Authentication & Authorization

Bilko Authentication & Authorization

> **Status:** SPECIFICATION (backend not implemented) **Last updated:** 2026-02-20

Purpose

This document specifies the authentication and authorization system for Bilko's backend. Covers JWT tokens, password hashing, 2FA, role-based access control (RBAC), and session management.

Authentication Flow

System Overview

```
graph LR
    CLIENT[Frontend\nbilko.io]

    subgraph AUTH [Auth Layer]
        LOGIN[POST /auth/login]
        REGISTER[POST /auth/register]
        REFRESH[POST /auth/refresh]
        LOGOUT[POST /auth/logout]
    end

    subgraph TOKENS [Token Storage]
```

```
        AT[Access Token\nBearer header\n15 min TTL]
        RT[Refresh Token\nhttpOnly Cookie\n7-30 days TTL]
        BL[Blacklist\nRevoked JTIs]
    end

    subgraph GUARDS [Middleware Guards]
        AG[authGuard\nVerify JWT]
        RG[roleGuard\nCheck role]
        RL[rateLimiter\n5 req/min auth]
    end

    CLIENT --> LOGIN
    CLIENT --> REGISTER
    CLIENT --> REFRESH
    CLIENT --> LOGOUT

    LOGIN --> AT
    LOGIN --> RT
    REGISTER --> AT
    REGISTER --> RT
    REFRESH --> AT
    LOGOUT --> BL

    AT --> AG
    AG --> RG
    RG --> HANDLER[Route Handler]

    style AT fill:#00E5A0,color:#000
    style RT fill:#ffd700,color:#000
    style BL fill:#f87171,color:#fff
```

1. Registration

Endpoint: `POST /api/v1/auth/register`

```
sequenceDiagram
    participant C as Client
    participant API as Express API
    participant DB as PostgreSQL
    participant JWT as JWT Service
```

```
    C->>API: POST /auth/register\n{email, password, orgName, country}
    API->>API: Validate Zod schema\nCheck password strength
    API->>DB: Check email uniqueness
    DB-->>API: Email available
    API->>API: bcrypt.hash(password, 12)
    API->>DB: BEGIN TRANSACTION\nCreate Organization\nCreate User (role=owner)\nCreate default
Chart of Accounts
    DB-->>API: Organization + User created
    API->>JWT: Generate access token (15min)\nGenerate refresh token (7days)
    JWT-->>API: { accessToken, refreshToken }
    API->>C: 201 Created\n{ user, organization, tokens }\nSet-Cookie: refreshToken (httpOnly)
```

Steps:

1. Validate request body (email uniqueness, password strength, country/currency codes)
2. Hash password with bcrypt (12 rounds)
3. Create database transaction:
   - Create Organization
   - Create User (role = 'owner')
   - Create default Chart of Accounts (seed accounts based on country)
4. Generate JWT access token (15 min expiry)
5. Generate refresh token (7 days expiry)
6. Set refresh token in httpOnly cookie
7. Return user + organization + tokens

Password Requirements:

- Minimum 8 characters
- At least 1 uppercase letter
- At least 1 lowercase letter
- At least 1 number
- Optional: 1 special character

Password Hashing:

```
import bcrypt from 'bcrypt'

const SALT_ROUNDS = 12

async function hashPassword(password: string): Promise<string> {
  return bcrypt.hash(password, SALT_ROUNDS)
}
```

```
async function verifyPassword(password: string, hash: string): Promise<boolean> {
  return bcrypt.compare(password, hash)
}
```

Errors:

- `400 Bad Request` — Email already exists
- `422 Unprocessable Entity` — Weak password, invalid country code

---

2. Login

Endpoint: `POST /api/v1/auth/login`

```
sequenceDiagram
    participant C as Client
    participant RL as Rate Limiter\n5 req/min/IP
    participant API as Express API
    participant DB as PostgreSQL
    participant JWT as JWT Service

    C->>RL: POST /auth/login\n{email, password}
    RL-->>C: 429 Too Many Requests (if exceeded)
    RL->>API: Pass through
    API->>DB: Find user by email
    DB-->>API: User record
    API->>API: bcrypt.compare(password, hash)

    alt Invalid credentials
        API-->>C: 401 Unauthorized
    else 2FA required
        API-->>C: 403 { requiresTwoFactor: true }
        C->>API: POST /auth/verify-2fa { code }
        API->>API: speakeasy.totp.verify()
    end

    API->>DB: UPDATE users SET lastLoginAt = now()
    API->>JWT: Generate access token (15min)\nGenerate refresh token\n(7d or 30d if
rememberMe)
    JWT-->>API: Tokens
```

```
    API->>C: 200 OK { user, tokens }\nSet-Cookie: refreshToken (httpOnly, Secure,
SameSite=Strict)
```

