Testing Automation

Setting up automated testing in a project, A to Z

This is the setup we used in `appointment-systemv2`, written so you can copy it into any Node/TypeScript project (API, worker, React app, kiosk app). Each step works on its own, so you can stop after any of them and already have something useful.

---

0. Before you start

0.1 Install on your machine

Tool	Version	Why	Check
Node.js	22 LTS	Runs everything below	`node -v`
npm	comes with Node	Installs the packages	`npm -v`
Docker Desktop	any recent	Runs the throwaway test database	`docker compose version`
Git	any	—	`git --version`
VS Code + the **Vitest** and **Playwright Test** extensions	optional	Run or debug a single test from the editor	—

On Windows, make sure Docker Desktop is running before you start the test DB.

0.2 The testing stack we use

Package	Version we use	What it does	Where

Vitest	5.x	The test runner: finds `*.test.ts` files and runs them. Gives `describe`/`it`/`expect`, mocking (`vi.mock`, `vi.fn`) and fake timers. Jest-compatible API, but native TypeScript/ESM support and much faster.	server + client
@vitest/coverage-v8	5.x	Coverage report (which lines ran). Same version as Vitest.	server + client
supertest	7.x	Sends real HTTP requests to your Express app in memory, no port needed: `request(app).get("/api/x")`.	server
Docker + Postgres 16	—	A real, empty database just for tests (your real DB engine, not a fake).	server
Prisma	7.x	Our ORM; `prisma db push` applies the schema to the test DB. Use your project's own ORM/migration tool instead.	server
tsx	4.x	Runs TypeScript scripts directly (seed scripts, starting the API for E2E).	server
jsdom	30.x	A fake browser DOM inside Node, so React components can render in tests.	client
React Testing Library (`@testing-library/react`, `/dom`, `/user-event`, `/jest-dom`)	16.x / 10.x / 14.x / 7.x	Render components, find elements the way a user does (by role/label/text), click and type, and assert with `toBeInTheDocument()` etc.	client
MSW (Mock Service Worker)	2.x	Fakes the backend API at the network level, so the component's real axios/fetch code runs.	client
Playwright (`@playwright/test`)	1.6x	Drives a real Chromium browser for end-to-end tests; records screenshots, video and traces when something fails.	e2e
GitHub Actions	—	Runs all of the above on every PR.	CI

Not on Node? The same structure works elsewhere: Python → **pytest** + **httpx/TestClient** + **testcontainers**; Go → `go test` + `httptest`; Flutter → `flutter_test` + `integration_test`. Playwright works for any web front end.

0.3 Install the packages in your project

```
# API / backend
cd server
npm i -D vitest @vitest/coverage-v8 supertest @types/supertest tsx

# React front end
cd ../client
npm i -D vitest @vitest/coverage-v8 jsdom msw \
  @testing-library/react @testing-library/dom @testing-library/user-event @testing-
library/jest-dom

# End-to-end (its own folder and package.json)
cd .. && mkdir e2e && cd e2e
npm init -y
npm i -D @playwright/test typescript
npx playwright install chromium        # downloads the browser (~150 MB, once per machine)
```

Keep `vitest` and `@vitest/coverage-v8` on the **same version**, or coverage won't run.

0.4 Every-day commands once it's set up

```
cd server && npm run test:db:up        # start the test DB (once per session)
cd server && npm test                  # unit + integration
cd server && npm run test:watch        # re-run on save
cd client && npm test
cd e2e && npm test                     # starts API + web itself, then runs the browser tests
cd e2e && npx playwright test --ui     # watch the browser step by step
cd e2e && npx playwright show-report   # open the last HTML report

npx vitest run path/to/file.test.ts    # one file
npx vitest run -t "part of a test name"  # one test
```

---

1. The rules (agree on these first)

1. **Every feature PR includes its tests.** The tests go in the same PR as the code.
2. **Every bug fix starts with a failing test** that reproduces the bug. Then the fix makes it pass.
3. **CI must be green to merge.**
4. **No coverage targets.** Test what would hurt if it broke: money, bookings and orders, auth, data belonging to the wrong customer/tenant, and anything that talks to hardware or payment.

