Architecture Decision Records (ADR)

Architecture Decision Records — Bilko

> **Project:** Bilko **Version:** 1.1 **Date:** 2026-02-24 **Author:** Petter Graff (ADR-001 to ADR-006), Security-Test-Writer Agent (ADR-007 to ADR-013) **Status:** Active

Document History

Version	Date	Author	Changes
0.1	2026-02-23	Petter Graff	Initial draft — ADR-001 to ADR-006
1.1	2026-02-24	Security-Test-Writer	Added ADR-007 to ADR-013 — auth, state, framework, hosting, ORM, validation, router

---

ADR-001 — Turborepo Monorepo with Separate Packages

> **ADR Number:** ADR-001 **Title:** Use Turborepo monorepo with separate npm packages for core and country plugins **Date:** 2026-02-23 **Author:** Petter Graff **Status:** Accepted

1. Context

1.1 Situation

Bilko is a multi-country accounting SaaS with: (1) a Next.js frontend, (2) an Express backend, (3) a country-agnostic accounting engine, and (4) three country-specific regulatory plugins. These components share TypeScript types and business logic but must be independently versioned and testable.

1.2 Forces & Constraints

Technical forces:

- TypeScript types (Prisma models, API response shapes) must be shared between frontend and backend without duplication
- Country plugins must be independently releasable (a VAT rate change in Serbia should not require redeploying Croatia logic)
- Build times must remain fast as the codebase grows

Business forces:

- MVP must launch quickly; tooling complexity must be low
- Team is small — cannot manage multiple separate repositories

1.3 Problem Statement

> **We need to decide:** How to organize the Bilko codebase so that frontend, backend, accounting engine, and country plugins share code but can be independently versioned and tested.

2. Decision

We will: Use Turborepo to manage a monorepo with the structure `apps/web`, `apps/api`, `packages/core`, `packages/database`, `packages/country-{rs|ba|hr}`, `packages/ui`.

Rationale: Turborepo provides incremental builds, shared TypeScript configuration, and workspace management without the overhead of publishing packages to npm. The plugin architecture is enforced at the package boundary — country plugins import `@bilko/core` but not each other.

3. Alternatives Considered

Option A: Turborepo Monorepo ? Selected

Pros:

- Fast incremental builds — only rebuilds changed packages
- Shared TypeScript types across all apps and packages
- Country plugins are independent packages — different release cadence per country regulation
- Single repo — one PR, one CI run, one git history

Cons:

- Turborepo learning curve for new developers
- All packages share the same git repo — sensitive to large PRs

Cost/Effort: Low — Turborepo setup is ~1 day

Option B: Separate Git Repositories (polyrepo)

Pros:

- Maximum isolation between packages
- Per-repo CI/CD pipelines

Cons:

- Type sharing requires publishing to npm registry — adds publish/version management overhead
- Cross-repo refactoring is painful
- Slow for a small team

Why not chosen: Type sharing overhead is too high for MVP pace.

Option C: Single Package (everything in one `apps/api` and `apps/web`)

Pros:

- Simplest setup — no workspace config

Cons:

- Cannot independently version or test the accounting engine
- Country tax logic is tangled with API routes
- Cannot reuse accounting engine in future products

Why not chosen: Violates separation of concerns; makes future product reuse impossible.

4. Consequences

4.1 Positive Consequences

- `@bilko/core` can be unit tested without a database
- Country plugins can update VAT rates in a hotfix without touching API or frontend code
- Prisma types from `@bilko/database` are shared — no duplicated interfaces

4.2 Negative Consequences

- Turborepo `turbo.json` pipeline config must be maintained as packages grow — *Mitigation: document pipeline in `PIPELINE.md`*

4.3 Technical Debt Created

- `packages/ui` is an empty scaffold at MVP — not yet used — *Plan: populate with shared shadcn components in v1.1*

ADR-002 — NUMERIC(19,4) and Decimal.js for All Monetary

Values

> **ADR Number:** ADR-002 **Title:** Use PostgreSQL NUMERIC(19,4) and Decimal.js for all monetary amounts — never IEEE 754 float **Date:** 2026-02-23 **Author:** Petter Graff **Status:** Accepted

1. Context

1.1 Situation

Bilko is an accounting system. Every stored value represents money. JavaScript's native `number` type uses IEEE 754 double-precision floating point, which cannot represent 0.1 exactly — `0.1 + 0.2 === 0.30000000000000004`. This is catastrophic for financial software where rounding errors cause balance mismatches that fail audits.

1.2 Forces & Constraints

Technical forces:

- Accounting software requires exact decimal arithmetic — not approximations
- PostgreSQL's `NUMERIC` type supports arbitrary precision; `FLOAT` does not
- Prisma maps `NUMERIC` to its `Decimal` type (backed by `decimal.js`)

Regulatory:

- Serbian, Croatian, and BiH tax law require monetary precision to 4 decimal places for VAT calculations

1.3 Problem Statement

> **We need to decide:** What data type to use for all monetary values in the database and application layer.

2. Decision

We will: Store all monetary values as `NUMERIC(19,4)` in PostgreSQL and use `Decimal.js` (via Prisma's `Decimal` type) in all application code. JavaScript `number` is prohibited for any value representing money.

Rationale: `NUMERIC(19,4)` can represent amounts up to 999,999,999,999,999.9999 (sufficient for SMB range). `Decimal.js` provides arbitrary-precision arithmetic. The combination eliminates all floating-point rounding errors.

