Coding Standards

Coding Standards

> **Project:** Bilko **Version:** 0.1 **Date:** 2026-02-23 **Author:** Ops Architect **Status:** Draft **Reviewers:** Tech Lead, Alem Bašić

Document History

Version	Date	Author	Changes
0.1	2026-02-23	Ops Architect	Initial draft

---

1. Language & Type Safety

TypeScript Strict Mode (Required)

All code must compile with TypeScript strict mode. No exceptions.

```
// tsconfig.json
{
  "compilerOptions": {
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true
  }
}
```

Rules:

- No `any` type without an inline comment explaining why: `// eslint-disable-next-line @typescript-eslint/no-explicit-any`
- Prefer `unknown` over `any` for untyped inputs (Zod validate before using)
- Use interface for object shapes, type for unions/intersections
- All function parameters and return types must be typed

```
// WRONG
function calculateVAT(amount: any, rate: any) {
  return amount * rate / 100;
}

// CORRECT
function calculateVAT(amount: Decimal, rate: number): Decimal {
  return amount.mul(rate).div(100);
}
```

---

2. Financial Logic (Non-Negotiable Rules)

These rules cannot be overridden by any other consideration.

Rule F1: NEVER Use JavaScript `number` for Money

```
// WRONG — IEEE 754 float precision errors
const vatAmount = 123.45 * 0.20;  // 24.690000000000003

// CORRECT — decimal.js exact arithmetic
import { Decimal } from 'decimal.js';
const vatAmount = new Decimal('123.45').mul('0.20');  // 24.6900
```

Rule F2: All Monetary Values Are Strings in API Responses

Prisma returns `NUMERIC` columns as strings. Keep them as strings. Never `.toNumber()` monetary values.

```
// WRONG — loses NUMERIC precision
const invoice = await prisma.invoice.findUnique({ where: { id } });
return { totalAmount: invoice.totalAmount.toNumber() };  // NEVER

// CORRECT — return as string, JSON serializes fine
return { totalAmount: invoice.totalAmount.toString() };
```

Rule F3: Double-Entry Is Always Enforced

Every financial event creates equal debit and credit entries. No exceptions.

```
// CORRECT pattern for creating a transaction
async function createTransaction(data: CreateTransactionInput) {
  const debitTotal = data.entries
    .filter(e => e.type === 'debit')
    .reduce((sum, e) => sum.plus(e.amount), new Decimal(0));
  const creditTotal = data.entries
    .filter(e => e.type === 'credit')
    .reduce((sum, e) => sum.plus(e.amount), new Decimal(0));

  if (!debitTotal.equals(creditTotal)) {
    throw new DoubleEntryError(
      `Imbalance: debit ${debitTotal} ≠ credit ${creditTotal}`
    );
  }

  return prisma.transaction.create({ data: { ...data } });
}
```

Rule F4: VAT Rates Are from Database, Never Hardcoded

```
// WRONG
const vatRate = country === 'RS' ? 20 : 17;

// CORRECT — get from VatRate table, allow configuration
const vatRate = await getVatRate(organizationId, country, itemType, transactionDate);
```

Rule F5: Exchange Rates Are Locked at Transaction Date

```
// WRONG
const rate = await getCurrentExchangeRate(fromCurrency, toCurrency);

// CORRECT
const rate = await getExchangeRateForDate(fromCurrency, toCurrency, transactionDate);
if (!rate) throw new Error(`No exchange rate for ${fromCurrency}/${toCurrency} on
${transactionDate}`);
```

---

3. API Design

RESTful Conventions

Action	Method	Path	Success Code
List	GET	`/api/v1/invoices`	200
Create	POST	`/api/v1/invoices`	201
Read	GET	`/api/v1/invoices/:id`	200
Update	PATCH	`/api/v1/invoices/:id`	200
Delete (soft)	DELETE	`/api/v1/invoices/:id`	200
Bulk action	POST	`/api/v1/invoices/bulk-send`	200