2. The four kinds of test

Kind	What it checks	Speed	Tooling
Unit	One pure function (pricing, date maths, parsing)	ms	Vitest
Integration	Real HTTP request → route → middleware → DB, against a real throwaway database	~50 ms each	Vitest + supertest + Docker Postgres
Component	One React page/component with the API faked	~100 ms each	Vitest + jsdom + Testing Library + MSW
E2E	A real browser clicking through the real app and API	seconds each	Playwright

Most of your tests should be **integration** tests. They catch the bugs that matter (wrong status codes, missing tenant filters, auth mix-ups), and they're barely slower than unit tests. Keep E2E to a handful of critical journeys.

3. Folder layout

```
server/
  src/
    app.ts          ← builds the Express app (no listen!)
    index.ts        ← imports app, calls listen, starts workers
  tests/
    setup/          ← env, DB guard, global mocks
    helpers/        ← factories (create test data), auth tokens
```

```
    unit/
    integration/
  vitest.config.ts
client/
  tests/            ← outside src/, so the production build ignores it
    setup.ts
    msw.ts
    render.tsx
e2e/                ← own package.json
  tests/
  playwright.config.ts
docker-compose.test.yml
.github/workflows/test.yml
```

---

4. Step A: add the test scripts

(Install the packages first; see 0.3.) In `server/package.json`, add these scripts:

```
"test": "vitest run",
"test:unit": "vitest run --project unit",
"test:integration": "vitest run --project integration",
"test:watch": "vitest",
"test:coverage": "vitest run --coverage",
"test:db:up": "docker compose -f ../docker-compose.test.yml up -d --wait",
"test:db:down": "docker compose -f ../docker-compose.test.yml down"
```

5. Step B: make the app testable

Split building the app from starting it. Tests import the app without opening a port or starting background workers.

```
// src/app.ts
const app = express();
app.use(express.json());
app.use("/api/orders", orderRoutes);
export default app;
```

```
// src/index.ts
import app from "./app.js";
app.listen(process.env.PORT ?? 5000);
startWorkers();
```

Do the same for anything with background loops (queues, cron, sync jobs): export a `start…()` function and only call it from `index.ts`.

6. Step C: a throwaway test database

`docker-compose.test.yml`:

```
services:
  postgres-test:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: myapp_test
    ports: ["5434:5432"]          # not your dev port
    tmpfs: /var/lib/postgresql/data   # in memory: fast and always empty
    healthcheck:
      test: pg_isready -U postgres -d myapp_test
      interval: 2s
      retries: 15
```

`tests/setup/test-env.ts` forces test settings **before anything reads** `process.env`, and refuses to run against anything that isn't clearly a test DB:

```
Object.assign(process.env, {
  NODE_ENV: "test",
  TZ: "Asia/Kuala_Lumpur",          // same as production
  DB_HOST: "localhost",
  DB_PORT: "5434",
  DB_NAME: process.env.TEST_DB_NAME ?? "myapp_test",
  JWT_SECRET: "test-secret",
```

```
});

export function assertSafeTestDatabase() {
  const { DB_HOST, DB_NAME } = process.env;
  if (!["localhost", "127.0.0.1"].includes(DB_HOST!) || !DB_NAME!.endsWith("_test")) {
    throw new Error(`Refusing to run tests against ${DB_HOST}/${DB_NAME}`);
  }
}
```

> Never let tests read your real `.env`. Integration tests wipe every table.

`tests/setup/global-db.ts` pushes the schema once per run (Prisma example):

```
import { execSync } from "child_process";
export default function setup() {
  execSync("npx prisma db push --skip-generate", { stdio: "inherit" });
}
```

`tests/setup/integration.ts` empties the DB **before every test**:

```
beforeEach(async () => {
  const rows = await prisma.$queryRawUnsafe<{ tablename: string }[]>(
    `SELECT tablename FROM pg_tables WHERE schemaname='public'`);
  await prisma.$executeRawUnsafe(
    `TRUNCATE ${rows.map(r => `"${r.tablename}"`).join(",")} RESTART IDENTITY CASCADE`);
});
afterAll(() => prisma.$disconnect());
```

