Testing Guide

Bilko — Testing Guide

Status: NO TESTS EXIST YET — This document defines the testing strategy for implementation.

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** — Tests run in <5 minutes locally

Testing Pyramid

```
        /\
       /E2E\        ← 10% (Critical user flows)
      /------\
     /  Integ \     ← 30% (API endpoints, DB queries)
    /----------\
   /    Unit    \   ← 60% (Business logic, utilities)
  /--------------\
```

Distribution:

- **60% Unit Tests** — Fast, isolated, test business logic
- **30% Integration Tests** — Test API + database together
- **10% E2E Tests** — Test full user flows (expensive, slow)

```
graph TD
    subgraph STACK["Bilko Testing Stack"]
        direction TB

        subgraph E2E_LAYER["E2E Layer — 10% — Playwright"]
            PW1["invoice-flow.spec.ts<br/>Create → Send → Paid"]
            PW2["expense-flow.spec.ts<br/>Add → Approve → Pay"]
            PW3["report-flow.spec.ts<br/>P&L → Export PDF"]
            PW4["auth-flow.spec.ts<br/>Register → Login → Logout"]
        end

        subgraph INT_LAYER["Integration Layer — 30% — Supertest"]
            ST1["auth.routes.test.ts<br/>register / login / refresh / logout"]
            ST2["invoices.routes.test.ts<br/>CRUD + status transitions"]
            ST3["expenses.routes.test.ts<br/>CRUD + approve/pay"]
            ST4["reports.routes.test.ts<br/>P&L / trial-balance / VAT"]
            ST5["isolation.test.ts<br/>Cross-org data access prevention"]
        end

        subgraph UNIT_LAYER["Unit Layer — 60% — Vitest"]
            VT1["accounting.test.ts<br/>validateDoubleEntry,
createJournalEntry<br/>calculateTrialBalance"]
            VT2["tax.test.ts<br/>calculateVAT: RS 20% / BA 17% / HR 25%<br/>calculateCIT,
getVATRates"]
            VT3["multi-currency.test.ts<br/>convertCurrency,
lockExchangeRate<br/>calculateForexGainLoss"]
            VT4["bank-import.test.ts<br/>parseCSV, detectDuplicates"]
            VT5["chart-of-accounts.test.ts<br/>Structure validation"]
        end
    end

    E2E_LAYER --> INT_LAYER
    INT_LAYER --> UNIT_LAYER

    style E2E_LAYER fill:#ffc107,stroke:#e0a800
    style INT_LAYER fill:#fd7e14,color:#fff,stroke:#e8690b
    style UNIT_LAYER fill:#198754,color:#fff,stroke:#157347
```

Tech Stack

Test Type	Framework	Purpose
Unit	Vitest	Business logic, utilities, components
Integration	Supertest	API endpoint testing
E2E	Playwright	Browser automation, user flows
Coverage	c8 (built into Vitest)	Code coverage reporting

Why These Tools?

Vitest (not Jest)

- Faster (ESM native, Vite-based)
- Compatible with Vite/Turborepo
- Watch mode with HMR
- Same API as Jest (easy migration if needed)

Supertest (not Postman)

- Programmatic API testing
- Works with Express
- Can test without starting server

Playwright (not Cypress)

- Multi-browser (Chromium, Firefox, WebKit)
- Auto-wait (no flaky tests from race conditions)
- Parallel execution
- Video recording on failure

Unit Tests (Vitest)

Scope

Test pure functions and business logic in isolation:

- Invoice calculations (subtotal, tax, discount, total)
- VAT calculations (Serbia 20%, BiH 17%, Croatia 25%)
- Currency conversion (exchange rate locking)

- Double-entry validation (debit = credit)
- Date utilities (fiscal year, due date calculation)
- Number formatting (currency display)

File Structure

```
apps/api/src/
├── services/
│   ├── invoice.service.ts
│   └── invoice.service.test.ts  ← Unit test
├── utils/
│   ├── vat.ts
│   └── vat.test.ts  ← Unit test
```

Example: VAT Calculation Test

```
// apps/api/src/utils/vat.test.ts
import { describe, it, expect } from 'vitest';
import { calculateVAT } from './vat';

describe('calculateVAT', () => {
  it('calculates Serbia VAT (20%)', () => {
    const result = calculateVAT(100, 20);
    expect(result).toBe(20);
  });

  it('calculates BiH VAT (17%)', () => {
    const result = calculateVAT(100, 17);
    expect(result).toBe(17);
  });

  it('calculates Croatia VAT (25%)', () => {
    const result = calculateVAT(100, 25);
    expect(result).toBe(25);
  });

  it('handles zero VAT', () => {
    const result = calculateVAT(100, 0);
    expect(result).toBe(0);
```

```
  });

  it('handles decimal amounts', () => {
    const result = calculateVAT(123.45, 20);
    expect(result).toBe(24.69);
  });

  it('rounds to 2 decimal places', () => {
    const result = calculateVAT(10.01, 20);
    expect(result).toBe(2.00); // Not 2.002
  });
});
```