3. Alternatives Considered

Option A: NUMERIC(19,4) + Decimal.js ? Selected

Pros:

- Exact decimal arithmetic — 0.1 + 0.2 = 0.3000 exactly
- Range: up to ~1 quadrillion — sufficient for any SMB
- Prisma's `Decimal` type integrates seamlessly

Cons:

- Developers must remember to never use `number` for money
- `Decimal.js` operations are slightly more verbose: `new Decimal(a).plus(b)` vs `a + b`

Option B: Store as integers (minor units — e.g., paras/cents)

Pros:

- Integer arithmetic is exact with no library needed
- Common approach in payment systems (Stripe stores cents)

Cons:

- Accounting requires 4 decimal places — minor units would need to be in 1/10000 of a currency unit (unusual)
- Prisma schema, API responses, and display logic all require conversion
- Confusing for accountants reviewing data directly

Why not chosen: 4-decimal-place requirement makes minor-unit storage awkward.

Option C: JavaScript `number` / PostgreSQL `DOUBLE PRECISION`

Pros:

- No library needed
- Simple arithmetic operators

Cons:

- **Fatal flaw:** `0.1 + 0.2 !== 0.3` — causes balance sheet mismatches in any real scenario
- Cannot pass an audit with floating-point rounding errors in ledger balances

Why not chosen: Fundamentally incompatible with financial software requirements.

4. Consequences

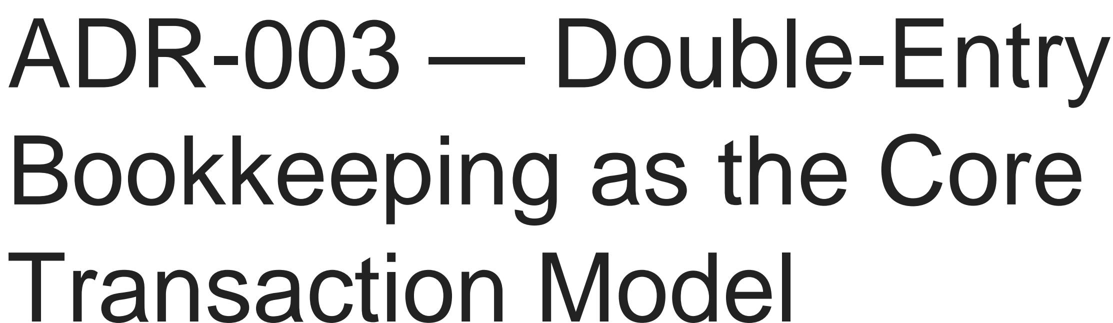
4.1 Positive Consequences

- Trial balance always balances to exactly zero — `totalDebits.equals(totalCredits)`
- VAT calculations are exact — no rounding surprises for Serbian PDV declarations
- API responses return monetary values as strings (e.g., `"120000.0000"`) — clients parse as `Decimal`

4.2 Negative Consequences

- All API clients must handle monetary values as strings, not numbers — *Mitigation: document in API spec; TypeScript types enforce `string` for monetary fields*

ADR-003 — Double-Entry Bookkeeping as the Core Transaction Model



> **ADR Number:** ADR-003 **Title:** Use double-entry bookkeeping as the sole transaction model — every financial event creates one `Transaction` with debit and credit **Date:** 2026-02-23 **Author:** Petter Graff **Status:** Accepted

1. Context

1.1 Situation

Bilko must generate legally valid accounting reports: profit & loss, balance sheet, trial balance, VAT return. These reports require a general ledger where every financial event is recorded as a balanced journal entry. Many SaaS products use simpler "append events to a list" approaches that work for dashboards but fail for formal accounting.

1.2 Forces & Constraints

Regulatory:

- Serbian, BiH, and Croatian accounting law require double-entry bookkeeping for legal entities
- VAT filings require a precise split between output VAT (from revenue) and input VAT (from expenses)

Technical forces:

- Double-entry enables the `Trial Balance` to always balance — a fundamental correctness check
- It enables any accounting report to be derived from the same `Transaction` table

1.3 Problem Statement

> **We need to decide:** How to store financial events — simple event list vs. double-entry journal.

2. Decision

We will: Use double-entry bookkeeping as the sole transaction model. Every financial event creates exactly one `Transaction` record with `debitAccountId`, `creditAccountId`, and `amount`. The `validateDoubleEntry()` function in `@bilko/core` is called before every Prisma transaction to enforce balance.

3. Alternatives Considered

Option A: Double-Entry Bookkeeping ? Selected

Pros:

- Legally correct for all three target countries
- Enables Trial Balance, Balance Sheet, P&L from one table
- `isBalanced` check provides immediate data integrity verification

Cons:

- Higher implementation complexity — developers must understand debit/credit semantics
- Every status change requires finding correct GL accounts

Option B: Simple Ledger (credit/debit as +/- amounts, single account per row)

Pros:

- Simpler to implement and understand
- Easier to query (no join to debit/credit accounts)

Cons:

- Cannot produce a legally valid trial balance
- VAT report requires cross-referencing invoices and expenses separately — cannot derive from ledger
- Not compliant with Pravilnik (Serbia), RRiF (Croatia), or FBiH Pravilnik (BiH)

Why not chosen: Regulatory non-compliance disqualifies this option.

Option C: No ledger — compute reports directly from invoices/expenses tables

Pros:

- Simplest: no separate transaction table
- Fast to build for MVP

Cons:

- Cannot produce a balance sheet (assets, liabilities, equity) without a ledger
- Cannot handle manual journal entries (adjustments, corrections)
- Cannot reconcile bank transactions against GL

Why not chosen: Insufficient for formal accounting; blocks future chartered accountant use.

