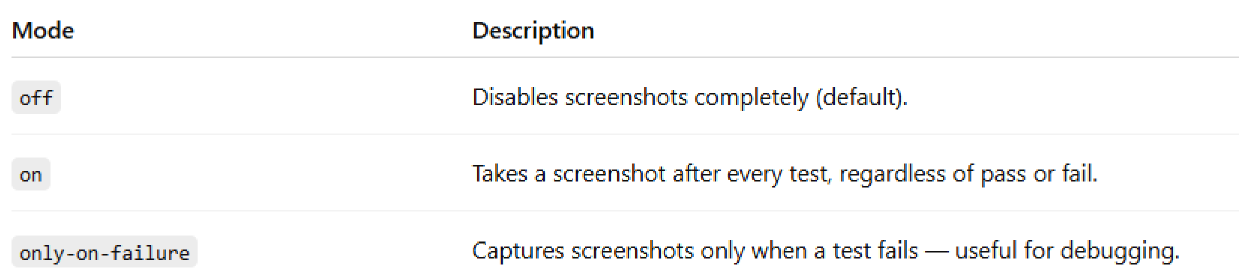
In playwright.config.js:

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

Automatic screenshots are great for debugging failed tests, while manual screenshots help you capture key test steps or UI states.

14. What are the different screenshot modes? (on, off, only-on-failure)

Playwright provides three screenshot modes that you can set in the playwright.config.js file under the use section:



Mode	Description
`off`	Disables screenshots completely (default).
`on`	Takes a screenshot after every test, regardless of pass or fail.
`only-on-failure`	Captures screenshots only when a test fails — useful for debugging.

✅ Example:

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

🎯 Best Practice: Use only-on-failure in CI pipelines to save time and storage while still capturing useful debugging data.

15. How do you handle authentication in Playwright?

You can handle authentication in Playwright using storage state or login automation.

✅ 1. Using storage state (recommended):

First, log in once and save the session:

```
// auth.setup.js
import { test as setup, expect } from '@playwright/test';


setup('Login and save state', async ({ page }) => {
  await page.goto('https://example.com/login');
  await page.fill('#username', 'user');
  await page.fill('#password', 'password');
  await page.click('button[type="submit"]');
  await page.context().storageState({ path: 'auth.json' });
});
```

Then, reuse it in your config or tests:

```
use: {
  storageState: 'auth.json',
}
```

✅ 2. Or log in before each test (less efficient):

```
await page.goto('https://example.com/login');
await page.fill('#user', 'user');
await page.fill('#pass', 'password');
await page.click('text=Login');
```

🎯 Best Practice: Use storage state to avoid repeated logins — it makes tests faster and more stable.

16. What is the use of storageState in Playwright?

The storageState in Playwright is used to save and reuse authentication data (like cookies and local storage) across tests.

✅ Use cases:

Keep users logged in between tests.

Avoid repeating login steps for every test.

✅ Example:

Save login state:

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

Reuse saved state:

```
use: {
  storageState: 'auth.json',
}
```

🎯 Benefit:

It speeds up test execution, reduces flakiness, and ensures consistent authenticated sessions across multiple test runs.

17. How do you reuse a logged-in state between multiple tests?

You can reuse a logged-in state between multiple tests in Playwright using the storageState feature.

✅ Step 1: Log in once and save the state

Create a setup file (e.g., auth.setup.js):

```
import { test as setup } from '@playwright/test';
setup('Login and save state', async ({ page }) => {
  await page.goto('https://example.com/login');
  await page.fill('#username', 'user');
  await page.fill('#password', 'password');
  await page.click('button[type="submit"]');
  await page.context().storageState({ path: 'auth.json' });
});
```

✅ Step 2: Reuse the saved state in config

In your playwright.config.js:

```
use: {
  storageState: 'auth.json',
}
```

✅ Step 3: Run your regular tests

All tests will start in the logged-in state, no need to repeat the login steps.

🎯 Benefit: Faster tests, less flakiness, and consistent sessions across multiple test files.

18.How do you emulate mobile devices in Playwright?

You can emulate mobile devices in Playwright using the built-in device descriptors from @playwright/test.

✅ Example:

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

export default defineConfig({
  projects: [
    {
      name: 'Mobile Chrome',
      use: { ...devices['Pixel 5'] }, // Emulates Pixel 5
    },
    {
      name: 'Mobile Safari',
      use: { ...devices['iPhone 12'] }, // Emulates iPhone 12
    },
  ],
});
```

✅ What this does:

Simulates screen size, resolution, user agent, and touch behavior of real devices.

Lets you test responsive design and mobile-specific features.

🎯 Tip: You can view all available devices using:

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

console.log(devices);
```

19. What is the devices object used for in Playwright?

The devices object in Playwright provides predefined device configurations that simulate real mobile and tablet environments.

✅ It includes:

Screen size and resolution

Device pixel ratio

User agent string

Touch support

Default browser settings

✅ Example usage:

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

export default {

  use: { ...devices['iPhone 12'] }, // Emulates iPhone 12

};
```

🎯 Purpose:

The devices object helps you test responsive layouts and mobile behaviors easily , without manually setting viewport or user agent details.

20. How can you handle multiple browser tabs in Playwright?

You can handle multiple browser tabs (or pages) in Playwright using the context object, which can manage multiple pages within the same browser session.

✅ Example:

```
const [newPage] = await Promise.all([
  context.waitForEvent('page'),      // Waits for the new tab
  page.click('a[target="_blank"]'),  // Action that opens a new tab
]);


await newPage.bringToFront();       // Switch focus to the new tab
await newPage.waitForLoadState();
console.log(await newPage.title());
```

✅ Explanation:

context.waitForEvent('page') → Listens for a new tab opening.

newPage → Represents the newly opened tab.

bringToFront() → Switches control to that tab.

🎯 Tip:

Each tab in Playwright is treated as a separate Page object, so you can perform actions and assertions independently on each.