Running Unit Tests

```
# Run all unit tests
npm run test:unit

# Watch mode (re-run on file change)
npm run test:unit -- --watch

# Coverage report
npm run test:unit -- --coverage

# Specific file
npm run test:unit -- vat.test.ts
```

Coverage Requirements

Category	Target	Rationale
Financial logic	>95%	Critical for correctness
Utilities	>90%	Reused across codebase
Services	>80%	Business logic layer
Controllers	>60%	Thin layer (tested via integration)
Overall	>80%	Industry standard

Integration Tests (Supertest)

Scope

Test API endpoints with real database:

- Auth flow (register, login, refresh, logout)
- CRUD operations (invoices, expenses, contacts)
- Data validation (Zod schemas)
- Error handling (400, 401, 403, 404, 500)
- Database transactions
- Organization scoping (can't access other org's data)

File Structure

```
apps/api/src/
├── routes/
│   ├── auth.routes.ts
│   └── auth.routes.test.ts  ← Integration test
├── routes/
│   ├── invoices.routes.ts
│   └── invoices.routes.test.ts  ← Integration test
```

Test Database Setup

Use separate test database:

```
# .env.test
DATABASE_URL=postgresql://bilko_test:bilko_test@localhost:5432/bilko_test
```

Setup/teardown:

```
// apps/api/src/test/setup.ts
import { PrismaClient } from '@prisma/client';
import { beforeAll, afterAll, beforeEach } from 'vitest';

const prisma = new PrismaClient();

beforeAll(async () => {
```

```
  // Run migrations on test DB
  await execSync('npx prisma migrate deploy');
});

beforeEach(async () => {
  // Clear all tables before each test
  await prisma.$transaction([
    prisma.invoice.deleteMany(),
    prisma.expense.deleteMany(),
    prisma.contact.deleteMany(),
    prisma.user.deleteMany(),
    prisma.organization.deleteMany(),
  ]);
});

afterAll(async () => {
  await prisma.$disconnect();
});
```

Example: Invoice API Test

```
// apps/api/src/routes/invoices.routes.test.ts
import { describe, it, expect, beforeEach } from 'vitest';
import request from 'supertest';
import { app } from '../app';
import { prisma } from '../lib/prisma';

describe('POST /api/v1/invoices', () => {
  let authToken: string;
  let organizationId: string;
  let customerId: string;

  beforeEach(async () => {
    // Create test organization
    const org = await prisma.organization.create({
      data: {
        name: 'Test Company',
        baseCurrency: 'RSD',
        country: 'RS',
```

```
    },
  });
  organizationId = org.id;

  // Create test user
  const user = await prisma.user.create({
    data: {
      organizationId,
      email: 'test@bilko.io',
      passwordHash: '$2b$12$...', // bcrypt hash
      fullName: 'Test User',
      role: 'admin',
    },
  });

  // Login to get token
  const loginRes = await request(app)
    .post('/api/v1/auth/login')
    .send({ email: 'test@bilko.io', password: 'test123' });
  authToken = loginRes.body.accessToken;

  // Create test customer
  const customer = await prisma.contact.create({
    data: {
      organizationId,
      type: 'customer',
      name: 'Test Customer',
      email: 'customer@example.com',
    },
  });
  customerId = customer.id;
});

it('creates invoice with valid data', async () => {
  const res = await request(app)
    .post('/api/v1/invoices')
    .set('Authorization', `Bearer ${authToken}`)
    .send({
      customerId,
```

```
      invoiceDate: '2026-02-20',
      dueDate: '2026-03-20',
      currencyCode: 'RSD',
      items: [
        {
          description: 'Web Development',
          quantity: 10,
          unitPrice: 5000,
          taxRate: 20,
        },
      ],
    });

  expect(res.status).toBe(201);
  expect(res.body.invoiceNumber).toMatch(/^INV-\d+$/);
  expect(res.body.subtotal).toBe(50000);
  expect(res.body.taxAmount).toBe(10000);
  expect(res.body.totalAmount).toBe(60000);
});

it('rejects invoice without auth', async () => {
  const res = await request(app)
    .post('/api/v1/invoices')
    .send({ customerId, items: [] });

  expect(res.status).toBe(401);
});

it('rejects invoice for customer in different org', async () => {
  // Create another org
  const otherOrg = await prisma.organization.create({
    data: { name: 'Other Company', baseCurrency: 'EUR', country: 'RS' },
  });

  // Create customer in other org
  const otherCustomer = await prisma.contact.create({
    data: {
      organizationId: otherOrg.id,
      type: 'customer',
```

```
        name: 'Other Customer',
      },
    });

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

    expect(res.status).toBe(403); // Forbidden (can't access other org's data)
  });
});
```

