Testing Guide

Bilko — Testing Guide

Status: Active but partially stale — implementation moved to Kotlin/Ktor API and current Playwright partitioning is in progress **Version:** 2.1 **Last Updated:** 2026-05-21 **Author:** ALAI Documentation Team

> **Canonical testing policy:** see TEST-STRATEGY.md, E2E-TEST-PLAN.md, and DEMO-TESTING-PLAN.md. Older sections in this guide may still mention Express/Prisma/API Vitest inventory and should not override the current Ktor + Playwright policy.

Table of Contents

---

1. Testing Philosophy

Financial software has a higher correctness bar than typical web apps. Bilko's testing strategy prioritizes:

1. **Financial Logic Accuracy** — VAT calculations, double-entry bookkeeping, currency conversion
2. **Data Integrity** — No lost transactions, no balance discrepancies
3. **Regression Prevention** — Once fixed, bugs stay fixed
4. **Fast Feedback** — Prefer fast unit tests and targeted integration/contract tests; use real PostgreSQL/Testcontainers where behavior depends on the database

---

2. Testing Pyramid

```
        /\
       /E2E\        ← 10% (Critical user flows only)
      /------\
     /  Integ \     ← 20% (API endpoints, mocked DB)
    /----------\
   /    Unit    \   ← 70% (Business logic, financial engine)
  /--------------\
```

Distribution:

- **70% Unit Tests** — Fast, isolated, test business logic in `@bilko/core`
- **20% Integration Tests** — Test API endpoints with mocked Prisma client
- **10% E2E Tests** — Full API integration test (single file in `tests/e2e/`)

Coverage Targets by Module

Module	Unit	Integration	Reason
`@bilko/core` accounting engine	**95%**	N/A	Double-entry errors = financial loss
`@bilko/core` tax/VAT calculations	**95%**	N/A	Tax miscalculations = regulatory penalty
`@bilko/core` multi-currency	**90%**	N/A	FX errors = revenue leakage
Auth API	85%	**90%**	Security boundary
Invoice API	80%	**90%**	Core revenue feature

Module	Unit	Integration	Reason
Expense API	80%	**85%**	Core cost tracking
Reports API	75%	**80%**	Regulatory output
Banking API	75%	**75%**	Complex matching logic
Multi-tenant isolation	N/A	**100%**	GDPR + security critical

---

3. Tech Stack

Test Type	Framework	Purpose
Unit	Vitest	`@bilko/core` business logic (financial engine)
Integration	Vitest + Supertest	API endpoint testing with mocked Prisma
E2E	Vitest + Supertest	Full API integration (`tests/e2e/api.test.ts`)

Why Vitest (not Jest)

- Native ESM support, Vite-based — faster than Jest
- Compatible with Turborepo workspace structure
- Watch mode with HMR
- Same API as Jest (easy migration)

Why Supertest (not Postman)

- Programmatic API testing within Vitest
- Works with Express app instance directly
- No running server required

---

4. Actual Test Files

`apps/api/tests/` — Mock Suite (11 test files)

Tests with mocked Prisma — fast, no database required.

File	Tests	What It Covers
`setup.ts`	—	Shared test setup: Prisma mock, JWT helpers, test data factories
`auth.test.ts`	11	Register, login, refresh token, logout, GET /me — auth flows with mocked DB
`invoices.test.ts`	11	List, create, get, update, status change (send/pay), delete invoice endpoints
`expenses.test.ts`	9	List, create, get, update, approve/reject, delete expense endpoints
`contacts.test.ts`	9	List, create, get, update, delete contact endpoints
`accounts.test.ts`	4	List, get, create, delete chart-of-accounts endpoints
`banking.test.ts`	10	Bank account management, transaction import, reconciliation endpoints
`reports.test.ts`	9	P&L, balance sheet, VAT report, trial balance endpoints
`transactions.test.ts`	9	Transaction ledger list, filter, and detail endpoints
`country.test.ts`	27	Country plugin integration — tax rates for RS/BA/HR, invoice number formats
`chatbot.test.ts`	9	Chatbot message, history, clear history — rate limit (429) test
`invoice-gl-reversal.test.ts`	6	Invoice cancellation GL reversal — double-entry stays balanced
`new-endpoints.test.ts`	~10	Receipt, VAT export PDF/XML, dashboard, security audit log, data export

Total: ~120+ mock suite test cases

`apps/api/tests/unit/` — Unit Suite (4 test files)

Service-layer tests. Mocked Prisma. No HTTP layer.

