🔹 PART 1: Foundations of Async/Await (JavaScript/TypeScript)

🔸 What is `async/await`?

`async/await` is a modern syntax in JavaScript used to work with asynchronous code in a more readable and synchronous-like fashion.

🔑 **Key Definitions:**

- `async` keyword: Makes a function return a **Promise**.
- `await` keyword: Pauses the execution of an `async` function until the **Promise resolves**.

🔸 Why it's important in Playwright

Playwright APIs are **asynchronous** — they return **Promises** because browser interactions are time-based. Using `await` ensures:
✅ Correct sequencing

🧠 Example:

```
test('Login flow', async ({ page }) => {
  await page.goto('https://example.com/login');
  await page.fill('#username', 'nikhil');
  await page.fill('#password', '123456');
  await page.click('button[type="submit"]');
  await expect(page).toHaveURL(/dashboard/);
});
```

> Without `await`, the steps will not execute in order, leading to flaky or failing tests.

❓ Quick Quiz:

Q1. What happens if you forget to use `await` with a Playwright action?
Q2. Can you use `await` inside a non-async function?

🔹 PART 2: Playwright Test Runner Basics and Annotations

🔸 What are Test Annotations?

Annotations in Playwright help **control, tag, and configure** your tests. They are typically added using special functions like:

- `test.describe()`
- `test.only()`, `test.skip()`
- `test.use()`
- `test.fixme()`, `test.fail()`, `test.slow()`

🔑 Common Annotations & Their Purpose

Annotation	Purpose
`test.skip()`	Skips a test
`test.only()`	Runs only this test
`test.describe()`	Groups related tests
`test.use()`	Sets test-specific configurations
`test.beforeEach()`	Hook before each test
`test.afterEach()`	Hook after each test

✅ Real Use Case:

```
test.describe('Dashboard tests', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('/login');
    await page.fill('#username', 'admin');
    await page.fill('#password', 'admin');
    await page.click('button[type="submit"]');
  });

  test('should show the analytics section', async ({ page }) => {
    await expect(page.locator('#analytics')).toBeVisible();
  });

  test.skip('should allow deleting user', async ({ page }) => {
    // Temporarily skipping due to backend issues
  });
});
```

🔍 Common Misconceptions:

Misconception	Reality
"Annotations are optional"	True, but they're powerful for clean, maintainable test suites
"You can use await anywhere"	❌ Only inside `async` functions
"`test.use()` works globally"	❌ Only affects tests within its scope

📌 Tip:

Use `test.describe()` to group tests by features or modules. It helps in test isolation and filtering during CI runs.

❓ Quiz Time:

Q1. What does `test.skip()` do?
Q2. What's the difference between `test.beforeEach()` and `test.use()`?
Q3. Where must `await` be used in Playwright?

🔹 PART 3: Advanced Async + Annotations Integration

🔸 Combining Async Patterns with Annotations

In real-world test automation, combining `async/await` **inside hooks and annotations** ensures setup and teardown logic works correctly.

🔁 **Example:**

```
test.describe('Admin Access Flow', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('/admin');
    await page.fill('#admin-user', 'root');
    await page.fill('#admin-pass', 'root123');
    await page.click('button[type="submit"]');
  });

  test('should allow admin to access settings', async ({ page }) =>
{
    await expect(page.locator('#settings')).toBeVisible();
  });

  test.afterEach(async ({ page }) => {
    await page.close();
  });
});
```

🧰 Real-World Scenario: API Setup Before Test

```
test.use({
  storageState: 'loggedInState.json',
});

test('should view profile with API token', async ({ page }) => {
```

```
  await page.goto('/profile');
  await expect(page.locator('h1')).toContainText('Nikhil');
});
```

- You could also fetch token via `request.newContext()` and store it.

⚠️ Common Pitfalls:

1. **Mixing sync & async**: Never forget `await` for navigation or element actions.
2. **Global async setup**: Place it in `globalSetup.ts`, not test files.
3. **Flaky tests**: Due to missing `await`, async actions overlap.

🔁 Retention Tips:

- Always wrap Playwright actions in `await`
- Practice test hooks: `beforeEach`, `afterEach`
- Group tests by `describe()` — think in modules
- Bookmark and review Playwright's Annotations Docs

🧪 Practice Problems

1. ✅ Write a Playwright test that:
    - Logs in as a user
    - Navigates to a dashboard
    - Verifies the presence of a "Logout" button
    - Uses `test.describe()` and `beforeEach`

2. 🧠 Modify the above to:
    - Skip one test using `test.skip()`
    - Tag another as slow using `test.slow()`
3. 🚀 Use `test.use()` to set viewport size and storage state for a test

---

📋 Wrap-Up Quiz:

1. What is the main use of `async/await` in Playwright tests?
2. How does `test.describe()` help in organizing test suites?
3. What's the difference between `test.use()` and `test.beforeEach()`?
4. What happens if you forget `await` on `page.click()`?
5. Can we use async logic inside `test.skip()`?

✅ Next Steps:

- ✅ Practice writing 3 test suites using `describe`, `beforeEach`, `afterEach`
- ✅ Challenge: Write an e2e test using API login (async context), save storage state
- ✅ Read and annotate your own Playwright tests, identify missed `await` keywords

---

🎯 Topic: Mastering Browser Context and Page Fixtures in Playwright (JavaScript)

🧱 PART 1: Understanding the Basics

🔹 What is a "Fixture"?

Fixture in Playwright Test means:

> *A reusable object or setup that you can inject into your test.*

🔹 What is a "Page" Fixture?

- `page` is the most common built-in fixture.
- Represents a **single tab** or **browser window**.
- It's **isolated per test** for consistency.

```
test('Open Google', async ({ page }) => {
  await page.goto('https://www.google.com');
  await expect(page).toHaveTitle(/Google/);
});
```

🔹 What is a "Browser Context"?