Running Integration Tests

```
# Run all integration tests
npm run test:integration

# Specific file
npm run test:integration -- invoices.routes.test.ts
```

E2E Tests (Playwright)

Scope

Test critical user flows from browser:

- **Invoice Flow:** Create → Preview → Send → Mark Paid
- **Expense Flow:** Add → Upload Receipt → Approve → Pay
- **Report Flow:** Generate P&L → Export PDF
- **Auth Flow:** Register → Login → 2FA → Logout

File Structure

```
apps/e2e/
├── tests/
│   ├── invoice-flow.spec.ts
│   ├── expense-flow.spec.ts
│   ├── report-flow.spec.ts
│   └── auth-flow.spec.ts
├── fixtures/
│   └── test-data.ts
└── playwright.config.ts
```

Configuration

```
// apps/e2e/playwright.config.ts
import { defineConfig } from '@playwright/test';

export default defineConfig({
  testDir: './tests',
  timeout: 60000, // 60s per test
  retries: 1, // Retry flaky tests once
  workers: 4, // Run 4 tests in parallel
  use: {
    baseURL: 'http://localhost:3000',
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
  },
  projects: [
    { name: 'chromium', use: { browserName: 'chromium' } },
    { name: 'firefox', use: { browserName: 'firefox' } },
    { name: 'webkit', use: { browserName: 'webkit' } },
  ],
});
```

Example: Invoice E2E Test

```
// apps/e2e/tests/invoice-flow.spec.ts
import { test, expect } from '@playwright/test';

test.describe('Invoice Flow', () => {
```

```
test.beforeEach(async ({ page }) => {
  // Login
  await page.goto('/login');
  await page.fill('input[name="email"]', 'demo@bilko.io');
  await page.fill('input[name="password"]', 'demo123');
  await page.click('button[type="submit"]');
  await expect(page).toHaveURL('/dashboard');
});

test('create invoice and mark as paid', async ({ page }) => {
  // Navigate to invoices
  await page.click('a[href="/invoices"]');
  await expect(page).toHaveURL('/invoices');

  // Click "New Invoice"
  await page.click('button:has-text("New Invoice")');
  await expect(page).toHaveURL('/invoices/new');

  // Fill invoice form (6-step wizard)
  // Step 1: Customer
  await page.selectOption('select[name="customerId"]', { label: 'Acme Corp' });
  await page.click('button:has-text("Next")');

  // Step 2: Details
  await page.fill('input[name="invoiceDate"]', '2026-02-20');
  await page.fill('input[name="dueDate"]', '2026-03-20');
  await page.click('button:has-text("Next")');

  // Step 3: Items
  await page.fill('input[name="items.0.description"]', 'Web Development');
  await page.fill('input[name="items.0.quantity"]', '10');
  await page.fill('input[name="items.0.unitPrice"]', '5000');
  await page.selectOption('select[name="items.0.taxRate"]', '20');
  await page.click('button:has-text("Next")');

  // Step 4: Review
  await expect(page.locator('text=Subtotal')).toContainText('50,000.00 RSD');
  await expect(page.locator('text=Tax')).toContainText('10,000.00 RSD');
  await expect(page.locator('text=Total')).toContainText('60,000.00 RSD');
```

```
    await page.click('button:has-text("Create Invoice")');

    // Verify redirect to invoice detail
    await expect(page).toHaveURL(/\/invoices\/[a-f0-9-]+$/);
    await expect(page.locator('h1')).toContainText('INV-');

    // Mark as paid
    await page.click('button:has-text("Mark as Paid")');
    await page.click('button:has-text("Confirm")');

    // Verify status changed
    await expect(page.locator('.status-badge')).toContainText('Paid');
  });

  test('validates required fields', async ({ page }) => {
    await page.goto('/invoices/new');

    // Try to submit without customer
    await page.click('button:has-text("Next")');

    // Verify error message
    await expect(page.locator('.error')).toContainText('Customer is required');
  });
});
```