File	Tests	What It Covers

`invoice-service-calculations.test.ts`	~15	`InvoiceService.createInvoice()` arithmetic — line totals, VAT, FX
`two-factor.test.ts`	8	`TwoFactorService` — enable, verify, disable TOTP with backup codes
`sef-submission.test.ts`	~10	`SefClient` HTTP calls, `InvoiceService.submitToSef()` fire-and-forget
`vat-calculation.test.ts`	~20	Pure VAT functions from country-rs, country-ba, country-hr packages

`apps/api/tests/e2e/` — E2E Suite (2 test files)

End-to-end workflow tests. Mocked services, no real DB required.

File	Tests	What It Covers
`api.test.ts`	—	Full Express stack integration test (live server, no mocks)
`billing-flow.e2e.test.ts`	~8	Full billing workflow: contact → invoice → send → pay → P&L → credit note

`apps/api/tests/integration/` — Real DB Suite (5 test files)

Requires `docker-compose.test.yml` PostgreSQL. Run with `npm run test:integration`.

File	Tests	What It Covers
`auth.integration.test.ts`	4	Registration + login + refresh against real DB
`invoice.integration.test.ts`	5	Full invoice CRUD lifecycle against real DB
`credit-note-gl.integration.test.ts`	3	Credit note GL entries balance in real DB
`report.integration.test.ts`	3	Reports with real seeded transactions
`tenant-isolation.integration.test.ts`	5	Cross-tenant data isolation (org A cannot read org B's data)

Grand Total: ~390 tests across 27 test files

`packages/core/tests/` — Unit Tests (5 test files)

File	Tests	What It Covers
`accounting.test.ts`	20	`validateDoubleEntry`, `createJournalEntry`, `calculateTrialBalance` — double-entry engine
`chart-of-accounts.test.ts`	32	Chart of accounts operations: account creation, hierarchy, account types
`invoicing.test.ts`	22	`generateInvoiceNumber`, `calculateInvoiceTotals`, `validateLineItem`
`multi-currency.test.ts`	24	Currency conversion, exchange rate locking, precision handling
`tax.test.ts`	23	VAT calculations for RS (20%), BA (17%), HR (25%), mixed rates, edge cases

Total: 121 unit test cases

Grand Total: ~220 tests across 14 test files

---

5. Running Tests

Run All Tests (Turborepo)

```
# From project root — runs all tests in all packages
npm run test

# Or with turbo directly
npx turbo run test
```

Run API Mock Suite

```
cd apps/api
npx vitest run
```

```
# Watch mode
npx vitest

# Specific test file
npx vitest run tests/auth.test.ts
npx vitest run tests/chatbot.test.ts

# Specific test by name pattern
npx vitest run --reporter=verbose -t "POST /api/v1/auth/login"
```

Run API Unit Suite

```
cd apps/api
npx vitest run tests/unit/

# Specific service test
npx vitest run tests/unit/two-factor.test.ts
npx vitest run tests/unit/vat-calculation.test.ts
```

Run API E2E Suite

```
cd apps/api
npx vitest run tests/e2e/

# Full billing flow
npx vitest run tests/e2e/billing-flow.e2e.test.ts
```

Run Real DB Integration Suite

Requires Docker. Uses `docker-compose.test.yml` (port 5433).

```
# Start test database
docker-compose -f docker-compose.test.yml up -d

# Run integration tests
cd apps/api
npm run test:integration
```

```
# Or directly:
npx vitest run tests/integration/

# Stop test database
docker-compose -f docker-compose.test.yml down
```

Run Core Unit Tests

```
cd packages/core
npx vitest run

# Watch mode
npx vitest

# Specific file
npx vitest run tests/tax.test.ts

# With coverage
npx vitest run --coverage
```

Run in Verbose Mode

```
npx vitest run --reporter=verbose
```

Test Without Building

Tests resolve workspace packages from TypeScript source (not `dist/`) via `vitest.config.ts` aliases. No build step required before running tests.

