**Security and compliance plan**

**What you’ll get (at a glance)**

* **JWT end‑to‑end**: RS256 with **JWKS** + **key rotation**, short‑lived access tokens, refresh tokens, audience & role claims, **tenant binding**.
* **Service‑side enforcement**: shared **auth middleware** + **RBAC policy** helpers; reject non‑admin product creation; consistent X-Tenant-ID ↔ token.tid checks.
* **Audit Trail**: append‑only tables + Kafka **audit-events** producer library & a small **audit-service** consumer that writes tamper‑evident logs.
* **PII protection**: envelope encryption (AES‑256‑GCM) with per‑tenant DEKs; deterministic hashes for lookup; GDPR **export/delete** endpoints.
* **MFA**: TOTP enroll/verify flows in Auth; login telemetry + suspicious‑activity flags.
* **Integration Gateway**: API key **usage audit**, **Redis** rate‑limit, **Coinbase** webhook HMAC verify, (optional) **partner JWT/OAuth2**.
* **Observability**: Prometheus /metrics, structured JSON logs.
* **DB migrations**: Postgres DDL for keys, audit, PII columns, GDPR tombstones.

**Phased plan (execute in this order)**

1. **Foundations (shared libs)**
   * Add @company/common-auth, @company/common-audit, @company/common-crypto packages.
   * Wire every microservice to use common auth middleware (JWT + tenant + roles).
2. **Auth Service**
   * Implement JWT minting, **JWKS** endpoint, **key rotation**, **MFA**.
   * Add API key CRUD (hash at rest) for Integration Gateway partners.
3. **Audit**
   * Publish audit events from all services (including Integration Gateway).
   * Add **audit-service** consumer to persist immutable logs.
4. **Role enforcement**
   * Apply requireRole/requirePermission to protected routes (e.g., Product create).
5. **PII encryption & GDPR**
   * Customer Service: add encrypted columns + read/write hooks.
   * Add export + delete flows with audit + Kafka events.
6. **Integration Gateway**
   * Add API key usage audit, Redis limiter, Coinbase signature verification.
   * Optional partner JWT/OAuth2 token issue/validation.
7. **Monitoring**
   * Expose /metrics everywhere; add login/abuse counters; dashboards.

**Environment (new/updated)**

# Shared

JWT\_ISSUER=https://auth.yourdomain.com

JWT\_AUDIENCE=pos-api

JWKS\_CACHE\_TTL\_MS=600000

# Auth service

AUTH\_DB\_URL=postgres://...

ARGON2\_MEMORY\_COST=15360

ARGON2\_TIME\_COST=3

ARGON2\_PARALLELISM=1

TOKEN\_ACCESS\_TTL\_SECONDS=900

TOKEN\_REFRESH\_TTL\_SECONDS=2592000

MASTER\_KEY\_HEX=<32-byte hex> # for envelope encryption of tenant DEKs, api keys

KEY\_ROTATION\_INTERVAL\_HOURS=24

# Kafka

KAFKA\_BROKERS=broker1:9092,broker2:9092

KAFKA\_CLIENT\_ID=pos-platform

AUDIT\_TOPIC=audit.events.v1

# Customer service

CUSTOMER\_DB\_URL=postgres://...

# Integration gateway

REDIS\_URL=redis://...

COINBASE\_WEBHOOK\_SECRET=<from Coinbase>

GATEWAY\_RATE\_LIMIT\_RPM=60

# Audit service

AUDIT\_DB\_URL=postgres://...

**Database migrations (Postgres)**

Apply in your migration tool (Knex/Prisma/Flyway/etc.). Names are illustrative.

-- 001\_auth\_signing\_keys.sql

CREATE TABLE auth\_signing\_keys (

kid TEXT PRIMARY KEY,

public\_pem TEXT NOT NULL,

private\_pem TEXT NOT NULL,

alg TEXT NOT NULL DEFAULT 'RS256',

created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),

rotated\_at TIMESTAMPTZ,

active BOOLEAN NOT NULL DEFAULT TRUE

);

-- 002\_auth\_refresh\_tokens.sql

CREATE TABLE auth\_refresh\_tokens (

jti UUID PRIMARY KEY,

user\_id UUID NOT NULL,

tenant\_id TEXT NOT NULL,

issued\_at TIMESTAMPTZ NOT NULL DEFAULT now(),

expires\_at TIMESTAMPTZ NOT NULL,

revoked\_at TIMESTAMPTZ

);

-- 003\_auth\_api\_keys.sql

CREATE TABLE auth\_api\_keys (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

tenant\_id TEXT NOT NULL,

name TEXT NOT NULL,

key\_prefix TEXT NOT NULL, -- showable

key\_hash BYTEA NOT NULL, -- SHA-256(salt||key)

salt BYTEA NOT NULL,

created\_by UUID NOT NULL,

created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),

revoked\_at TIMESTAMPTZ

);