Running E2E Tests

```
# Start dev server first
npm run dev

# In another terminal:
npm run test:e2e

# Headless (CI mode)
npm run test:e2e -- --headed

# Debug mode (pause on failure)
npm run test:e2e -- --debug
```

```
# Specific browser
npm run test:e2e -- --project=firefox
```

VAT Test Matrix — Country Coverage

```
graph TD
    VAT["calculateVAT Tests<br/>@bilko/core/tax"]

    VAT --> RS["Serbia RS<br/>Standard: 20%<br/>Reduced: 10%<br/>Zero: 0% exports<br/>CIT: 15% flat<br/>Pausal: &lt; 6M RSD<br/>VAT reg: &gt;= 8M RSD"]

    VAT --> BA["Bosnia BA<br/>Standard: 17%<br/>No reduced rate<br/>Zero: 0% exports<br/>FBiH CIT: 10%<br/>RS CIT: 10%<br/>WHT FBiH: 5% div<br/>VAT reg: &gt;= 100K BAM"]

    VAT --> HR["Croatia HR<br/>Standard: 25%<br/>Reduced: 13%<br/>Super-red: 5%<br/>CIT small: 10%<br/>CIT large: 18%<br/>VAT reg: &gt;= 60K EUR"]

    RS --> RS_T["Test: calculateSerbianPDV<br/>Test: qualifiesForPausalRegime<br/>Test: requiresVATRegistration RS"]
    BA --> BA_T["Test: calculateBosnianPDV<br/>Test: calculateCITFBiH / CITRS<br/>Test: calculateDividendWHT"]
    HR --> HR_T["Test: calculateCroatianPDV<br/>Test: calculateCroatianCIT<br/>Test: requiresVATRegistration HR"]

    style VAT fill:#0d6efd,color:#fff
    style RS fill:#c0392b,color:#fff
    style BA fill:#2c3e50,color:#fff
    style HR fill:#e74c3c,color:#fff
```

Test Data Management

Factories (Recommended)

Create reusable test data generators:

```
// apps/api/src/test/factories/invoice.factory.ts
import { faker } from '@faker-js/faker';
import { prisma } from '../../lib/prisma';

export async function createInvoice(overrides = {}) {
  return prisma.invoice.create({
    data: {
      organizationId: faker.string.uuid(),
      customerId: faker.string.uuid(),
      invoiceNumber: `INV-${faker.number.int({ min: 1000, max: 9999 })}`,
      invoiceDate: faker.date.recent(),
      dueDate: faker.date.future(),
      currencyCode: 'RSD',
      subtotal: 50000,
      taxAmount: 10000,
      totalAmount: 60000,
      baseAmount: 60000,
      status: 'draft',
      ...overrides,
    },
  });
}
```

Usage:

```
const invoice = await createInvoice({ status: 'paid' });
```

Coverage Reporting

Generate Coverage Report

```
npm run test:unit -- --coverage
```

Output:

```
File             | % Stmts | % Branch | % Funcs | % Lines
-----------------|---------|----------|---------|--------
```

```
All files        |   82.5  |   75.3   |   80.1  |  82.5
 vat.ts          |   95.0  |   90.0   |  100.0  |  95.0
 invoice.ts      |   88.2  |   80.5   |   85.0  |  88.2
 currency.ts     |   78.0  |   70.0   |   75.0  |  78.0
```

Coverage Thresholds (CI)

Fail build if coverage drops below threshold:

```
// vitest.config.ts
export default defineConfig({
  test: {
    coverage: {
      provider: 'c8',
      reporter: ['text', 'json', 'html'],
      statements: 80,
      branches: 75,
      functions: 80,
      lines: 80,
    },
  },
});
```

Testing Best Practices

1. Test Behavior, Not Implementation

 Bad: Test internal state

```
it('sets status to paid', () => {
  invoice.status = 'paid';
  expect(invoice.status).toBe('paid');
});
```

 Good: Test observable behavior

```
it('marks invoice as paid', async () => {
  await invoiceService.markAsPaid(invoice.id);
  const updated = await prisma.invoice.findUnique({ where: { id: invoice.id } });
  expect(updated.status).toBe('paid');
  expect(updated.paidAt).toBeTruthy();
});
```

---

2. Use Descriptive Test Names

 Bad: Vague test name

```
it('works', () => { /* ... */ });
```

 Good: Descriptive test name

```
it('calculates Serbian VAT at 20% on €100 as €20', () => { /* ... */ });
```