Steps:

1. Find user by email
2. Verify password with bcrypt.compare()
3. If 2FA enabled, send TOTP challenge (not covered in MVP)
4. Update user.lastLoginAt
5. Generate JWT access token (15 min expiry)
6. Generate refresh token (7 days or 30 days if rememberMe = true)
7. Set refresh token in httpOnly cookie
8. Return user + tokens

Rate Limiting:

- Max 5 login attempts per 1 minute per IP address
- After 5 failed attempts, return `429 Too Many Requests`
- Lockout duration: 15 minutes

Errors:

- `401 Unauthorized` — Invalid email or password
- `403 Forbidden` — Account disabled or requires 2FA
- `429 Too Many Requests` — Rate limit exceeded

---

3. Token Refresh

Endpoint: `POST /api/v1/auth/refresh`

```
sequenceDiagram
    participant C as Client
    participant API as Express API
    participant BL as Blacklist\n(Redis/PostgreSQL)
    participant JWT as JWT Service

    C->>API: POST /auth/refresh\n[Cookie: refreshToken]
    API->>API: Extract JWT from httpOnly cookie
    API->>JWT: Verify signature & expiry
    JWT-->>API: RefreshTokenPayload { sub, jti, exp }
    API->>BL: Check if jti is blacklisted
    BL-->>API: Not blacklisted
```

```
    API->>JWT: Generate new access token (15min)
    JWT-->>API: New accessToken
    API->>C: 200 OK { accessToken }

    note over C,API: Access token expires every 15min\nClient must silently refresh via cookie
```

Steps:

1. Extract refresh token from httpOnly cookie
2. Verify refresh token signature
3. Check if token is blacklisted (revoked)
4. Check expiry
5. Generate new access token (15 min expiry)
6. Return new access token

Refresh Token Storage:

- Stored in httpOnly cookie (prevents XSS attacks)
- Secure flag = true (HTTPS only)
- SameSite = Strict (prevents CSRF attacks)
- Path = /api/v1/auth/refresh

Token Revocation:

- On logout, add refresh token to blacklist (Redis or PostgreSQL)
- Blacklist stores token JTI (JWT ID) + expiry
- Expired blacklist entries auto-deleted after 30 days

Errors:

- `401 Unauthorized` — Invalid or expired refresh token

---

4. Logout

Endpoint: `POST /api/v1/auth/logout`

Steps:

1. Extract refresh token from cookie
2. Add token JTI to blacklist
3. Clear httpOnly cookie
4. Return 204 No Content

JWT Tokens

Access Token

Purpose: Short-lived token for API authentication.

Claims:

```
interface AccessTokenPayload {
  sub: string           // User ID (UUID)
  email: string         // User email
  role: UserRole        // owner, admin, accountant, viewer
  orgId: string         // Organization ID (UUID)
  iat: number           // Issued at (Unix timestamp)
  exp: number           // Expires at (Unix timestamp, iat + 15 min)
}
```

Expiry: 15 minutes

Header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Usage:

- Sent in `Authorization: Bearer <token>` header
- Verified on every API request via `authGuard` middleware
- If expired, client requests new token via `/auth/refresh`

Refresh Token

Purpose: Long-lived token for obtaining new access tokens.

Claims:

```
interface RefreshTokenPayload {
  sub: string           // User ID (UUID)
```

```
  jti: string           // JWT ID (for revocation)
  iat: number           // Issued at (Unix timestamp)
  exp: number           // Expires at (Unix timestamp, iat + 7 days or 30 days)
}
```

Expiry: 7 days (default) or 30 days (if rememberMe = true)

Storage:

- httpOnly cookie (client cannot access via JavaScript)
- Secure = true (HTTPS only in production)
- SameSite = Strict (prevents CSRF)

Revocation:

- On logout, JTI added to blacklist
- On password change, all refresh tokens revoked
- On user deletion, all refresh tokens revoked

JWT Secret Management

CRITICAL: JWT secret MUST be stored securely.