-- 010\_audit\_logs.sql (for audit-service)

CREATE TABLE audit\_logs (

id BIGSERIAL PRIMARY KEY,

occurred\_at TIMESTAMPTZ NOT NULL,

actor\_user\_id UUID,

actor\_tenant\_id TEXT NOT NULL,

actor\_roles TEXT[] NOT NULL,

action TEXT NOT NULL,

target\_type TEXT,

target\_id TEXT,

severity TEXT NOT NULL,

ip INET,

user\_agent TEXT,

meta JSONB,

event\_id UUID NOT NULL, -- from producer

prev\_row\_sig BYTEA, -- chain-of-custody

row\_sig BYTEA NOT NULL -- HMAC(master, row)

);

CREATE INDEX ON audit\_logs (actor\_tenant\_id, occurred\_at DESC);

-- 020\_customer\_pii\_encryption.sql

ALTER TABLE customers

ADD COLUMN enc\_key\_id TEXT, -- which tenant DEK encrypted these fields

ADD COLUMN email\_hash BYTEA, -- deterministic SHA-256(salt||lower(email))

ADD COLUMN first\_name\_enc BYTEA,

ADD COLUMN last\_name\_enc BYTEA,

ADD COLUMN email\_enc BYTEA,

ADD COLUMN phone\_enc BYTEA;

CREATE INDEX customers\_email\_hash\_idx ON customers (email\_hash);

-- 021\_tenant\_data\_keys.sql

CREATE TABLE tenant\_data\_keys (

tenant\_id TEXT NOT NULL,

key\_id TEXT NOT NULL,

enc\_dek BYTEA NOT NULL, -- AES-GCM(MASTER\_KEY, DEK)

created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),

rotated\_at TIMESTAMPTZ,

PRIMARY KEY (tenant\_id, key\_id)

);

-- 030\_gdpr\_tombstones.sql

CREATE TABLE gdpr\_tombstones (

tenant\_id TEXT NOT NULL,

user\_id UUID NOT NULL,

deleted\_at TIMESTAMPTZ NOT NULL DEFAULT now(),

reason TEXT,

PRIMARY KEY (tenant\_id, user\_id)

);

**New shared libraries (monorepo packages/)**

**packages/common-auth/src/index.ts**

// Lightweight, framework-agnostic helpers for Express/Fastify.

// pnpm add jose undici

import { createRemoteJWKSet, jwtVerify, JWTPayload } from 'jose';

import { URL } from 'url';

const ISSUER = process.env.JWT\_ISSUER!;

const AUDIENCE = process.env.JWT\_AUDIENCE!;

const JWKS\_URI = new URL('/.well-known/jwks.json', ISSUER).toString();

const JWKS = createRemoteJWKSet(new URL(JWKS\_URI));

const JWKS\_TTL = Number(process.env.JWKS\_CACHE\_TTL\_MS ?? 600000);

export type AuthContext = {

sub: string; // user id

tid: string; // tenant id

roles: string[]; // e.g. ['admin']

scope?: string[]; // optional scopes

};

export async function verifyBearer(authorization?: string): Promise<AuthContext> {

if (!authorization?.startsWith('Bearer ')) {

throw Object.assign(new Error('Missing/invalid Authorization header'), { status: 401 });

}

const token = authorization.slice('Bearer '.length);

const { payload } = await jwtVerify(token, JWKS, {

issuer: ISSUER,

audience: AUDIENCE,

maxTokenAge: '15m',

clockTolerance: '30s'

});

const ctx = mapPayload(payload);

if (!ctx.tid || !ctx.sub || !Array.isArray(ctx.roles)) {

throw Object.assign(new Error('Invalid token claims'), { status: 401 });

}

return ctx;

}

function mapPayload(p: JWTPayload): AuthContext {

return {

sub: String(p.sub),

tid: String((p as any).tid),

roles: (p as any).roles ?? [],

scope: (p as any).scope

};

}

// Express middleware

export function authMiddleware(req: any, res: any, next: any) {

verifyBearer(req.headers['authorization'])

.then(ctx => {

// Tenant header must match token claim

const headerTid = req.header('x-tenant-id');

if (!headerTid || headerTid !== ctx.tid) {

throw Object.assign(new Error('Tenant mismatch'), { status: 403 });

}

(req as any).auth = ctx;

next();

})

.catch(err => res.status(err.status || 401).json({ error: err.message }));

}

export function requireRole(...roles: string[]) {

return (req: any, res: any, next: any) => {

const ctx: AuthContext | undefined = (req as any).auth;

if (!ctx) return res.status(401).json({ error: 'Unauthenticated' });

if (!roles.some(r => ctx.roles.includes(r))) {

return res.status(403).json({ error: 'Forbidden: role required', need: roles });