---

6. Test Configuration

`apps/api-express/vitest.config.ts`

```
import { defineConfig } from 'vitest/config'
import path from 'path'

export default defineConfig({
  resolve: {
    alias: {
      // Resolves workspace packages from source — no dist/ build needed
      '@bilko/core': path.resolve(__dirname, '../../packages/core/src/index.ts'),
      '@bilko/domain-rs': path.resolve(__dirname, '../../packages/domain-rs/src/index.ts'),
      '@bilko/domain-ba': path.resolve(__dirname, '../../packages/domain-ba/src/index.ts'),
      '@bilko/domain-hr': path.resolve(__dirname, '../../packages/domain-hr/src/index.ts'),
      '@bilko/database': path.resolve(__dirname, '../../packages/database/src/index.ts'),
    },
  },
  test: {
    globals: false,
    environment: 'node',
    setupFiles: [], // Setup is imported per test file via tests/setup.ts
    include: ['tests/**/*.test.ts'],
    exclude: ['node_modules', 'dist'],
  },
})
```

`packages/core/vitest.config.ts`

```
import { defineConfig } from 'vitest/config'

export default defineConfig({
  test: {
    globals: true,
    root: '.',
    include: ['tests/**/*.test.ts'],
  },
})
```

Key Configuration Differences

Setting	`apps/api-express`	`packages/core`

`globals`	`false` — imports explicit	`true` — describe/it/expect global
`setupFiles`	None (per-file import of `./setup`)	None
`aliases`	Workspace package path aliases	Not needed
`environment`	`node` (explicit)	Default node

---

7. Test Setup & Mocking

`apps/api/tests/setup.ts`

The API integration tests use a **mocked Prisma client** — tests run without a real PostgreSQL database. The setup file:

1. Sets required environment variables (JWT secrets, rate limit config)
2. Mocks the entire `src/lib/prisma` module with `vi.mock()`
3. Provides test data factories and JWT token generators

```
// Import setup (must be first import in test file)
import {
  createTestUser,
  generateTestAccessToken,
  generateTestRefreshToken,
  TEST_USER_EMAIL,
  TEST_USER_ID,
  TEST_ORG_ID,
} from './setup'
```

Mock Prisma Pattern

```
// In test file — reference the mocked prisma
import { prisma } from '../src/lib/prisma'
const mockPrisma = prisma as any

// Configure mock for a specific test
mockPrisma.user.findUnique.mockResolvedValue(testUser)
mockPrisma.user.findUnique.mockResolvedValue(null) // simulate not found
```

```
// Clear mocks between tests
beforeEach(() => {
  vi.clearAllMocks()
})
```

Auth Token Generation

```
// Generate a valid JWT for test requests
const authToken = generateTestAccessToken()

// Generate token with specific role
const adminToken = generateTestAccessToken({ role: 'admin' })
const viewerToken = generateTestAccessToken({ role: 'viewer' })

// Generate refresh token (used in cookies)
const refreshToken = generateTestRefreshToken(TEST_USER_ID)
```

Service Mocking Pattern (for route tests)

```
// Mock entire service module
vi.mock('../src/services/invoice.service', () => {
  const mockService = {
    listInvoices: vi.fn(),
    createInvoice: vi.fn(),
    // ...
  }
  return { invoiceService: mockService }
})

import { invoiceService } from '../src/services/invoice.service'
const mockInvoiceService = invoiceService as any

// Configure per test
mockInvoiceService.createInvoice.mockResolvedValue(newInvoice)

// Verify calls
expect(mockInvoiceService.createInvoice).toHaveBeenCalledWith(
  TEST_ORG_ID,
  TEST_USER_ID,
```

```
  expect.any(Object),
)
```

---

8. CI Integration

Tests run via GitHub Actions on every push and pull request. See `.github/workflows/ci.yml`.

