1️⃣ What is Playwright (from framework view)

Playwright is an **end-to-end automation framework** for modern web apps that supports:

- Chromium, Firefox, WebKit
- Headless & headed execution
- Auto-waits (huge stability win)
- Parallel execution out of the box
- API + UI testing in the same framework

When we say **"Playwright Framework"**, we mean:

> A **structured, scalable test automation solution**, not just test scripts.

2️⃣ Technology Stack (Standard)

Layer	Tool
Language	JavaScript / TypeScript
Test Runner	Playwright Test
Assertion	Built-in expect
Reporting	HTML, Allure
CI/CD	Jenkins / GitHub Actions
Design Pattern	Page Object Model
Config	playwright.config.js
Environment	dotenv / config files

3️⃣ Recommended Folder Structure (Industry-grade)

```
playwright-automation/
│
├── tests/
│   ├── login/
│   │   └── login.spec.js
│   ├── dashboard/
│   │   └── dashboard.spec.js
│   └── api/
│       └── user.api.spec.js
│
├── pages/
│   ├── LoginPage.js
│   ├── DashboardPage.js
│   └── BasePage.js
│
├── utils/
│   ├── testData.js
│   ├── apiHelper.js
│   ├── waitUtils.js
│   └── logger.js
│
├── fixtures/
│   └── testFixture.js
│
├── config/
│   ├── dev.env
│   ├── qa.env
│   └── prod.env
│
├── reports/
│
├── playwright.config.js
├── package.json
├── .env
├── .gitignore
└── README.md
```

4️⃣ Core Framework Components (Deep Explanation)

A□ playwright.config.js (Heart of Framework)

Controls:

- Browsers
- Parallelism
- Timeouts
- Base URL
- Reporting
- Screenshots / videos

Example (realistic)

js Copy

```
import { defineConfig } from '@playwright/

export default defineConfig({
  testDir: './tests',
  timeout: 30 * 1000,

  expect: {
    timeout: 5000
  },

  retries: 1,

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

  use: {
    baseURL: 'https://example.com',
    headless: true,
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
    trace: 'on-first-retry'
  },

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

B Page Object Model (POM)

Why POM?

- Separation of test logic and UI locators
- Easy maintenance
- Reusable actions

BasePage.js

js Copy

```
export class BasePage {
  constructor(page) {
    this.page = page;
  }

  async navigate(url) {
    await this.page.goto(url);
  }

  async getTitle() {
    return await this.page.title();
  }
}
```

LoginPage.js

```js
import { BasePage } from './BasePage';

export class LoginPage extends BasePage {

  constructor(page) {
    super(page);
    this.username = page.locator('#usernam
    this.password = page.locator('#passwor
    this.loginBtn = page.locator('#login')
  }

  async login(user, pass) {
    await this.username.fill(user);
    await this.password.fill(pass);
    await this.loginBtn.click();
  }
}
```

📌 Best Practice

- No assertions in page classes
- Only actions + locators

C□ Test Layer (tests/)

login.spec.js

```js
import { test, expect } from '@playwright/
import { LoginPage } from '../../pages/Log

test('Valid user should login successfully
  const loginPage = new LoginPage(page);

  await loginPage.navigate('/login');
  await loginPage.login('admin', 'passwor

  await expect(page).toHaveURL(/dashboard/
});
```

📌 Tests:

- Contain assertions
- Read like **business scenarios**

D□ Fixtures (Advanced Control)

Used to:

- Share setup/teardown
- Inject authenticated state
- Reuse browser context

testFixture.js

```js
import { test as base } from '@playwright/te

export const test = base.extend({
  loggedInPage: async ({ page }, use) => {
    await page.goto('/login');
    await page.fill('#username', 'admin');
    await page.fill('#password', 'password')
    await page.click('#login');
    await use(page);
  }
```

E Utilities Layer

Examples

Utility	Purpose
testData.js	Centralized data
apiHelper.js	API calls
logger.js	Logs
waitUtils.js	Custom waits

F API + UI in Same Framework

```js
test('Create user via API', async ({ reque
  const response = await request.post('/us
    data: { name: 'John' }
  });

  expect(response.status()).toBe(201);
});
```

5️⃣ Reporting & Debugging

Built-in:

- HTML Report
- Trace Viewer
- Video recording
- Screenshots

