TypeScript for Playwright Automation

End-to-end checklist students will use throughout the course (with definitions and examples).

How to use this handout

Keep this as your daily reference. Each item below is something you will repeatedly apply while writing Playwright tests in TypeScript. Aim to follow the patterns consistently to keep tests stable, readable, and scalable.

Quick Checklist (1-25)

1.	const vs let
2.	Primitive types and type inference
3.	Typed arrays and objects
4.	Functions and return types
5.	async/await and Promises
6.	Error handling with try/catch
7.	ES Modules: import/export
8.	Folder structure conventions
9.	Reusable helper functions
10.	type vs interface
11.	Optional properties and union types
12.	Type narrowing (safe guards)
13.	Enums or string unions for constants
14.	Generics
15.	Typed Page Objects
16.	Typed fixtures with test.extend
17.	Strong typing for locators and returns
18.	Typed config and environment variables
19.	Typed API utilities (APIRequestContext)
20.	Custom assertion helpers (typed)
21.	Linting and formatting (ESLint/Prettier)
22.	Avoid any (prefer unknown + narrowing)
23.	Test data factories/builders
24.	Clean error messages and context logging
25.	Scalability: composition over inheritance

Definitions and Examples

For each item: read the definition, then copy the example into your own project to practice.

1. `const` vs `let`

Definition: Use `const` for values that should not be reassigned; use `let` only when reassignment is required. This reduces bugs and improves readability.

Example:

```
const baseUrl = "https://example.com";
let retries = 0;

retries++;
```

2. Primitive types and type inference

Definition: TypeScript infers types from assigned values; explicit primitive types improve clarity when needed.

Example:

```
const suiteName = "Smoke Suite"; // inferred string
const maxUsers: number = 5;
const headless: boolean = true;
```

3. Typed arrays and objects

Definition: Define types for arrays and objects to enforce consistent structure and prevent invalid values.

Example:

```
const browsers: string[] = ["chromium", "firefox", "webkit"];

const user: { username: string; password: string } = {
  username: "qa_user",
  password: "secret",
};
```

4. Functions and return types

Definition: Declare parameter and return types to make behavior predictable and enforce correct usage.

Example:

```
function buildEmail(name: string): string {
  return `${name.toLowerCase()}@example.com`;
}
```

5. async/await and Promises

Definition: Use `async/await` for clear asynchronous code. Most Playwright actions are async and must be awaited.

Example:

```
import { test, expect } from "@playwright/test";

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

6. Error handling with try/catch

Definition: Catch errors to add context (screenshot/logs) while still failing the test correctly.

Example:

```
import { test } from "@playwright/test";

test("safe action", async ({ page }) => {
  try {
    await page.goto("https://example.com");
    await page.click("text=Start");
  } catch (err) {
    await page.screenshot({ path: "error.png", fullPage: true });
    throw err;
  }
});
```

7. ES Modules: import/export

Definition: Organize code into reusable files and share functions or classes using module syntax.

Example:

```
// utils/date.ts
export function todayISO(): string {
  return new Date().toISOString().split("T")[0];
}

// test file
import { todayISO } from "../utils/date";

console.log(todayISO());
```

8. Folder structure conventions

Definition: Use a consistent structure so teams can scale: tests, pages, fixtures, utils, and data.

Example:

```
tests/
pages/
fixtures/
utils/
data/
```

9. Reusable helper functions

Definition: Create small, pure utilities to avoid duplicating logic across tests.

Example:

```
export function assertNonEmpty(value: string, label: string): void {
  if (!value.trim()) throw new Error(`${label} must not be empty`);
}
```

10. `type` vs `interface`

Definition: Use `interface` for object shapes (often extended) and `type` for unions, compositions, and aliases.

Example:

```
export interface Credentials {
  username: string;
  password: string;
}

export type UserRole = "admin" | "member" | "guest";
```

11. Optional properties and union types

Definition: Optional properties allow missing fields; unions allow multiple valid types for a value.

Example:

```
interface UserProfile {
  name: string;
  phone?: string;
}

function formatPhone(phone: string | undefined): string {
  return phone ?? "N/A";
}
```

12. Type narrowing (safe guards)

Definition: Convert broad types (like `unknown` or unions) into safe specific types using checks before using them.

Example:

```
function toErrorMessage(err: unknown): string {
  if (err instanceof Error) return err.message;
  return String(err);
}
```

13. Enums or string unions for constants

Definition: Restrict values to approved constants (env, role, browser). String unions are often the simplest choice.

Example:

```
type Env = "dev" | "qa" | "prod";

const env: Env = "qa";
```

14. Generics

Definition: Write reusable functions that work with many types while preserving type safety.

Example:

```
function first<T>(items: T[]): T {
  return items[0];
}

const n = first([1, 2, 3]);       // number
const s = first(["a", "b", "c"]); // string
```

15. Typed Page Objects

Definition: Encapsulate locators and actions into a class to reduce duplication and improve readability.

Example:

```
import { Page, Locator, expect } from "@playwright/test";

export class LoginPage {
  readonly username: Locator;
  readonly password: Locator;
  readonly submit: Locator;

  constructor(private page: Page) {
    this.username = page.locator("#username");
    this.password = page.locator("#password");
    this.submit = page.locator("button[type=submit]");
  }

  async goto(): Promise<void> {
    await this.page.goto("/login");
  }

  async login(user: string, pass: string): Promise<void> {
    await this.username.fill(user);
    await this.password.fill(pass);
    await this.submit.click();
    await expect(this.page).toHaveURL(/dashboard/);
  }
}
```

16. Typed fixtures with test.extend

Definition: Use fixtures to inject reusable setup (pages, auth, API clients) into tests with strong typing.

Example:

```
import { test as base } from "@playwright/test";
import { LoginPage } from "../pages/LoginPage";

type Fixtures = { loginPage: LoginPage };

export const test = base.extend<Fixtures>({
  loginPage: async ({ page }, use) => {
    await use(new LoginPage(page));
  },
});

export { expect } from "@playwright/test";
```

17. Strong typing for locators and returns

Definition: Ensure helpers return correct Playwright types (Locator, Page, etc.) for safer usage and better IDE support.

Example:

```
import { Page, Locator } from "@playwright/test";

export function menuItem(page: Page, name: string): Locator {
  return page.getByRole("menuitem", { name });
}
```

18. Typed config and environment variables

Definition: Validate and type environment variables to avoid runtime surprises and misconfiguration.

Example:

```
type EnvName = "dev" | "qa" | "prod";

function getEnv(): EnvName {
  const v = process.env.ENV as EnvName | undefined;
  return v ?? "qa";
}
```

19. Typed API utilities (APIRequestContext)

Definition: Create reusable API helpers with typed inputs and outputs to create data and validate endpoints reliably.

Example:

```
import { APIRequestContext, expect } from "@playwright/test";

export async function createUser(
  api: APIRequestContext,
  payload: { name: string }
) {
  const res = await api.post("/api/users", { data: payload });
  expect(res.ok()).toBeTruthy();
  return (await res.json()) as { id: string; name: string };
}
```

20. Custom assertion helpers (typed)

Definition: Centralize common assertions for consistency and clearer failure messages.

Example:

```
import { expect, Locator } from "@playwright/test";

export async function expectVisible(locator: Locator): Promise<void> {
  await expect(locator).toBeVisible();
}
```

21. Linting and formatting (ESLint/Prettier)

Definition: Enforce consistent style and catch common issues early. Keep lint clean before merging.

Example:

```
// Team rule (example)
npm run lint
npm run format
```

22. Avoid any (prefer unknown + narrowing)

Definition: `any` disables type safety. Use `unknown` and narrow safely before using the value.

Example:

```
function parseJson(text: string): unknown {
  return JSON.parse(text);
}
```

23. Test data factories/builders

Definition: Generate consistent test data with defaults and overrides to keep tests readable and scalable.

Example:

```
type User = { username: string; password: string; role: "admin" | "user" };

export function buildUser(overrides?: Partial<User>): User {
  return {
    username: "user1",
    password: "pass1",
    role: "user",
    ...overrides,
  };
}
```

24. Clean error messages and context logging

Definition: Provide actionable failure context (which data, which step) to speed up debugging.

Example:

```
export function failWithContext(
  msg: string,
  ctx: Record<string, unknown>
): never {
  throw new Error(`${msg} | context=${JSON.stringify(ctx)}`);
}
```

25. Scalability: composition over inheritance

Definition: Compose small components (Nav, Widgets, Sections) instead of deep inheritance to keep frameworks flexible.

Example:

```
import { Page } from "@playwright/test";

class Nav {
  constructor(private page: Page) {}
  async openSettings() {
    await this.page.getByRole("link", { name: "Settings" }).click();
  }
}

class SettingsPage {
  constructor(private page: Page, private nav: Nav) {}
  async goto() {
    await this.nav.openSettings();
  }
}
```