Request Validation with Zod

All request bodies and query parameters must be validated with Zod before use.

```
import { z } from 'zod';

const CreateInvoiceSchema = z.object({
  customerId: z.string().uuid(),
  invoiceDate: z.string().regex(/^\d{4}-\d{2}-\d{2}$/),
  dueDate: z.string().regex(/^\d{4}-\d{2}-\d{2}$/),
  currencyCode: z.enum(['RSD', 'EUR', 'BAM', 'HRK', 'USD']),
```

```
  items: z.array(z.object({
    description: z.string().min(1).max(500),
    quantity: z.number().positive(),
    unitPrice: z.string().regex(/^\d+(\.\d{1,4})?$/),  // Decimal string
    taxRate: z.number().min(0).max(100),
  })).min(1),
});

// In route handler
const parsed = CreateInvoiceSchema.safeParse(req.body);
if (!parsed.success) {
  return res.status(400).json({ error: 'Validation failed', details: parsed.error });
}
```

Error Response Format

```
{
  "error": "Validation failed",
  "code": "VALIDATION_ERROR",
  "details": [
    { "field": "customerId", "message": "Invalid UUID" }
  ]
}
```

Standard error codes: `VALIDATION_ERROR`, `NOT_FOUND`, `FORBIDDEN`, `UNAUTHORIZED`, `DOUBLE_ENTRY_ERROR`, `INSUFFICIENT_PERMISSIONS`

Pagination

All list endpoints must support pagination:

```
// Query: GET /api/v1/invoices?limit=20&cursor=<uuid>

// Response
{
  "data": [...],
  "pagination": {
    "cursor": "<next-cursor-uuid>",
    "hasMore": true,
    "total": 150
```

```
  }
}
```

---

4. Database (Prisma)

Query Rules

```
// ALWAYS include organizationId filter (multi-tenancy)
const invoices = await prisma.invoice.findMany({
  where: {
    organizationId: req.user.organizationId,  // Required
    deletedAt: null,
  },
});

// NEVER expose deleted records unless explicitly showing "trash"
// NEVER query without organizationId on any user-facing query

// Use select for response DTOs — never return full Prisma models
const invoice = await prisma.invoice.findUnique({
  where: { id, organizationId: req.user.organizationId },
  select: {
    id: true,
    invoiceNumber: true,
    totalAmount: true,
    status: true,
    // ... only what the endpoint needs
  },
});
```

Migration Rules

- NEVER edit existing migrations
- ALWAYS create new migrations for schema changes: `npx prisma migrate dev --name describe_change`
- Migrations must be backward-compatible (old code can still read data) unless explicitly planned otherwise

- NEVER drop columns in a migration — first deprecate, remove code references, then drop in a separate migration
- Always test migrations on staging before production

---

5. Authentication & Authorization

Every Route Needs Auth (No Exceptions)

```
// Middleware applied globally
app.use('/api/v1', authenticateJWT);

// Route-level RBAC
router.post('/invoices', requireRole(['owner', 'admin', 'accountant']), createInvoice);
router.delete('/invoices/:id', requireRole(['owner', 'admin']), deleteInvoice);
router.get('/invoices/:id', requireRole(['owner', 'admin', 'accountant', 'viewer']),
getInvoice);
```

RBAC Roles and Permissions

Role	Can Read	Can Create	Can Approve	Can Delete	Can Manage Users
viewer	Own org only	No	No	No	No
accountant	Own org only	Invoices, Expenses	No	No	No
admin	Own org only	All	Expenses	Soft delete	No
owner	Own org only	All	All	All	Yes