CI Pipeline (4 parallel jobs)

```
# .github/workflows/ci.yml
on:
  push:
    branches: ['*']
  pull_request:
    branches: [main, staging]

jobs:
  lint: # npx turbo run lint
  type-check: # npx turbo run type-check (after prisma generate)
  test: # npx turbo run test (unit + integration)
  build: # npx turbo run build (needs lint, type-check, test)
```

Job Details

Job	Command	Needs Prisma Generate	Blocks
`lint`	`npx turbo run lint`	No	`build`
`type-check`	`npx turbo run type-check`	Yes	`build`
`test`	`npx turbo run test`	Yes	`build`
`build`	`npx turbo run build`	Yes	Deploy

Prisma Client Generation (required in CI)

```
- name: Generate Prisma Client
  run: npx prisma generate --schema=packages/database/prisma/schema.prisma
```

This is required before `type-check` and `test` because the API source references `@prisma/client` types.

Concurrency

```
concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true
```

Cancels in-flight CI runs on the same branch when new commits are pushed.

CI Gate Rules

Gate	Condition	Blocks
Lint	Any lint error	PR merge + build
Type check	Any TypeScript error	PR merge + build
Tests	Any test failure	PR merge + build
Build	Lint/type-check/test all pass	Deploy

---

9. Writing New Tests

API Integration Test — Pattern

```
// apps/api/tests/my-feature.test.ts
import { describe, it, expect, vi, beforeEach } from 'vitest'
import request from 'supertest'
import { generateTestAccessToken, TEST_ORG_ID, TEST_USER_ID } from './setup'
import app from '../src/app'

// Mock the service
vi.mock('../src/services/my-feature.service', () => {
  return {
    myFeatureService: {
      listItems: vi.fn(),
      createItem: vi.fn(),
    },
```

```
  }
})

import { myFeatureService } from '../src/services/my-feature.service'
const mockService = myFeatureService as any
const authToken = generateTestAccessToken()

describe('GET /api/v1/my-feature', () => {
  beforeEach(() => {
    vi.clearAllMocks()
  })

  it('returns 200 with paginated list', async () => {
    mockService.listItems.mockResolvedValue({ data: [], total: 0, page: 1, pageSize: 20 })

    const res = await request(app)
      .get('/api/v1/my-feature')
      .set('Authorization', `Bearer ${authToken}`)

    expect(res.status).toBe(200)
    expect(res.body).toHaveProperty('data')
  })

  it('returns 401 without auth', async () => {
    const res = await request(app).get('/api/v1/my-feature')
    expect(res.status).toBe(401)
  })
})
```

Core Unit Test — Pattern

```
// packages/core/tests/my-calculation.test.ts
import { describe, it, expect } from 'vitest'
import Decimal from 'decimal.js'
import { myCalculation } from '../src/my-module/index'

describe('myCalculation', () => {
  it('returns correct result for standard input', () => {
    const result = myCalculation('100', '20')
```

```
    expect(result.eq(new Decimal('20'))).toBe(true)
  })

  it('handles zero input', () => {
    const result = myCalculation('0', '20')
    expect(result.eq(new Decimal('0'))).toBe(true)
  })

  it('throws for negative amounts', () => {
    expect(() => myCalculation('-100', '20')).toThrow()
  })
})
```

Naming Conventions

```
// Describe block: HTTP method + route OR function name
describe('POST /api/v1/invoices', () => { ... })
describe('calculateInvoiceTotals', () => { ... })

// It block: be specific about scenario and expected outcome
it('returns 201 with created invoice when data is valid', ...)
it('returns 400 when customerId is missing', ...)
it('calculates Serbian VAT at 20% on 100 RSD as 20 RSD', ...)
it('throws for negative amounts', ...)
```