`vitest.config.ts`:

```
import { defineConfig } from "vitest/config";
import "./tests/setup/test-env.js";

export default defineConfig({
  test: {
    coverage: { provider: "v8", include: ["src/**/*.ts"], reporter: ["text-summary", "html",
"json-summary"] },
    projects: [
      { test: { name: "unit", include: ["tests/unit/**/*.test.ts"], setupFiles:
```

```
["tests/setup/mocks.ts"] } },
      { test: {
          name: "integration",
          include: ["tests/integration/**/*.test.ts"],
          globalSetup: ["tests/setup/global-db.ts"],
          setupFiles: ["tests/setup/integration.ts"],
          fileParallelism: false,   // one shared DB
          testTimeout: 20_000,
      } },
    ],
  },
});
```

Not on Postgres/Prisma? The idea is the same: a disposable DB in Docker, schema applied once, every table wiped before each test. SQLite can run in memory. For MongoDB use `mongodb-memory-server`.

7. Step D: factories and auth helpers

Tests should read like a story, not 40 lines of setup. Put every "create a valid X" in one file:

```
// tests/helpers/factories.ts
let seq = 0;
const next = () => `${Date.now().toString(36)}${++seq}`;

export const createOutlet = (o = {}) =>
  prisma.outlet.create({ data: { name: `Outlet ${next()}`, ...o } });

export const createUser = (o: { role?: Role; outletId?: string } = {}) =>
  prisma.user.create({ data: { email: `u-${next()}@test.local`, password: HASH, role:
"OPERATOR", ...o } });

/** The common starting point: a tenant with an operator and one bookable thing. */
export async function createShop() {
  const outlet = await createOutlet();
  const operator = await createUser({ outletId: outlet.id });
```

```
  return { outlet, operator };
}
```

```
// tests/helpers/auth.ts
export const bearer = (t: string) => ({ Authorization: `Bearer ${t}` });
export const tokenFor = (u: User) => signToken({ userId: u.id, role: u.role, outletId:
u.outletId });
```

Tips:

- Use a **fixed future date** for anything time-based (`const MONDAY = "2030-01-07"`), so "can't book in the past" rules never interfere.
- Write times with an explicit offset (`2030-01-07T10:00:00+08:00`).

8. Step E: mock everything outside your process

Tests must never send real emails, WhatsApp messages or payments, or call OpenAI.

```
// one module, per test file
vi.mock("../../src/services/email.service.js", () => ({ sendEmail:
vi.fn().mockResolvedValue(undefined) }));

// a global one for every test (tests/setup/mocks.ts), e.g. Redis
vi.mock("../../src/utils/redis.js", () => ({ default: { zadd: vi.fn(), zrem: vi.fn(),
zrangebyscore: vi.fn().mockResolvedValue([]) } }));

// code that calls fetch() directly
vi.stubGlobal("fetch", vi.fn(async () => new Response(JSON.stringify([]), { status: 200 })));
```

Then **assert on the mock**. That's how you test "we sent the right message":

```
expect(sendEmail).toHaveBeenCalledWith(expect.objectContaining({ recipient: "a@b.com" }));
```

Timers and workers: fake **only** what you need, so DB drivers keep working:

```
vi.useFakeTimers({ toFake: ["setInterval"] });  // or ["Date"] to pin "now"
startWorker();
await vi.advanceTimersByTimeAsync(15_000);
```

9. Step F: write integration tests

```
import request from "supertest";
import app from "../../src/app.js";

it("creates an order", async () => {
  const { operator } = await createShop();
  const res = await request(app).post("/api/orders").set(bearer(tokenFor(operator))).send({
items: [] });
  expect(res.status).toBe(201);
});
```

Checklist for every endpoint:

- ☐ Happy path: status code + response shape
- ☐ Validation: missing/invalid fields → 400 (not 500!)
- ☐ Auth: no token → 401, wrong token type → 401, wrong role → 403
- ☐ **Tenant isolation:** another tenant's record → 404 / not listed
- ☐ **Can't move records between tenants:** `PUT {...body, outletId: other}` must not change the owner
- ☐ Side effects: DB rows written, emails/messages sent (assert the mock)

The two tenant checks found most of our real bugs.

10. Step G: frontend component tests

(Packages: see 0.3.) Add `"test": "vitest run"` to `client/package.json`.