}

next();

};

}

// Permission matrix (extensible)

const PERMISSIONS: Record<string,string[]> = {

'product:create': ['super\_admin','admin'],

'product:update': ['super\_admin','admin'],

'refund:create' : ['super\_admin','admin','manager'],

'user:create' : ['super\_admin','admin']

};

export function requirePermission(perm: keyof typeof PERMISSIONS) {

return (req: any, res: any, next: any) => {

const ctx: AuthContext | undefined = (req as any).auth;

if (!ctx) return res.status(401).json({ error: 'Unauthenticated' });

const allowed = PERMISSIONS[perm] ?? [];

if (!allowed.some(r => ctx.roles.includes(r))) {

return res.status(403).json({ error: `Forbidden: ${perm} requires ${allowed.join(',')}` });

}

next();

};

}

**packages/common-audit/src/index.ts**

// pnpm add kafkajs

import { Kafka, logLevel } from 'kafkajs';

import crypto from 'crypto';

const kafka = new Kafka({

clientId: process.env.KAFKA\_CLIENT\_ID || 'pos-platform',

brokers: (process.env.KAFKA\_BROKERS || 'localhost:9092').split(','),

logLevel: logLevel.NOTHING

});

const producer = kafka.producer();

let connected = false;

async function ensure() {

if (!connected) { await producer.connect(); connected = true; }

}

export type AuditEvent = {

eventId?: string;

occurredAt?: string; // ISO

action: string; // 'product.created', 'login.failed'

severity: 'info'|'warn'|'high';

actor: { userId?: string; tenantId: string; roles: string[]; };

target?: { type?: string; id?: string; };

ip?: string;

userAgent?: string;

meta?: Record<string, any>;

};

export async function audit(ev: AuditEvent) {

await ensure();

const eventId = ev.eventId ?? crypto.randomUUID();

const occurredAt = ev.occurredAt ?? new Date().toISOString();

const payload = { ...ev, eventId, occurredAt };

await producer.send({

topic: process.env.AUDIT\_TOPIC || 'audit.events.v1',

messages: [{ key: ev.actor.tenantId, value: JSON.stringify(payload) }]

});

}

**packages/common-crypto/src/index.ts**

// Envelope encryption helpers for per-tenant DEKs.

// pnpm add @noble/hashes

import crypto from 'crypto';

import { sha256 } from '@noble/hashes/sha256';

const MASTER\_KEY = Buffer.from(process.env.MASTER\_KEY\_HEX!, 'hex');

export function generateDEK(): Buffer {

return crypto.randomBytes(32);

}

export function encryptDEK(dek: Buffer): { iv: Buffer, tag: Buffer, ct: Buffer } {

const iv = crypto.randomBytes(12);

const cipher = crypto.createCipheriv('aes-256-gcm', MASTER\_KEY, iv);

const ct = Buffer.concat([cipher.update(dek), cipher.final()]);

const tag = cipher.getAuthTag();

return { iv, tag, ct };

}

export function decryptDEK(iv: Buffer, tag: Buffer, ct: Buffer): Buffer {

const decipher = crypto.createDecipheriv('aes-256-gcm', MASTER\_KEY, iv);

decipher.setAuthTag(tag);

const pt = Buffer.concat([decipher.update(ct), decipher.final()]);

return pt;

}

export type Encrypted = { iv: string, tag: string, ct: string }; // base64 strings

export function aesGcmEncrypt(plain: Buffer, dek: Buffer): Encrypted {

const iv = crypto.randomBytes(12);

const cipher = crypto.createCipheriv('aes-256-gcm', dek, iv);

const ct = Buffer.concat([cipher.update(plain), cipher.final()]);

const tag = cipher.getAuthTag();

return { iv: iv.toString('base64'), tag: tag.toString('base64'), ct: ct.toString('base64') };

}

export function aesGcmDecrypt(enc: Encrypted, dek: Buffer): Buffer {

const iv = Buffer.from(enc.iv, 'base64');

const tag = Buffer.from(enc.tag, 'base64');

const ct = Buffer.from(enc.ct, 'base64');

const decipher = crypto.createDecipheriv('aes-256-gcm', dek, iv);

decipher.setAuthTag(tag);

return Buffer.concat([decipher.update(ct), decipher.final()]);

}

// Deterministic hash for lookups (email) using SHA-256(salt||lower(value))

export function stableHash(value: string, salt: Buffer): Buffer {

const lower = value.trim().toLowerCase();

const h = sha256.create();

h.update(salt);

h.update(Buffer.from(lower));

return Buffer.from(h.digest());