---

10. Coverage Reporting

Generate Coverage (core unit tests)

```
cd packages/core
npx vitest run --coverage
```

Coverage Output Format

```
File                    | % Stmts | % Branch | % Funcs | % Lines
------------------------|---------|----------|---------|--------
All files               |   XX.X  |   XX.X   |   XX.X  |  XX.X
 accounting/index.ts    |   95.0  |   90.0   |  100.0  |  95.0
 invoicing/index.ts     |   88.2  |   80.5   |   85.0  |  88.2
 tax/index.ts           |   92.0  |   88.0   |  100.0  |  92.0
```

Coverage Targets

Category	Target	Rationale
Financial logic (`@bilko/core`)	**>95%**	Critical for correctness
API routes (`apps/api-express`)	**>80%**	Integration-level coverage
Services	**>80%**	Business logic layer

---

11. Testing Best Practices

1. Arrange-Act-Assert (AAA)

```
it('creates invoice with correct totals', async () => {
  // ARRANGE
  const newInvoice = mockInvoice({ status: 'draft' })
  mockInvoiceService.createInvoice.mockResolvedValue(newInvoice)

  // ACT
  const res = await request(app)
    .post('/api/v1/invoices')
    .set('Authorization', `Bearer ${authToken}`)
    .send({ customerId: CUSTOMER_UUID, items: [...] })

  // ASSERT
  expect(res.status).toBe(201)
  expect(res.body.invoiceNumber).toBe('INV-2026-001')
})
```

2. Test Behavior, Not Implementation

```
// BAD — tests internal mock call order
expect(mockService.createInvoice.mock.calls[0][0]).toBe(orgId)

// GOOD — tests observable API behavior
expect(res.status).toBe(201)
expect(res.body).toHaveProperty('invoiceNumber')
```

3. Always Clear Mocks

```
beforeEach(() => {
  vi.clearAllMocks()
})
```

4. Test Edge Cases for Financial Logic

Always test in core unit tests:

- Zero amounts (`calculateInvoiceTotals([])`)
- Empty input arrays
- Negative values (should throw)
- Decimal precision (`quantity: '3', unitPrice: '33.33'`)
- Large amounts (overflow protection)

5. Never Hard-Code UUIDs in Tests

```
// BAD
const orgId = '123e4567-e89b-12d3-a456-426614174000'

// GOOD
const orgId = TEST_ORG_ID // from setup.ts, generated per-run
```

---

12. Debugging Tests

Run in Verbose Mode

```
npx vitest run --reporter=verbose
```

Run Single Test by Name

```
npx vitest run -t "returns 201 with created invoice"
```

Debug Mode (Node Inspector)

```
npx vitest --inspect-brk
# Then attach VS Code debugger or open chrome://inspect
```

Print Mock Call History

```
// In a test, add temporarily:
console.log(mockService.createInvoice.mock.calls)
```

VS Code Launch Configuration

```
{
  "type": "node",
  "request": "launch",
  "name": "Debug Vitest",
  "runtimeExecutable": "npx",
  "runtimeArgs": ["vitest", "--inspect-brk", "--no-coverage"],
  "console": "integratedTerminal",
  "cwd": "${workspaceFolder}/apps/api-express"
}
```

Related Documents

- Test Inventory: TEST-INVENTORY.md
- Backend Architecture: ../backend/BACKEND-ARCHITECTURE.md
- CI/CD Pipeline: ../infrastructure/CI-CD.md

Last Updated: 2026-03-02 **Status:** Active — ~390 tests across 27 files (mock + unit + E2E + real-DB suites) **Coverage Target:** >95% for `@bilko/core` financial logic, >80% for API routes

---

Revision #19
Created 2026-02-18 08:44:52 UTC by John
Updated 2026-06-07 19:43:38 UTC by John