E2E Best Practices for testing WordPress themes/plugins

WordPress e2e best practices

- Forbid $, use locator instead: Avoid deprecated APIs like $, $$, $$eval that return ElementHandle; instead, use the Locator API for better performance, chaining, and compatibility with Playwright assertions, ensuring tests are more reliable in WordPress environments. see: Locator see: assertions

- Use accessible selectors: Prefer getByRole and similar accessible locators (e.g., getByLabel, getByText) to query elements based on ARIA roles and names, making tests resilient to UI changes and promoting accessibility in WordPress themes and plugins. see: getByRole

- Selectors are strict by default: Leverage Playwright's strict mode, which throws errors for ambiguous selectors (multiple matches), to write precise tests that fail fast and avoid flaky behavior in WordPress block editor interactions.

- Inline simple utility helper functions: For straightforward actions, define utility functions directly in tests rather than creating separate files, keeping code simple and reducing overhead when testing WordPress-specific features like blocks or patterns.

- Favor Page Object Model over utils: Organize reusable functions into Page Object Model classes for page-specific logic (e.g., Gutenberg editor pages), improving test maintainability and readability for complex WordPress theme/plugin workflows. see: POM

- Restify actions to clear or set states: Use REST API calls (via requestUtils.rest) to efficiently set up or reset WordPress states (e.g., posts, blocks, settings) before/after tests, avoiding slow manual UI interactions and ensuring consistent test environments. see: REST API

- Avoid global variables: Rely on Playwright fixtures to inject dependencies like page and browser into tests, enabling parallel execution and multiple tabs/contexts without conflicts in WordPress E2E scenarios. see: fixtures

- Make explicit assertions: Include clear assertions (e.g., expect(locator).toBeVisible()) throughout tests to verify states explicitly, improving test readability and debugging for WordPress plugin/theme functionality.

Playwright best practices

Testing Philosophy

- Focus on user-visible behavior: Test what the end user sees and interacts with, not internal implementation details like CSS classes or function names.
- Isolate tests: Each test should run independently with its own storage, cookies, and data. Use beforeEach hooks for common setup steps.
- Avoid testing third-party dependencies: Mock external APIs using Playwright’s route() instead of relying on external servers.

- Control your data: Use a staging environment for database tests and ensure consistency for visual regression tests.

Best Practices

Use Locators

- Prefer Playwright's built-in locators (getByRole, getByText, etc.) for resilience and auto-waiting.
- Avoid brittle selectors like XPath or CSS classes.

Generate Locators

- Use Playwright's codegen tool or VS Code extension to pick robust locators.

Use Web-First Assertions

- Assertions like await expect(locator).toBeVisible() automatically wait for conditions.
- Avoid manual checks like isVisible() which don't retry.

Debugging

- Locally: Use VS Code extension or --debug flag for live debugging.
- On CI: Use trace viewer for detailed failure analysis instead of screenshots/videos.

Leverage Playwright Tooling

- Use the vscode plugin: Test generator, trace viewer, UI Mode, and TypeScript support improve productivity and reliability.

Cross-Browser Testing

- Configure projects for Chromium, Firefox, and WebKit to ensure compatibility.

Keep Playwright Updated

- Regularly update to test against latest browser versions.

Run Tests on CI

- The vscode plugin for playwright will generate a starter .yml for e2e testing workflows on GitHub when installed.
- Integrate with CI/CD pipelines (e.g., GitHub Actions).
- Use Linux for cost efficiency and consider sharding for speed.

Optimize Browser Downloads

- Install only required browsers on CI to save time and space.

Lint Your Tests

- Use TypeScript and ESLint rules like @typescript-eslint/no-floating-promises to catch missing await.

Productivity Tips

- Use soft assertions to collect multiple failures without stopping the test immediately.