Environment Variables:

```
# .env
JWT_SECRET=<256-bit random string, minimum 32 chars>
JWT_REFRESH_SECRET=<different 256-bit random string>
```

Generation:

```
# Generate secure random secret
openssl rand -base64 32
```

Best Practices:

- Use different secrets for access and refresh tokens
- Rotate secrets every 90 days (requires re-login for all users)
- Store in environment variables, NEVER in code
- Use Vaultwarden or similar secret manager in production

Two-Factor Authentication (2FA)

Status: OPTIONAL in MVP, implement in v2

```
sequenceDiagram
    participant U as User
    participant APP as Frontend
    participant API as Backend
    participant DB as PostgreSQL

    Note over U,DB: 2FA Setup Flow
    U->>APP: Enable 2FA in settings
    APP->>API: POST /settings/2fa/enable
    API->>API: Generate 32-char base32 TOTP secret
    API->>APP: { secret, qrCodeUrl }
    APP->>U: Display QR code
    U->>U: Scan with Google Authenticator / Authy
    U->>APP: Enter 6-digit code to verify
    APP->>API: POST /settings/2fa/verify { code }
    API->>API: speakeasy.totp.verify(secret, code)
    API->>DB: UPDATE users SET twoFactorEnabled=true\ntwoFactorSecret=encrypted
    API->>APP: 2FA activated

    Note over U,DB: Login with 2FA
    U->>APP: Enter email + password
    APP->>API: POST /auth/login
    API->>DB: Find user, verify password
    API->>APP: 403 { requiresTwoFactor: true }
    APP->>U: Prompt for 6-digit code
    U->>APP: Enter TOTP code
    APP->>API: POST /auth/verify-2fa { code }
    API->>API: Verify TOTP (30-second window ±1 step)
    API->>APP: 200 OK { user, tokens }
```

Flow:

1. User enables 2FA in settings
2. Generate TOTP secret (32-char base32 string)
3. Display QR code (Google Authenticator, Authy compatible)
4. User scans QR code

5. User enters 6-digit code to verify
6. Store `twoFactorSecret` (encrypted) in `users` table
7. Set `twoFactorEnabled = true`

Login with 2FA:

1. User enters email + password
2. If `twoFactorEnabled = true`, return `403` with `requiresTwoFactor: true`
3. Frontend prompts for 6-digit code
4. User submits code via `POST /api/v1/auth/verify-2fa`
5. Verify TOTP code (30-second window)
6. If valid, issue tokens

TOTP Verification:

```
import speakeasy from 'speakeasy'

function verifyTOTP(secret: string, token: string): boolean {
  return speakeasy.totp.verify({
    secret,
    encoding: 'base32',
    token,
    window: 1  // Allow 1 time step before/after (30s window)
  })
}
```

Role-Based Access Control (RBAC)

RBAC Model

```
graph TD
    subgraph ROLES [User Roles — Hierarchy]
        OW[owner\nFull control]
        AD[admin\nManage users + all financials]
        AC[accountant\nCreate financials only]
        VW[viewer\nRead-only]
    end

    subgraph ACTIONS [Protected Actions]
```

```
    direction LR
    INV_C[Create Invoice]
    INV_S[Send Invoice]
    INV_P[Mark Invoice Paid]
    EXP_C[Create Expense]
    EXP_AP[Approve Expense]
    TXN[Create Transaction]
    USR_I[Invite User]
    USR_R[Change User Role]
    USR_D[Delete User]
    ORG_S[Org Settings]
    ORG_D[Delete Organization]
    RPT[View Reports]
end

OW --> INV_C & INV_S & INV_P
OW --> EXP_C & EXP_AP
OW --> TXN
OW --> USR_I & USR_R & USR_D
OW --> ORG_S & ORG_D
OW --> RPT

AD --> INV_C & INV_S & INV_P
AD --> EXP_C & EXP_AP
AD --> TXN
AD --> USR_I
AD --> ORG_S
AD --> RPT

AC --> INV_C & INV_S & INV_P
AC --> EXP_C
AC --> TXN
AC --> RPT

VW --> RPT

style OW fill:#00E5A0,color:#000
style AD fill:#60a5fa,color:#000
style AC fill:#fbbf24,color:#000
```

```
    style VW fill:#94a3b8,color:#000
```

Roles

Role	Permissions
owner	Full access: manage users, delete organization, change settings, all financial operations
admin	Manage users (except owner), change settings, all financial operations
accountant	Create/edit invoices, expenses, transactions. View reports. Cannot manage users or settings.
viewer	Read-only access to all financial data. Cannot create or edit.

Permission Matrix

Action	owner	admin	accountant	viewer
Create invoice	✅	✅	✅	✅
Edit invoice (draft)	✅	✅	✅	✅
Send invoice	✅	✅	✅	✅
Mark invoice paid	✅	✅	✅	✅
Create expense	✅	✅	✅	✅
Approve expense	✅	✅	✅	✅
Create manual transaction	✅	✅	✅	✅
View reports	✅	✅	✅	✅
Invite user	✅	✅	✅	✅
Change user role	✅	✅	✅	✅
Delete user	✅	✅	✅	✅
Update org settings	✅	✅	✅	✅
Delete organization	✅	✅	✅	✅

