CeylonBooking — QA & Testing Pitch

> *A quick overview of how we approached quality assurance on the CeylonBooking platform.*
> *For full architecture details, see ARCHITECTURE.md.*

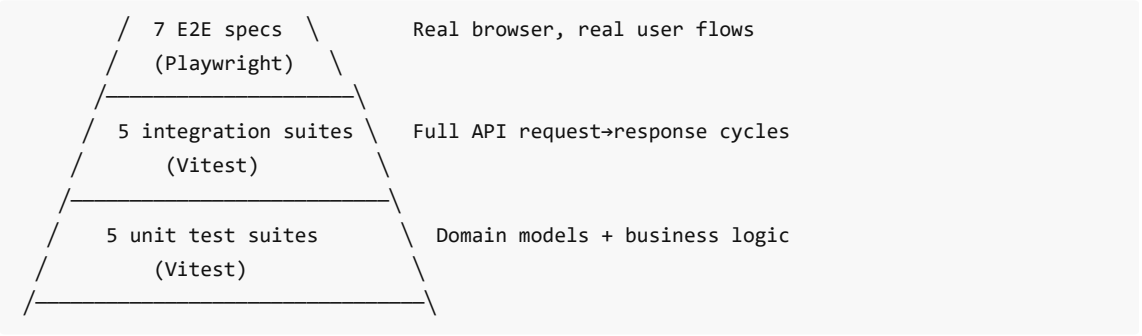
What Is CeylonBooking?

A tourism booking platform for Sri Lanka that solves three problems: manual scheduling (replaced with automated conflict detection), pricing friction (dual LKR/USD pricing for locals vs. tourists), and market fragmentation (one centralized, verified platform).

Stack: React 19 + Mantine → Hono REST API → Supabase (PostgreSQL + Auth + RLS).
(See Architecture §2–3 for the full breakdown.)

Testing Strategy: The Pyramid

We implemented a three-layer testing pyramid — heavily weighted toward fast, isolated unit tests, with targeted integration and E2E tests on top.



Why a pyramid? Unit tests run in milliseconds and catch logic bugs early. Integration tests verify that the API, database, and auth work together. E2E tests catch what neither layer can — broken UI, missing elements, bad navigation. The pyramid shape keeps the test suite fast while still covering real-world scenarios.

Unit Tests (Vitest)

5 test files in `tests/unit/` covering domain validation and business logic:

What we test	Example assertion
Zod schemas (User, Listing, Booking)	Invalid email → rejects; negative price → rejects; missing required field → rejects
SchedulingService	10 capacity − 7 booked = 3 remaining; request for 4 → unavailable
PricingService	Local user (country=LK) → LKR price; foreign user → USD price

Key decision: Domain models use Zod for schema-first validation. This means the same schema gives us runtime validation AND TypeScript types — no duplication, and every edge case we test in Zod is automatically enforced at the API boundary too.

```
npm run test:unit          # Run once
npm run test:coverage      # With V8 coverage report (text + HTML)
```

Integration Tests (Vitest)

5 test files in `tests/integration/` that hit the actual Hono API and Supabase database:

File	What it verifies
`api.test.ts`	Health check, consistent response format, error handling
`auth.test.ts`	Signup creates user + profile, login returns JWT, invalid creds → 401
`listings.test.ts`	CRUD operations, location/type filtering, auth-required endpoints
`bookings.test.ts`	Availability check, booking creation, conflict → 409 response
`security.test.ts`	Missing auth header → 401, invalid token → 401, RLS enforcement

Why separate from unit tests? Unit tests mock the database. Integration tests use a real local Supabase instance (via Docker), so they catch issues like SQL type mismatches, RLS policy bugs, and auth header handling that mocks would never reveal.

End-to-End Tests (Playwright)

This is where I invested the most thought. **7 spec files** in `frontend/tests/e2e/` simulate real user interactions in a Chromium browser.

Why Playwright?

Feature	Why it matters
Auto-waiting	No `sleep(2000)` hacks. Playwright waits for elements to be visible/clickable automatically, eliminating the #1 cause of flaky E2E tests.
Trace on retry	When a test fails and retries, Playwright records a full trace (DOM snapshots + network) you can step through.
Screenshot on failure	Automatically captures the browser state at the moment of failure — invaluable for debugging CI failures you can't reproduce locally.
Built-in web server	Playwright starts the dev server before tests and stops it after. No manual setup needed.

What We Test

Spec	Coverage
`home.spec.ts`	Hero renders, search input present, nav links work
`auth.spec.ts`	Login/signup forms have correct fields, links between them work

`navigation.spec.ts`	Multi-page navigation, auth redirects for protected routes
`dashboard.spec.ts`	Non-authenticated redirect, host-only access control
`listing.spec.ts`	Listing detail page renders, booking form elements present
`booking.spec.ts`	Booking flow from availability check to confirmation
`api.spec.ts`	API response format validation from browser context

Responsive Testing

We test **three viewport sizes** in the same suite — no separate mobile test infrastructure needed:

```
// Mobile (iPhone SE)
await page.setViewportSize({ width: 375, height: 667 });

// Tablet (iPad)
await page.setViewportSize({ width: 768, height: 1024 });

// Desktop (default Chromium viewport)
```

Auth Redirect Testing

One pattern I'm particularly proud of — we verify that protected routes enforce authentication at the UI level:

```
test('should redirect unauthenticated users from my-bookings', async ({ page }) => {
    await page.goto('/my-bookings');
    await expect(page).toHaveURL('/login');  // Redirected!
});

test('should redirect non-hosts from dashboard', async ({ page }) => {
    await page.goto('/dashboard');
    await expect(page).toHaveURL('/');  // Redirected!
});
```

This catches a common security issue: backend auth is solid, but the frontend forgets to guard a route.

Running E2E Tests

```
cd frontend
npm run test:e2e            # Headless (CI mode)
npm run test:e2e:headed     # Watch the browser
npm run test:e2e:ui         # Interactive step-through UI
```

Security Testing

Security is tested at **every layer**, not just one:

Layer	How
Schema validation	Unit tests confirm Zod rejects invalid/malicious input
Auth middleware	Integration tests send missing/invalid/expired JWTs → expect 401
Row Level Security	Integration tests verify users can't access other users' bookings
UI route guards	E2E tests confirm protected pages redirect unauthenticated users

(See Architecture §7 for the full auth flow and security layer breakdown.)

What I'd Do Next

If I had more time, these are the improvements I'd prioritize:

1. **CI pipeline** — Run all three test layers on every PR via GitHub Actions
2. **Visual regression** — Playwright screenshot comparisons to catch unintended CSS changes
3. **Load testing** — Verify the scheduling engine handles concurrent booking attempts correctly
4. **Contract testing** — Ensure frontend API expectations match backend response shapes automatically

---

Quick Reference

Command	What it does
`npm run test:unit`	Unit tests (domain + services)
`npm run test:integration`	API + DB integration tests
`npm run test:coverage`	All tests + coverage report
`cd frontend && npm run test:e2e`	Playwright E2E (headless)
`cd frontend && npm run test:e2e:ui`	Playwright interactive mode