`vitest.config.ts`: `environment: "jsdom"`, `include: ["tests/**/*.test.{ts,tsx}"]`, `setupFiles: ["tests/setup.ts"]`.

`tests/setup.ts`:

```
import "@testing-library/jest-dom/vitest";
import { cleanup } from "@testing-library/react";
import { server } from "./msw";

beforeAll(() => server.listen({ onUnhandledRequest: "error" }));  // unknown API calls fail
the test
afterEach(() => { cleanup(); server.resetHandlers(); localStorage.clear(); });
afterAll(() => server.close());

// jsdom gaps that Radix/shadcn need
window.matchMedia ??= (q) => ({ matches: false, media: q, addEventListener() {},
removeEventListener() {} } as any);
globalThis.ResizeObserver ??= class { observe() {} unobserve() {} disconnect() {} } as any;
Element.prototype.hasPointerCapture ??= () => false;
Element.prototype.scrollIntoView ??= () => {};
```

`tests/msw.ts`: `export const server = setupServer();`

`tests/render.tsx`: a `renderWithRouter(ui, { route })` that wraps in `QueryClientProvider` + `MemoryRouter`, plus a catch-all route that renders the current path. Then you can assert "it navigated to /dashboard".

A test:

```
it("logs in and goes to the dashboard", async () => {
  let body: unknown;
  server.use(http.post("*/api/auth/login", async ({ request }) => {
    body = await request.json();
    return HttpResponse.json({ token: "t", user: { id: "u1" } });
  }));
  renderWithRouter(<LoginPage />, { path: "/login", route: "/login" });

  const user = userEvent.setup();
  await user.type(screen.getByLabelText("Email"), "a@b.com");
  await user.type(screen.getByLabelText("Password"), "secret");
  await user.click(screen.getByRole("button", { name: "Sign in" }));

  expect(await screen.findByTestId("location")).toHaveTextContent("/dashboard");
  expect(body).toEqual({ email: "a@b.com", password: "secret" });
```

```
});
```

Query like a user does: `getByRole`, `getByLabelText`, `getByText`. Avoid CSS classes and test ids where you can. If a label can't be found, the label probably isn't linked to its input (`htmlFor`/`id`), which is an accessibility bug as well.

11. Step H: E2E with Playwright

A separate package (set up in 0.3), so the browsers don't bloat the app. Add `"test": "playwright test"` to `e2e/package.json`.

`playwright.config.ts` **starts its own API and web server on unusual ports against its own seeded DB**, never your dev servers:

```
export default defineConfig({
  testDir: "./tests",
  workers: 1,                                    // one shared DB
  use: { baseURL: "http://localhost:5175", timezoneId: "Asia/Kuala_Lumpur", trace: "retain-on-
failure" },
  webServer: [
    { command: "npx tsx tests/e2e/seed.ts && npx tsx tests/e2e/start-server.ts", cwd:
"../server",
      url: "http://localhost:5055/health", env: { TEST_DB_NAME: "myapp_e2e_test" },
reuseExistingServer: false },
    { command: "npm run dev -- --port 5175 --strictPort", cwd: "../client",
      url: "http://localhost:5175", env: { API_PROXY_TARGET: "http://localhost:5055" },
reuseExistingServer: false },
  ],
});
```

Keep seed names (login email, shop name) in one shared `fixtures.ts` that both the seed script and the specs import.

Write **3–8 journeys**, e.g. "sign up → book → see it in my list", "operator logs in → creates X", "admin restricts a user → user can't see it". Do setup through the API when the UI part isn't what you're testing.

12. Step I: CI (GitHub Actions)

`.github/workflows/test.yml` with one job per layer:

```
on: { pull_request: {}, push: { branches: [main] } }
jobs:
  server:
    runs-on: ubuntu-24.04
    defaults: { run: { working-directory: server } }
    services:
      postgres:
        image: postgres:16-alpine
        env: { POSTGRES_USER: postgres, POSTGRES_PASSWORD: postgres, POSTGRES_DB: myapp_test }
        ports: ["5434:5432"]
        options: --health-cmd "pg_isready -U postgres" --health-interval 2s --health-retries
15
    steps:
      - uses: actions/checkout@v5
      - uses: actions/setup-node@v5
        with: { node-version: 22, cache: npm, cache-dependency-path: server/package-lock.json
}
      - run: npm ci
      - run: npx prisma generate     # if generated code is gitignored
      - run: npm run typecheck
      - run: npm run test:coverage
  client:
    # npm ci → lint → tsc → npm run test:coverage
  e2e:
    # postgres service + npm ci in server, client, e2e → npx playwright install --with-deps
chromium → npx playwright test
    # upload e2e/playwright-report as an artifact so failures come with screenshots and traces
```

