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 only)
      /------\
     /  Integ \     ← 20% (API endpoints, DB queries)
    /----------\
   /    Unit    \   ← 70% (Business logic, utilities, financial engine)
  /--------------\
```

Distribution:

- **70% Unit Tests** — Fast, isolated, test business logic. Financial software demands this ratio because the accounting engine (`@bilko/core`) has complex pure functions (VAT, double-entry, currency conversion) where bugs are expensive.
- **20% Integration Tests** — Test API + real PostgreSQL together. Cover auth flows, RBAC, org isolation, transaction creation.

- **10% E2E Tests** — Full browser flows via Playwright. Reserve for critical user-facing journeys: invoice lifecycle, expense approval, VAT report.

Rationale for 70/20/10 (not 60/30/10):

- Financial calculation bugs have real-money consequences → more unit tests on the math
- Integration tests are slow (DB setup ~2s each) → minimize to essential endpoint coverage
- E2E tests are brittle and expensive → only run on critical flows that span full stack

Coverage Targets by Module

Module	Unit	Integration	E2E	Reason
`@bilko/core` accounting engine	**95%**	N/A	N/A	Double-entry errors = financial loss
`@bilko/core` tax/VAT calculations	**95%**	N/A	N/A	Tax miscalculations = regulatory penalty
`@bilko/core` multi-currency	**90%**	N/A	N/A	FX errors = revenue leakage
Auth API (login, register, 2FA, refresh)	85%	**90%**	1 flow	Security boundary
Invoice API (CRUD, lifecycle, calculations)	80%	**90%**	2 flows	Core revenue feature
Expense API (CRUD, approval, payment)	80%	**85%**	1 flow	Core cost tracking
Reports API (P&L, VAT, trial balance)	75%	**80%**	1 flow	Regulatory output
Banking API (import, reconciliation)	75%	**75%**	1 flow	Complex matching logic
Frontend UI components	N/A	N/A	**70%**	Smoke tests for critical UI
Multi-tenant isolation	N/A	**100%**	N/A	GDPR + security critical
Security middleware (RBAC, rate limit)	90%	**90%**	N/A	Auth boundary tests

---

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
```

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 and pull request via GitHub Actions. Three parallel jobs prevent the test suite from becoming a bottleneck.

GitHub Actions Configuration

```
# .github/workflows/test.yml
name: Test Suite

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  # ──────────────────────────────────────────
  # Job 1: Unit Tests — no DB, fast feedback
  # ──────────────────────────────────────────
  unit-tests:
    name: Unit Tests
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - run: npm ci

      - name: Run unit tests with coverage
        run: cd packages/core && npx vitest run --coverage
        env:
          NODE_ENV: test

      - name: Upload coverage report
        uses: codecov/codecov-action@v4
        with:
```

```
        files: packages/core/coverage/lcov.info
        flags: unit
        fail_ci_if_error: true

    - name: Assert coverage thresholds
      run: |
        # Fails build if coverage drops below minimums
        cd packages/core && npx vitest run --coverage \
          --coverage.thresholds.statements=90 \
          --coverage.thresholds.branches=85 \
          --coverage.thresholds.functions=90

# ──────────────────────────────────────────
# Job 2: Integration Tests — real DB
# ──────────────────────────────────────────
integration-tests:
  name: Integration Tests
  runs-on: ubuntu-latest
  services:
    postgres:
      image: postgres:15
      env:
        POSTGRES_DB: bilko_test
        POSTGRES_USER: bilko_test
        POSTGRES_PASSWORD: test_password
      ports:
        - 5432:5432
      options: >-
        --health-cmd pg_isready
        --health-interval 10s
        --health-timeout 5s
        --health-retries 5
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: '20'
        cache: 'npm'
```

```
      - run: npm ci

      - name: Apply DB migrations to test DB
        run: npx prisma migrate deploy
        working-directory: packages/database
        env:
          DATABASE_URL: postgresql://bilko_test:test_password@localhost:5432/bilko_test

      - name: Run integration tests
        run: npm run test:integration
        working-directory: apps/api
        env:
          DATABASE_URL: postgresql://bilko_test:test_password@localhost:5432/bilko_test
          JWT_PRIVATE_KEY: ${{ secrets.TEST_JWT_PRIVATE_KEY }}
          JWT_PUBLIC_KEY: ${{ secrets.TEST_JWT_PUBLIC_KEY }}
          FIELD_ENCRYPTION_KEY:
"0000000000000000000000000000000000000000000000000000000000000000"
          NODE_ENV: test

  # ──────────────────────────────────────────
  # Job 3: E2E Tests — full stack
  # ──────────────────────────────────────────
  e2e-tests:
    name: E2E Tests
    runs-on: ubuntu-latest
    needs: [unit-tests, integration-tests]   # Only run if unit + integration pass
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - run: npm ci

      - name: Install Playwright browsers
        run: npx playwright install --with-deps chromium

      - name: Start dev server
```

```
      run: npm run dev &
      env:
        NODE_ENV: test

    - name: Wait for server to start
      run: npx wait-on http://localhost:3000 --timeout 60000

    - name: Run E2E tests
      run: npm run test:e2e -- --project=chromium
      env:
        PLAYWRIGHT_BASE_URL: http://localhost:3000

    - name: Upload Playwright artifacts on failure
      if: failure()
      uses: actions/upload-artifact@v4
      with:
        name: playwright-results
        path: |
          playwright-report/
          test-results/
        retention-days: 7

# ──────────────────────────────────────────
# Job 4: Security & Dependency Audit
# ──────────────────────────────────────────
security-audit:
  name: Security Audit
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: '20'
        cache: 'npm'
    - run: npm ci
    - name: Audit dependencies for vulnerabilities
      run: npm audit --audit-level=high
      # Fails on HIGH or CRITICAL CVEs
    - name: Scan for committed secrets
```

```
        uses: trufflesecurity/trufflehog@main
        with:
          path: ./
          base: ${{ github.event.repository.default_branch }}
```