4. Consequences

4.1 Positive Consequences

- All accounting reports (P&L, balance sheet, cash flow, trial balance) derive from the same `Transaction` table
- `locked = true` on transactions creates an immutable audit trail
- `reconciled = true` flag enables bank reconciliation

4.2 Technical Debt Created

- `validateDoubleEntry()` is called in application code, not enforced by DB constraint — `debitAmount = creditAmount` relies on application logic — *Mitigation: unit test every code path that creates transactions*

ADR-004 — Lock Exchange Rates at Transaction Date

> **ADR Number:** ADR-004 **Title:** Lock exchange rates at transaction date — never recalculate historical amounts **Date:** 2026-02-23 **Author:** Petter Graff **Status:** Accepted

1. Context

1.1 Situation

Bilko supports multi-currency invoicing (EUR, RSD, BAM, HRK, USD). When an invoice is created in a foreign currency, it must be converted to the organization's base currency for the general ledger. Exchange rates change daily. The question is: should historical amounts be recalculated when rates change, or locked at transaction date?

1.2 Forces & Constraints

Regulatory:

- All three accounting systems (Serbian, BiH, Croatian) require that historical financial records show the exchange rate in effect at the transaction date — recalculation is not permitted for recognized transactions

Technical forces:

- Recalculating historical amounts on every rate update would invalidate the trial balance and all prior period reports

1.3 Problem Statement

> **We need to decide:** Whether to lock exchange rates at transaction date or recalculate dynamically.

2. Decision

We will: Lock the exchange rate at `invoiceDate` (or `expenseDate`) and store it permanently in the `exchangeRate` field. `baseAmount = totalAmount × exchangeRate` is computed once and stored. Neither field is ever updated after creation.

3. Alternatives Considered

Option A: Lock at Transaction Date ? Selected

Pros:

- Legally compliant — historical records reflect the rate in effect at the time

- `baseAmount` is stable — prior period reports never change
- No complex recalculation logic

Cons:

- If no rate exists for the exact date, fallback to nearest date — risk of slight inaccuracy
- Exchange rate population is a prerequisite for accuracy

Option B: Recalculate on every report

Pros:

- Always uses the current rate
- Simpler — no need to store historical rates

Cons:

- Illegal under Balkan accounting standards — invoices must show the rate at issue date
- Prior period P&L reports would change every time rates update
- Audit trail becomes meaningless if historical amounts drift

Why not chosen: Regulatory non-compliance.

4. Consequences

4.1 Positive Consequences

- Invoice `exchangeRate` and `baseAmount` are immutable after creation
- The `ExchangeRate` table with `effectiveDate` supports complete historical rate lookup
- `lockExchangeRate()` in `@bilko/core` centralizes this logic

4.2 Negative Consequences

- Exchange rate population (ECB daily cron) must be running before multi-currency invoices are created — *Mitigation: warn in UI if no rate found for selected date*

ADR-005 — Organization-Scoped Multi-Tenancy via

Middleware

> **ADR Number:** ADR-005 **Title:** Enforce multi-tenancy at the API middleware layer (organizationScope) — not via PostgreSQL RLS **Date:** 2026-02-23 **Author:** Petter Graff **Status:** Accepted

1. Context

1.1 Situation

Bilko is a multi-tenant SaaS. Every database record (invoices, expenses, transactions, etc.) belongs to one organization. Cross-organization data access would be a critical security breach. There are two primary approaches: enforce isolation in the database (RLS) or in the application (middleware).

1.2 Forces & Constraints

Technical forces:

- Team is more proficient in TypeScript/Node.js than PostgreSQL RLS configuration
- Prisma ORM does not have first-class RLS support — it would require raw SQL policies
- Application-layer enforcement is testable with unit tests

Business forces:

- MVP timeline is tight — RLS adds PostgreSQL configuration complexity

1.3 Problem Statement

> **We need to decide:** Where to enforce that users can only access data from their own organization.

2. Decision

We will: Enforce organization scoping via the `organizationScope` Express middleware (`apps/api/src/middleware/org-scope.ts`), which attaches `req.user.organizationId` from the JWT. All service methods receive `organizationId` as first parameter and all Prisma queries include `where: { organizationId }`.

3. Alternatives Considered

Option A: Application-layer middleware ? Selected

Pros:

- Enforceable in unit tests — mock `req.user.organizationId`
- Developers understand it — familiar TypeScript patterns
- Works with Prisma ORM without raw SQL

Cons:

- Defense depends on application code correctness — a missed `where: { organizationId }` is a bug
- No DB-level enforcement as a second layer

Option B: PostgreSQL Row-Level Security (RLS)

Pros:

- Enforcement at database level — even direct DB queries are isolated
- Defense-in-depth — second layer below application

Cons:

- Requires setting `app.current_tenant_id` on each connection via Prisma `$executeRaw`
- Prisma does not natively support connection-level session variables
- Complex to test and debug — policy errors show as unexpected empty results

Why not chosen: Prisma integration complexity and team expertise gap. Can be added post-MVP.

4. Consequences

4.1 Positive Consequences

- Enforcement is testable — every service method test verifies `organizationId` filter is applied

4.2 Negative Consequences

- Direct DB access (e.g., migrations, admin scripts) bypasses enforcement — *Mitigation: document that all data access must go through API*

4.3 Technical Debt Created

- RLS would provide a stronger defense-in-depth guarantee — *Plan: evaluate adding PostgreSQL RLS as a secondary enforcement layer post-MVP*

ADR-006 — Country Plugin Architecture as Separate npm Packages

> **ADR Number:** ADR-006 **Title:** Implement country-specific accounting rules as separate Turborepo packages with a shared interface **Date:** 2026-02-23 **Author:** Petter Graff **Status:** Accepted

1. Context

1.1 Situation

Bilko targets three countries with different VAT rates, tax thresholds, e-invoice platforms, chart of accounts templates, and filing deadlines. These rules change independently (e.g., Serbia changed e-invoice rules in 2023; Croatia mandated eRačun from Jan 2026). Rules must be extensible to future markets (Slovenia, North Macedonia) without modifying core accounting logic.

1.2 Problem Statement

> **We need to decide:** How to organize country-specific accounting rules to allow independent versioning and easy extensibility.