Middleware Implementation

Role Guard:

```
import { Request, Response, NextFunction } from 'express'

type UserRole = 'owner' | 'admin' | 'accountant' | 'viewer'

function roleGuard(allowedRoles: UserRole[]) {
  return (req: Request, res: Response, next: NextFunction) => {
    const user = req.user  // Attached by authGuard middleware

    if (!user) {
      return res.status(401).json({ error: 'Unauthorized', code: 'NO_AUTH' })
    }

    if (!allowedRoles.includes(user.role)) {
      return res.status(403).json({
        error: 'Forbidden',
        code: 'INSUFFICIENT_PERMISSIONS',
        details: { required: allowedRoles, current: user.role }
      })
    }

    next()
  }
}

// Usage in routes
app.post('/api/v1/invoices',
  authGuard,
  roleGuard(['owner', 'admin', 'accountant']),
  createInvoice
)
```

Session Management

Session Storage

Option 1: JWT-only (stateless, recommended for MVP):

- No server-side session storage

- All state in JWT claims
- Fast, scales horizontally
- Cannot revoke access tokens (must wait for expiry)

Option 2: Redis sessions (for v2):

- Store session data in Redis
- JWT contains only session ID
- Can revoke immediately
- Requires Redis infrastructure

Recommended for MVP: JWT-only (stateless)

Session Invalidation

On password change:

1. Hash new password
2. Update `users.passwordHash`
3. Delete all refresh tokens from blacklist older than 1 hour (force re-login)
4. Return success

On account deletion:

1. Soft-delete user (set `isActive = false`)
2. Add all user's refresh tokens to blacklist
3. Revoke access immediately

Security Best Practices

1. Password Storage

- **NEVER** store plain text passwords
- Use bcrypt with 12 rounds (2^12 iterations)
- Bcrypt auto-salts (no need to store salt separately)

2. Token Security

- Access tokens in `Authorization` header (NOT cookies to avoid CSRF)
- Refresh tokens in httpOnly cookies (prevent XSS)

- Use Secure flag (HTTPS only)
- Use SameSite=Strict (prevent CSRF)

3. Rate Limiting

- Login: 5 attempts per minute per IP
- Register: 5 attempts per minute per IP
- Refresh: 100 attempts per minute per user
- All other endpoints: 100 requests per minute per user

4. HTTPS Only

- All traffic over HTTPS in production
- Redirect HTTP → HTTPS
- HSTS header: `Strict-Transport-Security: max-age=31536000; includeSubDomains`

5. CORS Configuration

```
const corsOptions = {
  origin: ['https://bilko.io', 'http://localhost:3000'],
  credentials: true,  // Allow cookies
  methods: ['GET', 'POST', 'PUT', 'PATCH', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization']
}

app.use(cors(corsOptions))
```

6. Input Validation

- Validate all inputs with Zod schemas
- Sanitize SQL inputs (Prisma prevents SQL injection)
- Escape HTML in user-generated content

Example Implementation

Auth Middleware

```
import jwt from 'jsonwebtoken'
import { Request, Response, NextFunction } from 'express'

interface AuthRequest extends Request {
  user?: {
    id: string
    email: string
    role: UserRole
    organizationId: string
  }
}

async function authGuard(req: AuthRequest, res: Response, next: NextFunction) {
  const authHeader = req.headers.authorization

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'Unauthorized', code: 'NO_TOKEN' })
  }

  const token = authHeader.substring(7)  // Remove 'Bearer '

  try {
    const payload = jwt.verify(token, process.env.JWT_SECRET!) as AccessTokenPayload

    // Attach user to request
    req.user = {
      id: payload.sub,
      email: payload.email,
      role: payload.role,
      organizationId: payload.orgId
    }

    next()
  } catch (error) {
    if (error.name === 'TokenExpiredError') {
      return res.status(401).json({ error: 'Token expired', code: 'TOKEN_EXPIRED' })
    }

    return res.status(401).json({ error: 'Invalid token', code: 'INVALID_TOKEN' })
  }
```

```
}

export { authGuard, roleGuard }
```

Environment Variables

```
# JWT
JWT_SECRET=<256-bit secret for access tokens>
JWT_REFRESH_SECRET=<256-bit secret for refresh tokens>
JWT_ACCESS_EXPIRY=15m
JWT_REFRESH_EXPIRY=7d

# Rate Limiting
RATE_LIMIT_AUTH=5           # Max login attempts per minute
RATE_LIMIT_GENERAL=100      # Max requests per minute

# Session
SESSION_COOKIE_SECURE=true  # HTTPS only (production)
SESSION_COOKIE_SAMESITE=strict
```

End of Authentication Documentation

Revision #3
Created 2026-02-23 10:47:52 UTC by John
Updated 2026-05-31 20:02:36 UTC by John