```bash
npx playwright show-report
```

6️⃣ CI/CD (Jenkins Example)

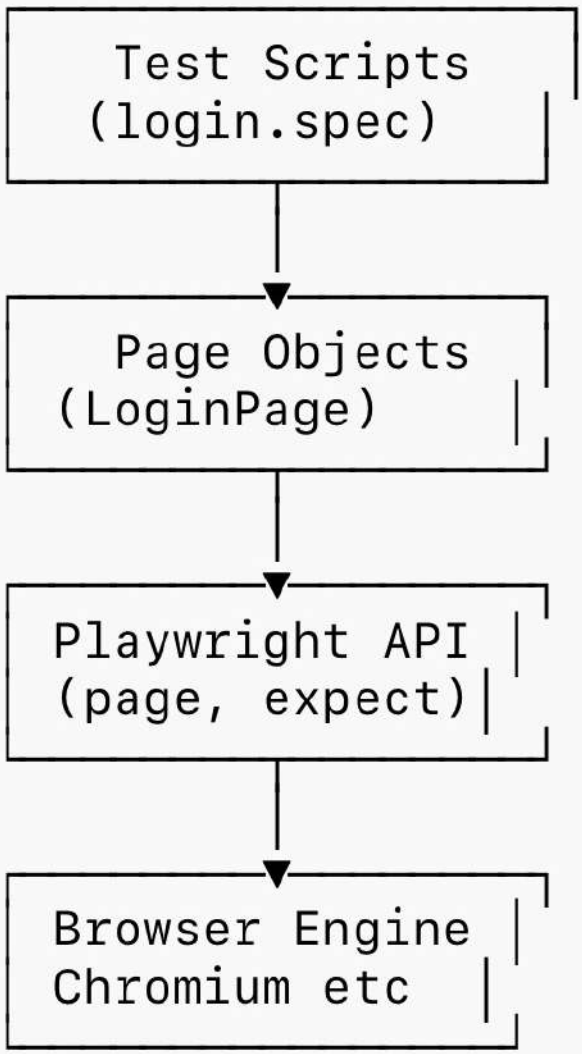
```bash
npm install
npx playwright install
npx playwright test
```

Artifacts:

- `/playwright-report`
- Videos
- Screenshots

7️⃣ Framework Diagram (High Level)

code Copy



8️⃣ Execution Flow Chart

code

Copy