CI Gate Rules

Gate	Condition	Blocks
Unit tests	Any test failure	PR merge
Unit coverage	Below threshold (90% core)	PR merge
Integration tests	Any test failure	PR merge
E2E tests	Any critical flow failure	PR merge
npm audit	HIGH or CRITICAL CVE found	PR merge
Secret scan	Credentials detected in code	PR merge

See CI-CD.md for full pipeline including deployment gates.

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
```

Mocking Strategy

API Mocking — MSW (Mock Service Worker)

For the Next.js frontend, use **MSW** to intercept API calls in tests without running the backend. MSW runs in Node.js for Vitest component tests and in the browser for Playwright E2E tests.

Why MSW over manual mocking:

- Intercepts at the network level (not module mocking) — more realistic
- Same mock handlers work for both unit/component and E2E tests
- Request/response inspection built-in

Setup — MSW Handlers

```
// apps/web/src/mocks/handlers.ts
import { http, HttpResponse } from 'msw';

export const handlers = [
  // Invoice list
  http.get('/api/v1/invoices', ({ request }) => {
    const url = new URL(request.url);
    const status = url.searchParams.get('status');

    return HttpResponse.json({
      data: mockInvoices.filter(i => !status || i.status === status),
      meta: { total: mockInvoices.length, page: 1, limit: 20 },
    });
  }),
```

```
  // Create invoice
  http.post('/api/v1/invoices', async ({ request }) => {
    const body = await request.json();
    const invoice = {
      id: crypto.randomUUID(),
      invoiceNumber: `INV-2026-001`,
      status: 'draft',
      subtotal: body.items.reduce((sum: number, i: any) => sum + i.quantity * i.unitPrice, 0),
      taxAmount: 0, // calculated separately
      totalAmount: 0,
      ...body,
    };
    return HttpResponse.json(invoice, { status: 201 });
  }),

  // Auth
  http.post('/api/v1/auth/login', async ({ request }) => {
    const body = await request.json();
    if (body.email === 'demo@bilko.io' && body.password === 'demo123') {
      return HttpResponse.json({
        accessToken: 'mock-access-token',
        user: { id: 'user-1', email: 'demo@bilko.io', role: 'owner' },
      });
    }
    return HttpResponse.json({ error: 'Invalid credentials' }, { status: 401 });
  }),
];
```