---

3. Arrange-Act-Assert (AAA)

```
it('creates invoice with correct totals', async () => {
  // ARRANGE — Set up test data
  const customer = await createCustomer();
  const invoiceData = { customerId: customer.id, items: [...] };

  // ACT — Perform action
  const invoice = await invoiceService.create(invoiceData);

  // ASSERT — Verify outcome
  expect(invoice.subtotal).toBe(50000);
  expect(invoice.taxAmount).toBe(10000);
  expect(invoice.totalAmount).toBe(60000);
});
```

---

4. Test Edge Cases

Always test:

- **Empty input** — `calculateVAT(0, 20)`
- **Null/undefined** — `formatCurrency(null)`
- **Negative numbers** — `calculateDiscount(100, -10)`
- **Large numbers** — `convertCurrency(999999999999.9999, 1.2)`
- **Boundary values** — Tax rate at 0%, 100%

---

5. Avoid Test Interdependence

 Bad: Tests depend on each other

```
let invoiceId;

it('creates invoice', async () => {
  const invoice = await createInvoice();
  invoiceId = invoice.id; // Shared state
});

it('updates invoice', async () => {
  await updateInvoice(invoiceId); // Depends on previous test
});
```

 Good: Tests are independent

```
it('creates invoice', async () => {
  const invoice = await createInvoice();
  expect(invoice.id).toBeTruthy();
});

it('updates invoice', async () => {
  const invoice = await createInvoice(); // Create fresh data
  await updateInvoice(invoice.id);
});
```

CI/CD Integration

Tests run automatically on every push (GitHub Actions):

```
# .github/workflows/main.yml
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - run: npm run test:unit -- --coverage
      - run: npm run test:integration
      - run: npm run test:e2e
```

```
flowchart LR
    GIT["git push"] --> GHA["GitHub Actions"]

    GHA --> PARALLEL["Parallel Jobs"]

    PARALLEL --> U["unit-tests<br/>vitest run<br/>@bilko/core<br/>~30s"]
    PARALLEL --> I["integration-tests<br/>supertest<br/>postgres:15<br/>~2min"]
    PARALLEL --> E["e2e-tests<br/>playwright<br/>all browsers<br/>~5min"]

    U --> COV{"Coverage<br/>>80%?"}
    I --> API{"All API<br/>tests pass?"}
    E --> E2E{"All flows<br/>pass?"}

    COV -->|Pass| GATE["Merge Gate"]
    API -->|Pass| GATE
    E2E -->|Pass| GATE

    COV -->|Fail| BLOCK1["Block PR"]
    API -->|Fail| BLOCK1
    E2E -->|Fail| BLOCK1

    GATE --> MAIN["Merge to main"]
    MAIN --> DEPLOY["Deploy to Staging"]

    style GIT fill:#6c757d,color:#fff
    style MAIN fill:#198754,color:#fff
    style BLOCK1 fill:#dc3545,color:#fff
    style DEPLOY fill:#0d6efd,color:#fff
```

See CI-CD.md for full pipeline.

Debugging Tests

Unit/Integration Tests (Vitest)

```
# Debug mode (pause on debugger statement)
npm run test:unit -- --inspect-brk

# VS Code launch.json:
{
  "type": "node",
  "request": "launch",
  "name": "Debug Vitest",
  "runtimeExecutable": "npm",
  "runtimeArgs": ["run", "test:unit", "--", "--inspect-brk"],
  "console": "integratedTerminal"
}
```

E2E Tests (Playwright)

```
# Debug mode (opens inspector)
npm run test:e2e -- --debug

# Headed mode (see browser)
npm run test:e2e -- --headed

# Trace viewer (after failure)
npx playwright show-trace trace.zip
```

Performance Testing (Future)

Load Testing (k6)

Test API under load:

```
// apps/e2e/load/invoices.js
import http from 'k6/http';
import { check } from 'k6';

export let options = {
  vus: 100, // 100 virtual users
  duration: '30s',
};

export default function () {
  const res = http.get('http://localhost:4000/api/v1/invoices');
  check(res, { 'status is 200': (r) => r.status === 200 });
}
```

Run:

```
k6 run apps/e2e/load/invoices.js
```

Target: API handles 1,000 requests/second with <200ms p95 latency.

Status: PLANNED (Phase 2)

Related Documents

- CI/CD Pipeline: ../infrastructure/CI-CD.md
- Test Inventory: TEST-INVENTORY.md
- Security Testing: ../security/SECURITY-ARCHITECTURE.md

Last Updated: 2026-02-20 **Status:** NO TESTS EXIST YET — Implement tests during backend development **Coverage Target:** >80% overall, >95% for financial logic

---

Revision #4
Created 2026-02-23 10:48:09 UTC by John
Updated 2026-05-31 20:02:44 UTC by John