2. Decision

We will: Implement each country as a separate Turborepo package (`@bilko/country-rs`, `@bilko/country-ba`, `@bilko/country-hr`) with a standard module structure: `tax/`, `chart/`, `fiscal/`, `filing/`, `locale/`, `index.ts`. The API selects the plugin at runtime based on `org.country`.

3. Alternatives Considered

Option A: Separate packages per country ? Selected

Pros:

- A Serbia VAT rate change does not touch Croatia or BiH code
- Each plugin can be unit tested independently
- New countries (e.g., Slovenia) are added as a new package without touching existing code
- Regulatory changes trigger a focused PR in one package

Cons:

- Shared interface must be maintained — breaking change to `CountryPlugin` interface affects all plugins
- Slightly more files to create per new country

Option B: Single `@bilko/countries` package with all countries in subdirectories

Pros:

- Simpler dependency management

Cons:

- A Serbian VAT rate PR touches the same package as Croatia/BiH — increases risk of cross-country bugs
- Cannot version countries independently

Why not chosen: Independent versioning is critical given different regulatory cadences per country.

Option C: Country rules stored in database (configurable per org)

Pros:

- Can update VAT rates without a deployment
- Admin UI for configuration

Cons:

- VAT calculation logic (not just rates) varies by country — cannot be reduced to a database record
- E-invoice XML format generation (UBL 2.1) must be code, not config
- Much higher complexity at MVP stage

Why not chosen: Regulatory logic (e-invoice XML, fiscal year rules) cannot be stored as configuration.

4. Consequences

4.1 Positive Consequences

- Serbia launched first — `@bilko/country-rs` is the most complete plugin; BiH and Croatia can lag
- `@bilko/core` remains country-agnostic — reusable in future non-Bilko products

4.2 Negative Consequences

- Adding a new country requires creating a new package, wiring it in the API country-selector — *Mitigation: document standard plugin creation guide in `CONTRIBUTING.md`*

ADR-007 — JWT for Authentication

> **ADR Number:** ADR-007 **Title:** Use JWT (RS256) with short-lived access tokens and httpOnly refresh tokens — no server-side sessions **Date:** 2026-02-24 **Author:** Security-Test-Writer Agent **Status:** Accepted

1. Context

1.1 Situation

Bilko's API must authenticate users on every request. The system must be stateless for horizontal scaling. Tokens must be secure against XSS and CSRF attacks. The mobile PWA requires a token-based approach (cookies work; sessions require sticky sessions).

1.2 Forces & Constraints

Technical forces:

- API is consumed by Next.js frontend (same-domain in production) and potentially mobile PWA in future
- Access tokens must be short-lived to limit exposure if intercepted
- Refresh tokens must survive browser tab reloads without re-authentication

Security forces:

- Access tokens in `localStorage` are vulnerable to XSS
- Session cookies without SameSite are vulnerable to CSRF

Business forces:

- Must avoid per-user costs of managed auth services at MVP scale
- Must work without Redis (no session store in MVP — see ADR-010)

1.3 Problem Statement

> **We need to decide:** How to authenticate API requests — JWT, session-based, or delegated OAuth.

2. Decision

We will: Use RS256-signed JWT access tokens (15 min TTL, stored in memory by the client) and RS256-signed refresh tokens (7 days, stored as `httpOnly; SameSite=Strict` cookie). Access tokens contain `sub`, `email`, `organizationId`, `role`, `iat`, `exp`. No server-side session store is required.

Implementation: `apps/api/src/utils/jwt.ts` (sign/verify), `apps/api/src/middleware/auth.ts` (authGuard middleware).

3. Alternatives Considered

Option A: JWT (RS256, access + refresh) ? Selected

Pros:

- Stateless — no Redis or DB session table required
- RS256 asymmetric signing — public key can be shared for future microservices validation
- 15-min access token limits breach window
- httpOnly refresh cookie prevents XSS token theft
- No per-user cost

Cons:

- Cannot invalidate individual access tokens before expiry (JWT is stateless)
- Refresh token rotation requires careful implementation to prevent replay

Option B: Session-based authentication (server-side sessions)

Pros:

- Instant revocation — delete session from store
- No token expiry concerns on the client

Cons:

- Requires server-side session store (Redis) — increases infrastructure complexity for MVP
- Not naturally stateless — complicates horizontal scaling without sticky sessions
- Session cookies are CSRF-vulnerable without additional mitigation

Why not chosen: Requires Redis which is not in the MVP stack (ADR-010).

Option C: Delegated OAuth (Auth0, Clerk, Supabase Auth)

Pros:

- Managed security, automatic token refresh, MFA support
- Reduces auth implementation burden

Cons:

- Auth0/Clerk: $23+/month for 1,000+ users — significant cost at MVP scale
- External dependency — outage on Auth0 = Bilko login down
- Couples auth to third-party vendor; migration is painful

Why not chosen: Cost and external dependency at MVP stage.

4. Consequences

4.1 Positive Consequences

- API is fully stateless — scales horizontally without sticky sessions
- `organizationId` and `role` in JWT payload mean 0 DB queries for authorization checks
- Future microservices can verify tokens independently using the public key

4.2 Negative Consequences

- Access token cannot be revoked before 15 min expiry — *Mitigation: 15 min window is acceptable; force re-login via refresh token revocation if compromise suspected*

4.3 Technical Debt Created

- 2FA TOTP field exists on User model but not yet wired into login flow — *Plan: wire 2FA check in auth middleware in v1.1*

ADR-008 — Zustand for Frontend State Management

> **ADR Number:** ADR-008 **Title:** Use Zustand for minimal global frontend state — React hooks for local component state **Date:** 2026-02-24 **Author:** Security-Test-Writer Agent **Status:** Accepted

1. Context

1.1 Situation

The Next.js frontend (`apps/web`) needs to share state across components: authenticated user info, current organization, UI preferences (sidebar open/closed, theme). Most application data is server-fetched per page — only a small slice of global state is needed in the client.