```
Start
  |
  ▼
Read playwright.config.js
  |
  ▼
Launch Browser
  |
  ▼
Create Context & Page
  |
  ▼
Execute Before Hooks / Fixtures
  |
  ▼
Run Test
  |
  ▼
Call Page Methods
  |
  ▼
Assertions
  |
  ▼
Capture Screenshot / Video on Failure
  |
  ▼
Generate Report
  |
  ▼
Close Browser
  |
  ▼
End
```

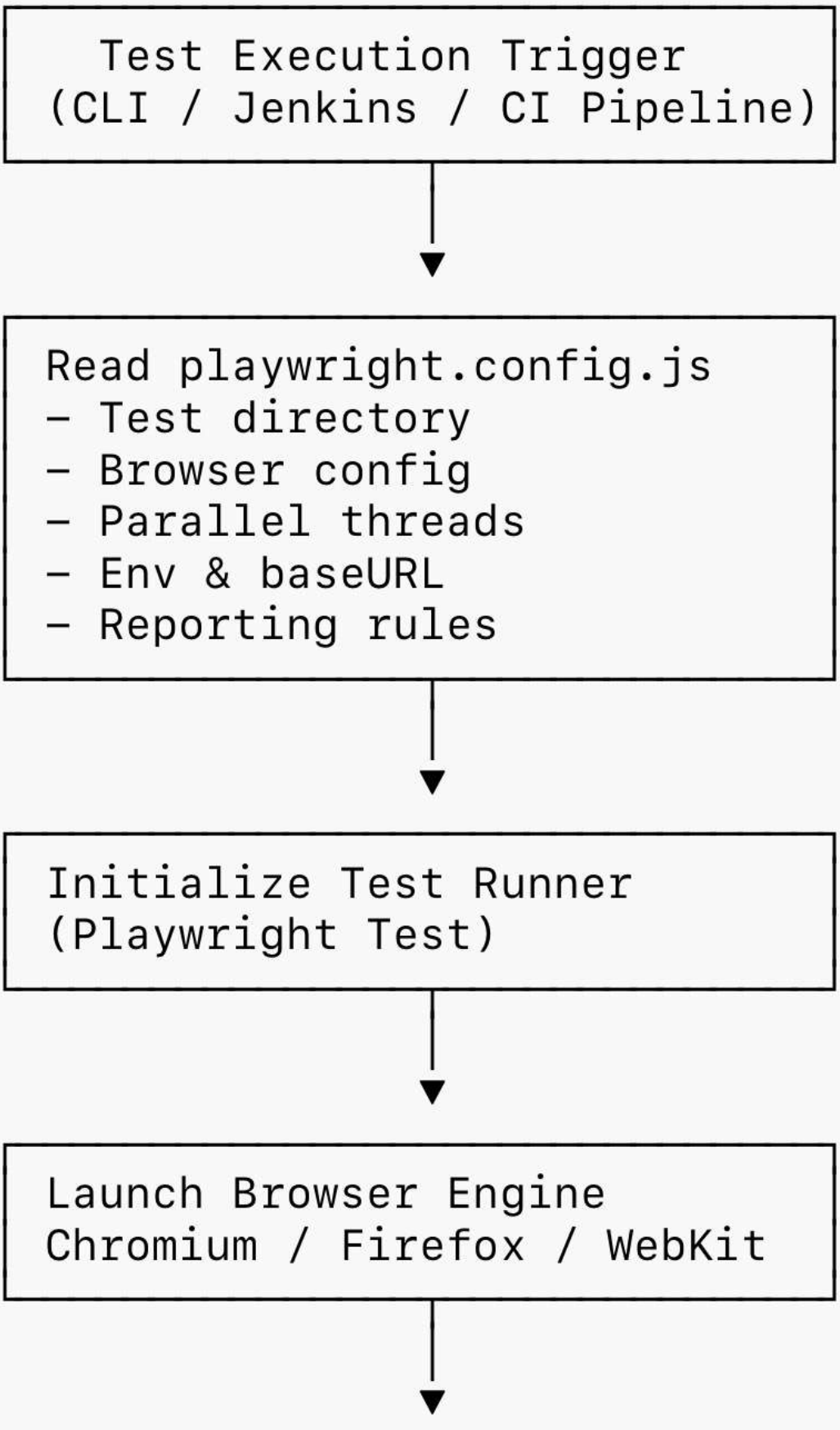
🔷 Framework Responsibility Mapping

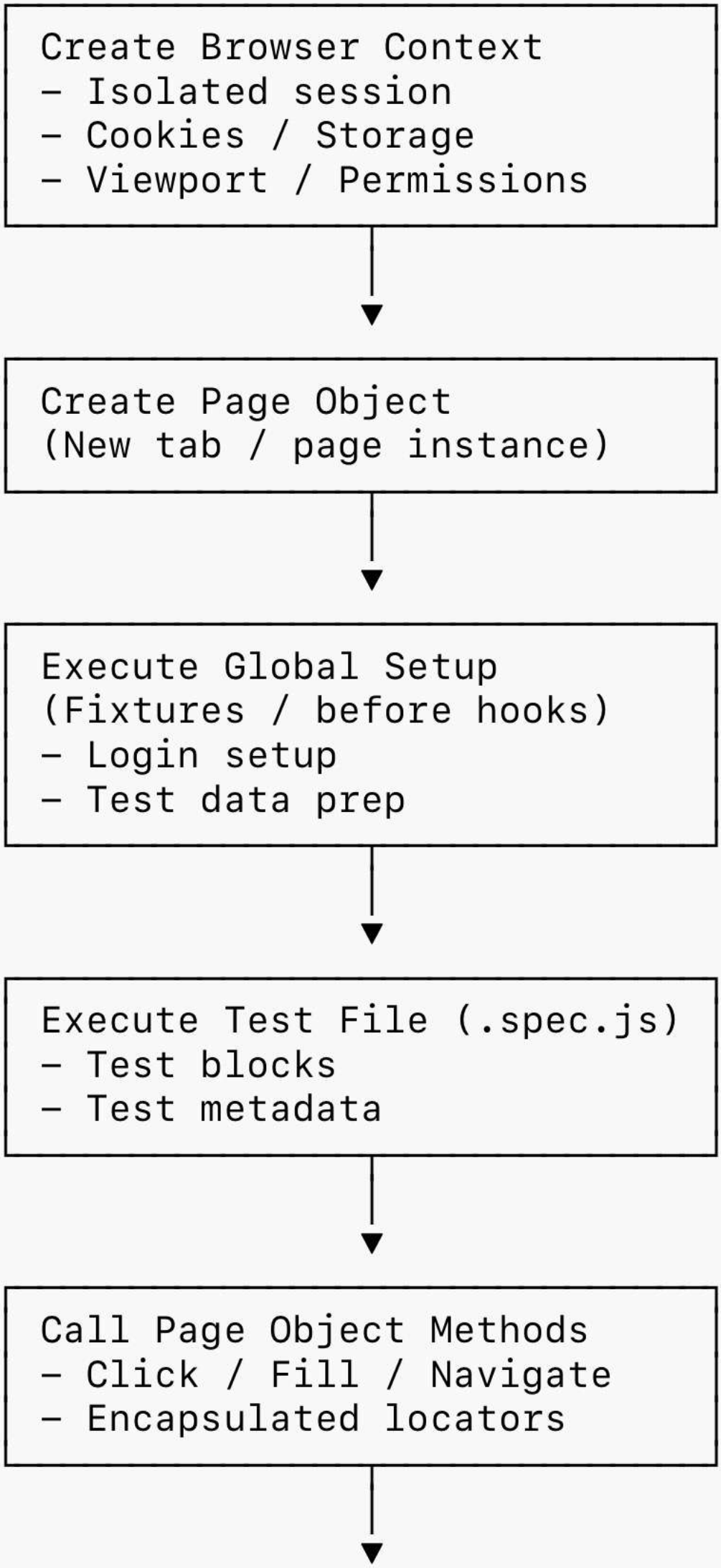
Layer	Responsibility
Config	Execution behavior
Fixtures	Setup & teardown
Page Objects	UI abstraction
Tests	Business validation