Nice extras:

- The `github-actions` Vitest reporter: failing tests are annotated inline on the PR.
- `davelosert/vitest-coverage-report-action`: a coverage comment on each PR.

Finally: **Settings → Branches → require the checks on `main`.**

13. When a test finds a bug

Write the test for the **correct** behaviour and leave it failing. Don't use `it.skip` or `it.fails`. Add a message that says where to fix it:

```
expect(res.status,
  "BUG: booked another outlet's service. createBooking() in controllers/booking.controller.ts
must filter by outletId",
).toBe(400);
```

The failure then explains itself in the terminal, in the CI summary and on the PR. Fix it in a follow-up PR; the test going green is the proof.

14. Rolling this out on an existing project (suggested order)

1. Steps A–C (Vitest + test DB + `app.ts` split). Add one `GET /health` test and get it green.
2. Step I (CI) straight away, so nothing regresses from now on.
3. Integration tests for **auth** and the **one flow that makes money** (orders, bookings, payments).
4. Tenant-isolation checks for every endpoint that takes an id.
5. Mocks for outbound services + tests that assert what was sent.
6. Component tests for login and the main form.
7. 2–3 Playwright journeys.
8. From now on: the rules in section 1.

15. Notes for specific kinds of project

These are general pointers; adapt them to how each project is actually built.

Background services (e.g. a notification engine):

- Export the processing function (`processJob(job)`) separately from the polling loop, so it can be tested directly.
- Mock the provider (WhatsApp/email/SMS SDK). Test: success marks the job sent, failure retries with a backoff, gives up after N tries, and the same job is never sent twice.
- Webhooks: test signature/token verification (wrong token → 403) and that the handler replies 200 immediately.

- Use fake timers for schedules, and pin `Date` for "send 2 hours before".

Operations apps (e.g. a laundry system):

- Money first: price calculation, discounts never exceeding the total, refunds, idempotent payment callbacks (the same callback twice must not double-credit).
- Machine/device state transitions (idle → running → done) as unit tests on a small state function.
- Mock the payment gateway and any IoT/MQTT client, and assert the commands sent.

Offline / kiosk apps:

- Keep the sync logic in plain functions (queue, merge, conflict rules) and unit-test them heavily: offline → queue grows; back online → queue flushes once, in order; a server conflict → the defined winner.
- Fake storage: `fake-indexeddb` for IndexedDB, an in-memory SQLite for local DBs.
- Fake the network: MSW for "server down" (`HttpResponse.error()`) and slow responses.
- Electron: Playwright can drive it (`_electron.launch`). Kiosk touch flows can be E2E-tested at a fixed viewport.

16. Gotchas we hit

- **Timezone:** set `TZ` in tests and `timezoneId` in Playwright to the production timezone, or date tests pass locally and fail in CI.
- **Shared DB:** integration files must run one at a time (`fileParallelism: false`), and so must E2E (`workers: 1`).
- **Fire-and-forget code** (`void sendReminder()`) can still be running when the next test truncates the DB, which logs noisy errors. Harmless, but `await` it where you can.
- **Fake timers:** faking everything freezes DB drivers and HTTP. Fake only `setInterval`/`Date`.
- **Playwright waiting:** use `await expect(locator).toBeVisible()`, never `isVisible()` or sleeps. `isVisible()` doesn't wait.
- **Windows:** `core.autocrlf` warnings about LF/CRLF are harmless; regexes over file contents should allow `\r?\n`.
- **Unhandled requests:** keep MSW's `onUnhandledRequest: "error"`. It catches components calling endpoints you didn't expect.

Revision #2
Created 2026-10-21 03:02:21 UTC by Jagjit
Updated 2026-10-21 03:03:26 UTC by Jagjit