A **Browser Context** is like a **clean browser profile**.

Think of it as:

> *Incognito mode: no cookies, no cache, no shared sessions.*

You can have **multiple contexts** per browser instance:

- Each test runs in its own **isolated context**
- Contexts are **lighter** than new browser instances

🔑 Key Differences: `browser`, `context`, `page`

Term	Meaning

`browser`	Full browser instance (e.g., Chromium)
`context`	Isolated session within a browser
`page`	Single tab within a context

📚 Real-World Analogy:

- 🧍 `browser`: You (the human)
- 🧢 `context`: Your incognito browser profile
- 📖 `page`: A tab in the incognito window

🧠 Quiz 1 (With Answers)

Q1. What is the difference between a page and a context?
A1. A page is a single tab; a context is an isolated browser session that can contain multiple pages.

Q2. Why is context isolation useful?
A2. It ensures tests don't share cookies, sessions, or local storage — reducing flakiness.

🚀 PART 2: Using `browser`, `context`, and `page` Manually

🔸 Manual Creation of Contexts and Pages

Sometimes, you want **manual control** over context and page (especially in setup, storageState, or advanced flows):

```
const { test, expect } = require('@playwright/test');

test('Manual context and page creation', async ({ browser }) => {
  const context = await browser.newContext();
  const page = await context.newPage();
```

```
  await page.goto('https://example.com');
  await expect(page).toHaveTitle(/Example/);

  await context.close(); // Clean up
});
```

✅ Use Cases:

- Login setup using API → save storage state
- Testing multiple tabs/windows
- Managing different users (admin vs user)

📌 Tips:

- Always `close()` your manual contexts to free memory
- Don't use `page` and `context` together unless needed — prefer one pattern

---

🧠 Quiz 2 (With Answers)

Q1. Why might you create a context manually?
A1. To control user sessions, simulate login, or reuse authenticated states.

Q2. What must you do after creating a context manually?
A2. Close it using `context.close()`.

🧰 PART 3: Creating Custom Fixtures (Advanced)

🔸 Custom `context` and `page` Fixtures

Let's say you want:

- Custom viewport or permissions
- Storage state preloaded
- Special configurations

➡️ Use `test.extend()` to **customize or override fixtures**.

```
// tests/fixtures.js
const base = require('@playwright/test');

exports.test = base.test.extend({
  context: async ({ browser }, use) => {
    const context = await browser.newContext({
      storageState: 'auth.json',
      permissions: ['geolocation']
    });
    await use(context);
    await context.close();
  },

  page: async ({ context }, use) => {
    const page = await context.newPage();
    await use(page);
    await page.close();
  }
});
```

Then in your tests:

```
// tests/example.spec.js
const { test, expect } = require('./fixtures');

test('Custom page with permissions', async ({ page }) => {
  await page.goto('https://maps.example.com');
  // Geolocation-related testing
});
```

📦 Real Use Case:

1. Login via API
2. Save session in `auth.json`

3. Use `storageState` in custom fixture

🔐 Pro Tip:

- Fixtures can be async or sync
- You can chain them: `browser → context → page`

🧠 Quiz 3 (With Answers)

Q1. How do you override the default `context` fixture?
A1. Use `test.extend()` and define a custom context with `use(context)`.

Q2. What's the benefit of using a shared fixture like `auth.json`?
A2. It avoids repeated login steps, speeding up test execution.

🧪 Practice Problems with Solutions

🧩 Problem 1: Create 2 separate contexts in a single test

```
test('Two user flows in separate contexts', async ({ browser }) => {
  const adminContext = await browser.newContext();
  const userContext = await browser.newContext();

  const adminPage = await adminContext.newPage();
  const userPage = await userContext.newPage();

  await adminPage.goto('/admin');
  await userPage.goto('/dashboard');

  await expect(adminPage.locator('h1')).toContainText('Admin');
  await expect(userPage.locator('h1')).toContainText('User');

  await adminContext.close();
  await userContext.close();
});
```

🧩 Problem 2: Build a fixture that auto-logs in using `auth.json`

Step 1: Login and save state

```
test('Login and save session', async ({ browser }) => {
  const context = await browser.newContext();
  const page = await context.newPage();

  await page.goto('/login');
  await page.fill('#username', 'admin');
  await page.fill('#password', 'password');
  await page.click('text=Login');

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

Step 2: Reuse it in a custom fixture

```
const base = require('@playwright/test');

exports.test = base.test.extend({
  context: async ({ browser }, use) => {
    const context = await browser.newContext({
      storageState: 'auth.json'
    });
    await use(context);
    await context.close();
  },
  page: async ({ context }, use) => {
    const page = await context.newPage();
    await use(page);
    await page.close();
  }
});
```

🧠 Quiz 4 (With Answers)

Q1. How do you share session across tests?
A1. Save the session using `context.storageState({ path: 'file.json' })` and reuse with `storageState` config.

Q2. What does `use(context)` do inside a fixture?
A2. It passes the custom context to the test and waits for the test to finish before cleanup.

🧠 Retention Tips:

- Draw this diagram from memory:

```
browser
├─ context1
│   └─ page1
└─ context2
    └─ page2
```

- Practice switching between default and custom fixtures
- Repeat login → storageState → reuse → fixture extension flow
- Use a consistent test project folder for custom fixtures

✅ Summary Cheatsheet:

Concept	Description
`page`	A single browser tab
`context`	Isolated browser session
`browser`	Main browser instance
`test.extend()`	Create custom fixtures
`storageState`	Saves login/session info
`use()`	Injects the fixture into tests