1.2 Forces & Constraints

Technical forces:

- Next.js App Router encourages React Server Components (RSC) for data fetching — most data doesn't need global client-side state
- Client Components still need access to auth state (user, org) without prop-drilling
- State management solution must not add heavy bundle to RSC-first architecture

Business forces:

- MVP pace — no time for Redux boilerplate

1.3 Problem Statement

> **We need to decide:** What library to use for global client-side state management in the Next.js frontend.

2. Decision

We will: Use Zustand 4.5 for a minimal global store containing `{ user, organization, accessToken, setUser, clearAuth }`. All per-page data is fetched server-side in RSCs or client-side with React hooks. Zustand is NOT used as a substitute for server-fetched data.

3. Alternatives Considered

Option A: Zustand ? Selected

Pros:

- 1KB bundle — negligible impact on LCP
- Hooks-based API — consistent with React patterns
- Zero boilerplate — `create((set) => ({ ... }))` is all that's needed
- No Provider wrapping required (unlike Context API)
- Excellent TypeScript inference

Cons:

- Less opinionated — developers must define store structure themselves
- No built-in devtools (Redux has excellent browser devtools)

Option B: Redux Toolkit (RTK)

Pros:

- Battle-tested at large scale
- Excellent browser devtools (time-travel debugging)
- Built-in RTK Query for server state caching

Cons:

- ~50KB bundle (vs 1KB for Zustand)
- Slice/action/reducer/dispatch boilerplate even with RTK
- RTK Query would duplicate Next.js RSC data fetching capabilities

Why not chosen: Bundle overhead and boilerplate disproportionate to MVP state needs.

Option C: React Context API

Pros:

- Built-in — no dependency
- Familiar to all React developers

Cons:

- Every `useContext` consumer re-renders on any state change — causes performance issues with auth state updates
- No atomic selector pattern — must split contexts manually to avoid cascading re-renders

Why not chosen: Performance characteristics unsuitable for auth state shared across many components.

Option D: Jotai / Recoil

Pros:

- Atomic state model — precise subscriptions, no unnecessary re-renders
- Jotai is React-focused, small bundle

Cons:

- Less community adoption than Zustand
- Atomic model adds cognitive overhead for simple use case (one auth store)

Why not chosen: Zustand is simpler and better supported for our use case.

4. Consequences

4.1 Positive Consequences

- Zero boilerplate for auth state management
- No Provider nesting required — clean component tree
- Tiny bundle contribution

4.2 Negative Consequences

- Custom devtools setup required for debugging — *Mitigation: Zustand supports Redux DevTools extension via `devtools` middleware*

ADR-009 — Express for Backend API Framework

> **ADR Number:** ADR-009 **Title:** Use Express.js with TypeScript as the backend API framework **Date:** 2026-02-24 **Author:** Security-Test-Writer Agent **Status:** Accepted

1. Context

1.1 Situation

Bilko's backend API (`apps/api`) handles all accounting operations. The framework must support TypeScript, middleware composition (auth, validation, rate limiting), and have a mature ecosystem for the integrations needed (Prisma, Zod, Helmet, express-rate-limit).

1.2 Forces & Constraints

Technical forces:

- Framework must compose cleanly with Prisma, Zod validators, Helmet, and custom middleware
- Middleware order matters for security (Helmet must run before routes)
- TypeScript support must be first-class

Business forces:

- Team familiarity reduces ramp-up time
- Must not over-engineer at MVP stage

1.3 Problem Statement

> **We need to decide:** Which Node.js HTTP framework to use for the Express API.

2. Decision

We will: Use Express 4.x with TypeScript. Middleware stack: `helmet → cors → json → rate-limit → auth → validate → handler → error-handler`. Route modules in `apps/api/src/routes/`, service layer in `apps/api/src/services/`.

3. Alternatives Considered

Option A: Express.js ? Selected

Pros:

- Battle-tested (10+ years, billions of installs)
- Largest ecosystem — every library has an Express adapter/example
- Minimal and unopinionated — no forced patterns
- Team familiarity — zero ramp-up time
- Excellent TypeScript support via `@types/express`

Cons:

- 2-3x slower throughput than Fastify in benchmarks (not meaningful at MVP scale <500 concurrent users)
- No built-in input validation — requires Zod middleware
- Verbose error handling without an error middleware

Option B: Fastify

Pros:

- 2-3x faster than Express in benchmarks
- Built-in JSON schema validation
- Strong TypeScript support

Cons:

- Plugin system is more complex than Express middleware
- Smaller ecosystem — some libraries require shimming for Fastify
- Less team familiarity — ramp-up cost

Why not chosen: Performance advantage irrelevant at MVP scale; ecosystem gaps outweigh speed benefit.

Option C: NestJS

Pros:

- Full framework with DI, modules, decorators
- Built-in Swagger generation
- Strong conventions reduce decision fatigue

Cons:

- Angular-style architecture with heavy boilerplate (modules, providers, controllers, decorators)
- 10-15x more files than equivalent Express app
- Forces specific patterns that conflict with our functional service layer approach
- Adds significant complexity for a small team

Why not chosen: Overkill for MVP. Complexity outweighs conventions benefit for a small team.

Option D: Hono

Pros:

- Extremely fast, edge-native, very small bundle
- Excellent TypeScript support (router types)

Cons:

- Very new (2023) — limited production track record for financial applications
- Smaller ecosystem — fewer middleware examples

Why not chosen: Unproven at scale for financial SaaS; ecosystem too immature.