---

6. Code Organization

File Structure (Backend)

```
apps/api/src/
├── routes/
│   ├── auth.routes.ts          # Route definitions only
```

```
│   ├── invoices.routes.ts
│   └── invoices.routes.test.ts  # Integration tests co-located
├── services/
│   ├── invoice.service.ts       # Business logic
│   └── invoice.service.test.ts  # Unit tests co-located
├── utils/
│   ├── vat.ts                   # VAT calculation utilities
│   ├── vat.test.ts
│   ├── double-entry.ts
│   └── currency.ts
├── middleware/
│   ├── auth.middleware.ts
│   └── rbac.middleware.ts
└── lib/
    ├── prisma.ts                # Prisma client singleton
    └── errors.ts                # Custom error classes
```

Naming Conventions

Thing	Convention	Example
Files	kebab-case	`invoice-service.ts`
Functions	camelCase	`calculateInvoiceTotal`
Classes	PascalCase	`InvoiceService`
Constants	UPPER_SNAKE	`MAX_INVOICE_ITEMS = 100`
Types/Interfaces	PascalCase	`CreateInvoiceInput`
Database models	PascalCase (Prisma)	`Invoice`, `InvoiceItem`
API routes	kebab-case	`/api/v1/invoice-items`

---

7. Testing Standards

Write Tests in the Same PR as the Feature

No "I'll add tests later". Tests are part of the feature.

Required Tests for Financial Logic

```
// For EVERY financial utility function, include:
it('calculates correctly', () => { ... });
it('handles zero amounts', () => { ... });
it('handles decimal precision (NUMERIC(19,4))', () => {
  const result = calculateVAT(new Decimal('0.1000'), 20);
  expect(result.toString()).toBe('0.0200');  // Not 0.020000000000000004
});
it('throws on invalid input', () => { ... });
```

Test File Co-location

Tests live next to the file they test:

- `vat.ts` → `vat.test.ts` (same directory)
- `invoices.routes.ts` → `invoices.routes.test.ts` (same directory)

---

8. Linting & Formatting

Configured via ESLint + Prettier. Run automatically on every commit (Husky) and in CI.

```
# Fix all linting issues
pnpm run lint -- --fix

# Check formatting
pnpm run lint
```

Key ESLint rules:

- `no-floating-decimal` — numbers like `.5` must be written as `0.5`
- `@typescript-eslint/no-explicit-any` — warn (add comment to suppress)
- `no-console` — warn in production code (use logger instead)
- Import order enforced

---

9. Security

Never Commit Secrets

```
# .gitignore includes:
.env
.env.*
!.env.example
```

OWASP Top 10 Checklist for New Code

- ☐ SQL injection: using Prisma parameterized queries only (never string concatenation)
- ☐ XSS: user input never inserted into HTML without escaping (React escapes by default)
- ☐ Auth: every route has authentication check
- ☐ IDOR: every DB query filtered by `organizationId`
- ☐ Sensitive data: no passwords/tokens in logs
- ☐ Input validation: Zod on all request bodies

---

10. Version Control

Commit Message Format

```
type(scope): description (#issue-id)

type: feat | fix | refactor | test | docs | chore | perf | security
scope: api | web | db | infra | deps
```

Examples:

```
feat(api): add VAT calculation for Croatia 25% rate (#234)
fix(api): correct decimal precision in invoice total calculation (#456)
test(api): add unit tests for double-entry validation
refactor(web): extract invoice form to separate component
```

Branch Naming

```
feature/<issue-id>-<short-description>
fix/<issue-id>-<short-description>
hotfix/<short-description>
```

PR Requirements

- Title follows commit message format
- Description: What changed, why, any testing notes
- Links to issue: "Closes #XXX" or "Ref #XXX"
- All CI checks green
- No `.only` or `.skip` in test files
- Coverage did not decrease below thresholds

Related Documents

- Test Strategy
- Definition of Done
- Developer Onboarding Guide
- BILKO CLAUDE.md

Approval

Role	Name	Date	Signature
Author	Ops Architect	2026-02-23	
Reviewer	Tech Lead		
Approver	Alem Bašić		

Revision #3
Created 2026-02-24 23:11:27 UTC by John
Updated 2026-05-31 20:04:19 UTC by John