Playwright Core	Stability & waits
Reporter	Evidence & metrics
CI	Trigger & publish

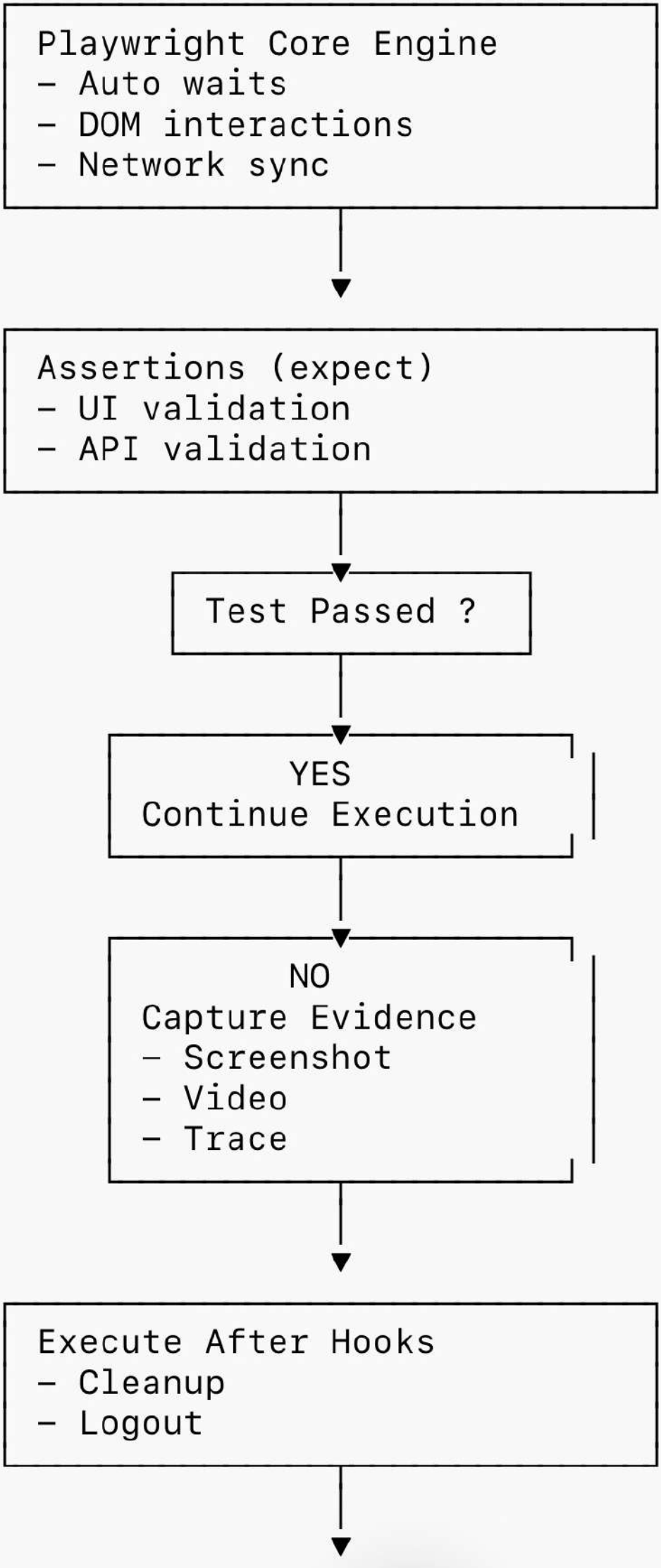
🔷 Playwright Framework – Execution Flow Chart

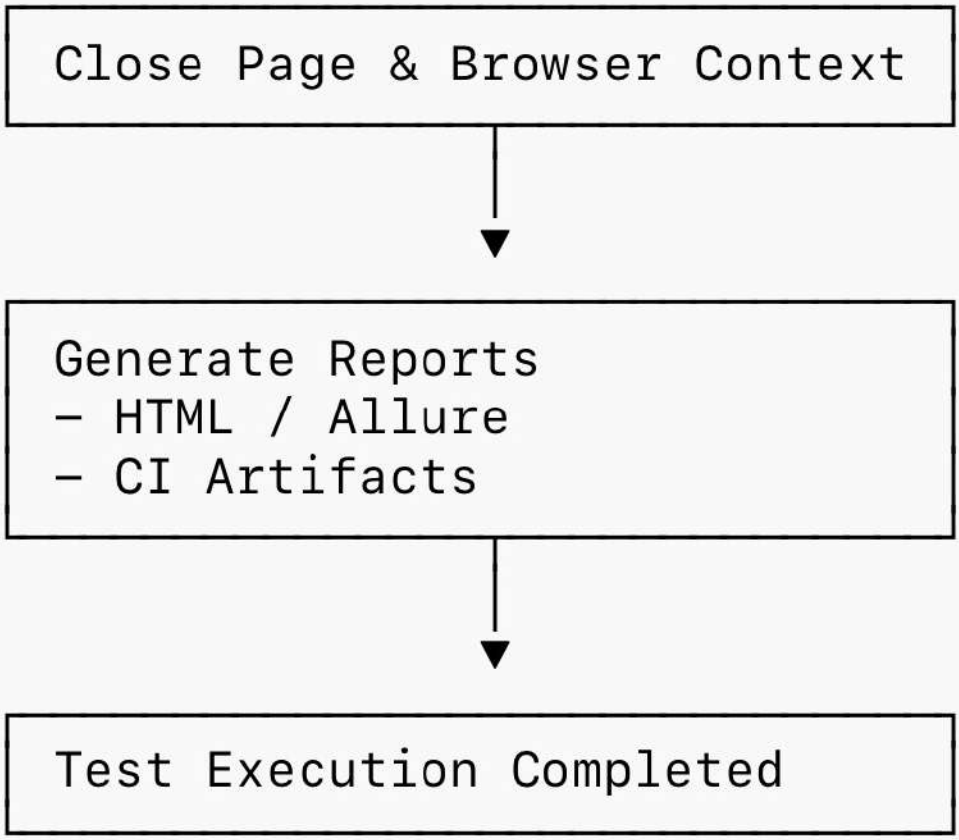
code

Copy




Create Browser Context
- Isolated session
- Cookies / Storage
- Viewport / Permissions
Create Page Object
(New tab / page instance)
Execute Global Setup
(Fixtures / before hooks)
- Login setup
- Test data prep
Execute Test File (.spec.js)
- Test blocks
- Test metadata
Call Page Object Methods
- Click / Fill / Navigate
- Encapsulated locators


Playwright Core Engine
- Auto waits
- DOM interactions
- Network sync
Assertions (expect)
- UI validation
- API validation
Test Passed ?
YES
Continue Execution
NO
Capture Evidence
- Screenshot
- Video
- Trace
Execute After Hooks
- Cleanup
- Logout



🔷 How to Explain This in Interviews (Important)

One-liner:

> “Playwright framework execution starts from config, spins isolated browser contexts, executes tests through page objects, validates via assertions, captures failure artifacts automatically, and generates reports before cleanup.”