4. Consequences

4.1 Positive Consequences

- Predictable middleware execution order — critical for security (Helmet before routes)
- Massive example library for every integration (Prisma, Zod, JWT, multer, etc.)
- Service layer design is framework-agnostic — can migrate to Fastify/Hono in v2 without rewriting services

4.2 Technical Debt Created

- If Bilko scales to 10K+ concurrent connections, benchmarks should be run to evaluate Fastify migration — *Plan: evaluate at post-10K scale*

ADR-010 — Vercel + Railway for Hosting

> **ADR Number:** ADR-010 **Title:** Use Vercel for frontend hosting and Railway for backend API + PostgreSQL — no self-managed infrastructure at MVP **Date:** 2026-02-24 **Author:** Security-Test-Writer Agent **Status:** Accepted

1. Context

1.1 Situation

Bilko needs hosting for: (1) Next.js 15 frontend, (2) Express API, (3) PostgreSQL database, (4) file storage (Cloudflare R2). The team is small; infrastructure management overhead must be minimal. GDPR compliance requires EU data residency.

1.2 Forces & Constraints

Business forces:

- MVP budget: <€50/month for hosting
- No dedicated DevOps resource — infra must be managed with minimal effort
- EU data residency for GDPR compliance (Serbia, BiH, Croatia users)

Technical forces:

- Next.js App Router is optimized for Vercel deployment (edge functions, ISR)
- PostgreSQL must be accessible from the API without VPC configuration complexity

1.3 Problem Statement

> **We need to decide:** Where to host the Bilko MVP — managed platforms vs. cloud VMs vs. self-hosted.

2. Decision

We will: Deploy the Next.js frontend to **Vercel** (Hobby → Pro tier) and the Express API + PostgreSQL to **Railway** (EU Frankfurt region). Cloudflare R2 for file storage (PDF invoices, receipts).

Cost breakdown:

Component	Service	Cost
Frontend	Vercel	€0 (Hobby) → €20 (Pro)
API + PostgreSQL	Railway Starter	€5–20/mo
File storage	Cloudflare R2	~€1/mo
Total MVP		**~€21/mo**

3. Alternatives Considered

Option A: Vercel + Railway ? Selected

Pros:

- €21/month — minimal cost for MVP
- Vercel: zero-config Next.js deployment, preview URLs per PR, global CDN, edge network
- Railway: PostgreSQL included, EU Frankfurt region (GDPR), git-push deploys, no Dockerfile needed
- No SSL configuration, no Nginx management, no PM2 setup

Cons:

- Railway has vendor lock-in risk (smaller company than AWS/GCP)
- Performance headroom limited to Railway's plan tiers

Option B: AWS (EC2 + RDS + CloudFront + Route 53)

Pros:

- Unmatched scalability and control
- eu-central-1 (Frankfurt) for GDPR compliance
- RDS Multi-AZ for HA

Cons:

- €80–150/month minimum for equivalent services (EC2 t3.small + RDS t3.micro + CloudFront)
- Requires VPC, security groups, IAM roles, Route 53 — significant DevOps overhead
- No preview deployments out of the box

Why not chosen: 4-7x more expensive at MVP scale; ops overhead unacceptable for small team.

Option C: GCP (Cloud Run + Cloud SQL + Firebase Hosting)

Pros:

- Competitive pricing, serverless Cloud Run
- Firebase Hosting for Next.js

Cons:

- Similar ops complexity to AWS
- Less team familiarity
- Firebase Hosting has limited Next.js App Router support

Why not chosen: Team unfamiliarity; less Next.js optimization than Vercel.

Option D: Self-hosted (Hetzner VPS / dedicated)

Pros:

- Cheapest at scale (€6/month for CX21)
- Full control

Cons:

- Full ops responsibility — nginx, SSL certs, PM2, backups, security updates, monitoring
- No preview deployments
- Single point of failure without manual HA setup

Why not chosen: Ops burden unacceptable for a 2-person team at MVP stage.

4. Consequences

4.1 Positive Consequences

- Zero infrastructure management time — team focuses on product
- Preview URLs per PR via Vercel — enables QA without staging environments
- PostgreSQL managed by Railway — automatic backups, connection pooling

4.2 Negative Consequences

- Railway limits: 2GB RAM, shared CPU on Starter — may require upgrade at 500+ concurrent users — *Mitigation: upgrade to Railway Pro (€20/mo) at scale*
- If Railway service is disrupted, requires migration to AWS or Fly.io — *Mitigation: keep Terraform modules ready for AWS deployment (in `infrastructure/terraform/`)*

4.3 Scaling Path

- **MVP (<500 users):** Vercel Hobby + Railway Starter (€21/mo)
- **Growth (500–2,000 users):** Vercel Pro + Railway Pro (€40–50/mo); add Redis for session caching if needed
- **Scale (2,000+ users):** Migrate API to AWS ECS/Fargate, RDS Multi-AZ; keep Vercel for frontend

---

ADR-011 — Prisma as the ORM

> **ADR Number:** ADR-011 **Title:** Use Prisma 5.x as the ORM with schema-as-code and generated TypeScript client **Date:** 2026-02-24 **Author:** Security-Test-Writer Agent **Status:** Accepted

1. Context

1.1 Situation

Bilko needs type-safe database access to PostgreSQL with: NUMERIC(19,4) for monetary values, UUID primary keys, Prisma-managed migrations, and Prisma middleware for the `LoggedAction` audit

trail.

1.2 Forces & Constraints

Technical forces:

- All monetary fields must be `Decimal` (mapped to `NUMERIC(19,4)`) — ORM must support this natively
- Migrations must be version-controlled and safe to run via CI/CD (`prisma migrate deploy`)
- Prisma Client must generate TypeScript types from schema — no manual type maintenance

1.3 Problem Statement

> **We need to decide:** Which ORM or database access library to use for Bilko's PostgreSQL interactions.