```
// apps/web/src/mocks/server.ts (for Node.js / Vitest)
import { setupServer } from 'msw/node';
import { handlers } from './handlers';

export const server = setupServer(...handlers);

// In Vitest setup file:
// beforeAll(() => server.listen());
// afterEach(() => server.resetHandlers());
// afterAll(() => server.close());
```

Override Handlers Per Test

```
import { server } from '../mocks/server';
import { http, HttpResponse } from 'msw';

test('shows error when invoice creation fails', async () => {
  // Override for this test only
  server.use(
    http.post('/api/v1/invoices', () => {
      return HttpResponse.json({ error: 'Server error' }, { status: 500 });
    })
  );

  // Render component and assert error is shown
  render(<CreateInvoiceForm />);
  await userEvent.click(screen.getByText('Create'));
  expect(await screen.findByText('Server error')).toBeInTheDocument();
});
```

Test Fixtures — Data Factories

Use typed factory functions for repeatable test data. Never hard-code UUIDs or dates in tests.

```
// apps/api/src/test/fixtures/invoice.fixtures.ts
import { randomUUID } from 'crypto';
import Decimal from 'decimal.js';

interface InvoiceFixtureOptions {
  status?: 'draft' | 'sent' | 'paid' | 'overdue' | 'cancelled';
  currencyCode?: string;
  organizationId?: string;
  customerId?: string;
  itemCount?: number;
}

export function makeInvoiceFixture(opts: InvoiceFixtureOptions = {}) {
  const orgId = opts.organizationId ?? randomUUID();
  const items = Array.from({ length: opts.itemCount ?? 1 }, (_, i) => ({
    id: randomUUID(),
    description: `Service line ${i + 1}`,
    quantity: new Decimal(1),
```

```
    unitPrice: new Decimal('10000.0000'),
    taxRate: new Decimal('20'),
    lineTotal: new Decimal('10000.0000'),
    taxAmount: new Decimal('2000.0000'),
  }));

  return {
    id: randomUUID(),
    organizationId: orgId,
    customerId: opts.customerId ?? randomUUID(),
    invoiceNumber: `INV-2026-001`,
    invoiceDate: new Date('2026-02-01'),
    dueDate: new Date('2026-03-01'),
    currencyCode: opts.currencyCode ?? 'RSD',
    status: opts.status ?? 'draft',
    subtotal: new Decimal('10000.0000'),
    taxAmount: new Decimal('2000.0000'),
    totalAmount: new Decimal('12000.0000'),
    baseAmount: new Decimal('12000.0000'),
    exchangeRate: new Decimal('1.000000'),
    notes: null,
    items,
  };
}

export function makePaidInvoiceFixture(opts = {}) {
  return {
    ...makeInvoiceFixture(opts),
    status: 'paid' as const,
    paidAt: new Date('2026-02-15'),
  };
}
```

```
// Use in tests
const draftInvoice = makeInvoiceFixture({ status: 'draft', currencyCode: 'EUR' });
const paidInvoice = makePaidInvoiceFixture({ organizationId: 'org-123' });
```

Mocking External Services (SEF, FINA)

For e-invoice integrations, mock at the HTTP level using MSW or `nock`:

```
// apps/api/src/test/mocks/sef.mock.ts
import nock from 'nock';

export function mockSEFSuccess(invoiceId: string) {
  nock('https://efaktura.mfin.gov.rs')
    .post('/api/v3/outgoing-invoice')
    .reply(200, {
      Status: 'Approved',
      Id: `SEF-${invoiceId}`,
      SalesInvoiceId: invoiceId,
    });
}

export function mockSEFError(statusCode: number, error: string) {
  nock('https://efaktura.mfin.gov.rs')
    .post('/api/v3/outgoing-invoice')
    .reply(statusCode, { error });
}

// In test:
test('SEF submission failure is handled gracefully', async () => {
  mockSEFError(500, 'Internal Server Error');
  const result = await sefService.submitInvoice(invoice);
  expect(result.status).toBe('failed');
  expect(result.retryScheduled).toBe(true);
});
```

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 #3
Created 2026-02-24 22:50:55 UTC by John
Updated 2026-05-31 20:04:02 UTC by John