2. Decision

We will: Use Prisma 5.x (`packages/database/prisma/schema.prisma`). Prisma generates the TypeScript client (`@bilko/database`), manages migrations, and maps `Decimal` fields to `NUMERIC(19,4)`.

3. Alternatives Considered

Option A: Prisma ? Selected

Pros:

- Declarative schema — `schema.prisma` is the single source of truth
- Auto-generated TypeScript client with exact types from schema
- `@db.Decimal` maps directly to `NUMERIC(19,4)` — no manual conversion
- `prisma.$transaction()` for atomic operations (essential for double-entry)
- Prisma middleware enables `LoggedAction` audit trail implementation
- First-class VS Code extension with IntelliSense
- `prisma migrate deploy` is CI/CD-safe (no interactive prompts)

Cons:

- N+1 query risk with nested relations (must use `include: { ... }`)
- Raw SQL sometimes needed for complex financial aggregation queries
- Prisma's `Decimal` type requires care when passing to `@bilko/core` (which uses `decimal.js`)

Option B: Drizzle ORM

Pros:

- Excellent TypeScript inference (schema defined in TypeScript, not a DSL)
- Faster query performance than Prisma in benchmarks
- Growing ecosystem with strong community

Cons:

- At time of decision (early 2026), Drizzle's migration system was less mature than Prisma Migrate
- Decimal type support required additional configuration
- Fewer examples for financial/accounting use cases

Why not chosen: Less mature migration tooling at time of decision; Prisma's `@db.Decimal` is a better fit for NUMERIC(19,4).

Option C: TypeORM

Pros:

- Very mature — exists since 2016
- Supports both decorator-based and data-mapper patterns

Cons:

- Decorator-based entities require `experimentalDecorators` — adds TypeScript config complexity
- TypeScript inference is weaker than Prisma — more `any` in practice
- Decimal support requires `@Column({ type: 'decimal', precision: 19, scale: 4 })` on every monetary column

Why not chosen: Inferior TypeScript DX compared to Prisma; decorator-heavy pattern conflicts with functional service layer.

Option D: Knex.js (query builder)

Pros:

- Lightweight, full SQL control

- Works with any DB driver

Cons:

- Not an ORM — no type generation from schema
- Requires manually maintaining TypeScript interfaces for every table
- No migration tooling comparable to Prisma Migrate

Why not chosen: No type generation means maintaining types manually — unacceptable for a 15-model schema.

4. Consequences

4.1 Positive Consequences

- Schema changes are captured in versioned migration files (`packages/database/prisma/migrations/`)
- `npx prisma generate` regenerates the client after any schema change
- `prisma.$transaction()` makes double-entry atomic — both GL entries commit or neither does

4.2 Negative Consequences

- Complex aggregation queries (VAT report, trial balance) require raw SQL via `prisma.$queryRaw` — *Mitigation: isolate raw queries in `ReportService`, document each query's purpose*

ADR-012 — Zod for Request Validation

> **ADR Number:** ADR-012 **Title:** Use Zod 3.x for all API request validation with full TypeScript type inference **Date:** 2026-02-24 **Author:** Security-Test-Writer Agent **Status:** Accepted

1. Context

1.1 Situation

All user input entering the Bilko API must be validated before processing. Validation must: (1) reject invalid data with field-level error messages, (2) provide TypeScript types from the schema without duplication, (3) be composable across endpoints (reuse sub-schemas).

1.2 Forces & Constraints

Technical forces:

- TypeScript types must match runtime validation — manual type + validator duplication causes drift bugs
- Financial fields (amounts, dates, currency codes) require strict validation patterns
- Validation schemas are used as the source of truth for request body types in route handlers

Security forces:

- All user input must be validated before any DB query or business logic (OWASP A03 Injection prevention)

1.3 Problem Statement

> **We need to decide:** Which validation library to use for API request body/query validation.

2. Decision

We will: Use Zod 3.x for all request validation. Zod schemas are defined per endpoint and composed from shared sub-schemas (e.g., `monetaryAmountSchema`, `isoDateSchema`). The `validate(schema)` middleware in `apps/api/src/middleware/validate.ts` calls `schema.parse(req.body)` and passes validated, typed data to route handlers.

```
// Example: createInvoiceSchema
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.string().length(3),
  items: z.array(z.object({
    description: z.string().min(1).max(500),
    quantity: z.number().positive(),
    unitPrice: z.number().min(0),
    taxRate: z.number().min(0).max(100),
    accountId: z.string().uuid().optional(),
  })).min(1),
});
type CreateInvoiceRequest = z.infer<typeof createInvoiceSchema>;
```

3. Alternatives Considered

Option A: Zod ? Selected

Pros:

- TypeScript-first — `z.infer<typeof schema>` generates the type automatically
- Single source of truth — schema IS the type
- Composable — `z.object().merge()`, `z.discriminatedUnion()`, partial schemas
- Tree-shakeable — only imports what's used
- `.parse()` throws `ZodError` with field-level messages — maps directly to `422` responses
- Excellent ecosystem integration (tRPC, react-hook-form, etc. — useful in v2)

Cons:

- Slightly verbose for complex unions
- Bundle size (~13KB) — acceptable for server-side

Option B: Yup

Pros:

- Mature — exists since 2016
- Async validation support

Cons:

- TypeScript support is an afterthought — inference is weaker than Zod
- Schema and type must be maintained separately in practice
- More verbose `.shape()` API

Why not chosen: Inferior TypeScript inference — requires manual type maintenance.

Option C: Joi

Pros:

- Very mature — Hapi ecosystem
- Rich validation methods

Cons:

- JavaScript library — TypeScript types are maintained separately (`@hapi/joi` types have gaps)
- No built-in TypeScript inference — requires manual `interface` definitions
- Verbose for TypeScript projects

Why not chosen: Not TypeScript-native; dual maintenance of types and schemas.

Option D: class-validator + class-transformer

Pros:

- Decorator-based — familiar to NestJS/Java developers
- Works well with class instances

Cons:

- Requires `experimentalDecorators` TypeScript config
- Only works with class instances — requires `plainToClass()` transformation before validation
- Incompatible with plain object service layer design
- Tightly coupled to NestJS patterns we are not using

Why not chosen: Requires class instance pattern — incompatible with functional service layer.

4. Consequences

4.1 Positive Consequences

- Zero type drift between validation schema and TypeScript types
- `422` responses automatically include field-level error messages from Zod
- Shared sub-schemas enforce consistent validation across endpoints (e.g., all UUID fields validated the same way)

4.2 Negative Consequences

- Zod errors must be mapped to Bilko's standard error format in `errorHandler` middleware — *Mitigation: already implemented in* `apps/api/src/middleware/error-handler.ts`

ADR-013 — Next.js App Router

> **ADR Number:** ADR-013 **Title:** Use Next.js 15 App Router with React Server Components — not Pages Router, Remix, or SvelteKit **Date:** 2026-02-24 **Author:** Security-Test-Writer Agent **Status:** Accepted

1. Context

1.1 Situation

The Bilko frontend is built on Next.js 15. Next.js offers two routing systems: the legacy Pages Router and the modern App Router (stable since Next.js 13, production-ready since Next.js 14). The choice affects how data is fetched, how layouts are composed, and whether React Server Components (RSC) can be used.

1.2 Forces & Constraints

Technical forces:

- Dashboard, invoice list, and report pages are data-heavy — RSC reduces client-side JS bundle by fetching data server-side
- Bilko has multiple layout levels: root layout (sidebar + topbar), auth layout, public layout
- Next.js 15 App Router enables `layout.tsx` nesting — perfect for Bilko's multi-level navigation

Business forces:

- App Router is Next.js's strategic direction — Pages Router will be maintained but not enhanced
- Vercel's deployment optimizations (ISR, edge functions) are App Router-first

1.3 Problem Statement

> **We need to decide:** Whether to use Next.js App Router or Pages Router (and whether Next.js is the right choice vs. alternative frameworks).

2. Decision

We will: Use Next.js 15 App Router with the following patterns:

- **React Server Components** for data-fetch pages (invoice list, reports, dashboard)
- **Client Components** (`"use client"`) for interactive UI (invoice wizard, date pickers, forms)
- **Nested layouts** (`layout.tsx`) for shared navigation: `RootLayout → AppLayout (sidebar) → PageLayout`
- **Server Actions** for form submissions (planned for v1.1 — v1.0 uses API calls from Client Components)

3. Alternatives Considered

Option A: Next.js App Router ? Selected

Pros:

- RSC reduces client JS bundle — invoice list page ships no JS for the list rendering itself
- Nested layouts eliminate per-page layout boilerplate
- `loading.tsx` / `error.tsx` conventions for granular loading states
- Vercel deployment optimized for App Router (edge streaming, ISR)
- Future-proof — Pages Router receives no new features

Cons:

- App Router mental model is newer — some patterns (client/server component boundary) require learning
- Some third-party libraries not yet compatible with RSC (`"use client"` workarounds needed)

- Debugging is more complex (server vs. client stack traces differ)

Option B: Next.js Pages Router

Pros:

- Stable, well-documented, massive example library
- No client/server component boundary to reason about

Cons:

- No React Server Components — all data fetching in `getServerSideProps` or client-side
- Layout patterns are more manual (no `layout.tsx` nesting)
- Will not receive new features — effectively legacy

Why not chosen: Legacy approach; loses RSC performance benefits; will require migration later anyway.

Option C: Remix

Pros:

- Excellent loader/action model for data fetching and mutations
- Very good TypeScript support
- No client/server boundary confusion (different mental model)

Cons:

- Cannot deploy on Vercel without adapter configuration
- Learning curve for team familiar with Next.js
- Smaller ecosystem than Next.js
- Less integrated with React Server Components ecosystem

Why not chosen: Team familiarity with Next.js; Vercel deployment optimization; RSC is a better long-term bet.

Option D: SvelteKit

Pros:

- Excellent performance (minimal JS, true reactivity without VDOM)
- Simpler mental model than React
- Good TypeScript support

Cons:

- Different language (Svelte), not TypeScript + JSX — requires team retraining
- Smaller ecosystem — fewer accounting/UI component libraries
- Cannot share React types with backend TypeScript code

Why not chosen: Full framework switch — team TypeScript/React expertise would not transfer; no shared type system with backend.

4. Consequences

4.1 Positive Consequences

- Invoice list, reports, and dashboard pages are server-rendered — faster TTFB and LCP
- No prop-drilling for layout — `layout.tsx` provides shared sidebar/topbar automatically
- RSC reduces Time to Interactive — less JavaScript shipped to browser

4.2 Negative Consequences

- Client/server boundary requires care — passing non-serializable props to Client Components causes runtime errors — *Mitigation: ESLint rule `react-compiler` warns on component boundary violations*

4.3 Technical Debt Created

- Current v1.0 uses mock data from `lib/mock-data.ts` in Client Components — these must be replaced with RSC data fetching when the backend API is built — *Plan: replace all mock data with RSC fetches in v1.1 (backend integration milestone)*

Approval

Role	Name	Date	Signature
Author (ADR-001–006)	Petter Graff	2026-02-23	
Author (ADR-007–013)	Security-Test-Writer Agent	2026-02-24	
Tech Lead			
CTO / Architect	Alem Bašić		

Revision #3
Created 2026-02-24 22:50:52 UTC by John
Updated 2026-05-31 20:03:52 UTC by John