}

**Auth Service (new/updated files)**

**Key features**

* RS256 JWT with kid header, **JWKS** endpoint
* Short‑lived Access, long‑lived Refresh tokens
* **MFA** TOTP enroll/verify
* API Keys (hashed) for partners
* Login telemetry + audit

Dependencies: pnpm add jose argon2 otplib qrcode kafkajs pg (or your ORM)

**auth-service/src/jwt.ts**

import crypto from 'crypto';

import { SignJWT, importPKCS8, exportJWK } from 'jose';

import { Pool } from 'pg';

const pool = new Pool({ connectionString: process.env.AUTH\_DB\_URL });

export async function currentSigningKey() {

const { rows } = await pool.query(

'SELECT kid, private\_pem, public\_pem FROM auth\_signing\_keys WHERE active = TRUE ORDER BY created\_at DESC LIMIT 1'

);

if (!rows[0]) throw new Error('No active signing key');

return rows[0] as { kid: string, private\_pem: string, public\_pem: string };

}

export async function mintAccessToken(payload: {

sub: string; tid: string; roles: string[]; scope?: string[];

}) {

const key = await currentSigningKey();

const alg = 'RS256';

const pk = await importPKCS8(key.private\_pem, alg);

const now = Math.floor(Date.now()/1000);

const exp = now + Number(process.env.TOKEN\_ACCESS\_TTL\_SECONDS || 900);

return new SignJWT({ ...payload })

.setProtectedHeader({ alg, kid: key.kid })

.setIssuer(process.env.JWT\_ISSUER!)

.setAudience(process.env.JWT\_AUDIENCE!)

.setSubject(payload.sub)

.setIssuedAt(now)

.setExpirationTime(exp)

.sign(pk);

}

export async function mintRefreshToken(userId: string, tenantId: string) {

const jti = crypto.randomUUID();

const now = new Date();

const ttlSec = Number(process.env.TOKEN\_REFRESH\_TTL\_SECONDS || 2592000);

const exp = new Date(now.getTime() + ttlSec\*1000);

await pool.query(

'INSERT INTO auth\_refresh\_tokens (jti, user\_id, tenant\_id, issued\_at, expires\_at) VALUES ($1,$2,$3,$4,$5)',

[jti, userId, tenantId, now, exp]

);

return { refreshToken: jti, expiresAt: exp };

}

export async function exposeJWKS() {

const { rows } = await pool.query('SELECT kid, public\_pem FROM auth\_signing\_keys WHERE active = TRUE');

const keys = await Promise.all(rows.map(async (r:any) => {

const jwk = await exportJWK({ type: 'public', format: 'pem', key: r.public\_pem } as any);

// jose exportJWK for PEM isn't direct; alternative: parse to KeyObject then exportJWK

// Simpler: build JWK via crypto

const keyObj = crypto.createPublicKey(r.public\_pem);

const jwk2 = await exportJWK(keyObj as any);

return { ...jwk2, kid: r.kid, alg: 'RS256', use: 'sig' };

}));

return { keys };

}

**auth-service/src/routes/jwks.ts**

import { Router } from 'express';

import { exposeJWKS } from '../jwt';

export const jwksRouter = Router();

jwksRouter.get('/.well-known/jwks.json', async (req, res) => {

try { res.json(await exposeJWKS()); }

catch (e:any) { res.status(500).json({ error: e.message }); }

});

**auth-service/src/routes/login.ts**

import { Router } from 'express';

import argon2 from 'argon2';

import { mintAccessToken, mintRefreshToken } from '../jwt';

import { audit } from '@company/common-audit';

import { Pool } from 'pg';

import { authenticator } from 'otplib';

const pool = new Pool({ connectionString: process.env.AUTH\_DB\_URL });

export const loginRouter = Router();

loginRouter.post('/v1/auth/login', async (req, res) => {

const { email, password, tenantId, mfaCode } = req.body ?? {};

const ip = req.ip; const ua = String(req.headers['user-agent'] || '');

try {

const { rows } = await pool.query(

'SELECT id, tenant\_id, pass\_hash, roles, mfa\_secret FROM users WHERE lower(email)=lower($1) AND tenant\_id=$2',

[email, tenantId]

);

if (!rows[0]) {

await audit({ action:'login.failed', severity:'warn', actor:{tenantId, roles:[], userId:undefined}, ip, userAgent:ua, meta:{email} });

return res.status(401).json({ error: 'Invalid credentials' });

}

const u = rows[0];

const ok = await argon2.verify(u.pass\_hash, password);

if (!ok) {

await audit({ action:'login.failed', severity:'warn', actor:{tenantId, roles:u.roles, userId:u.id}, ip, userAgent:ua });

return res.status(401).json({ error: 'Invalid credentials' });

}

if (u.mfa\_secret) {

if (!mfaCode || !authenticator.verify({ token: mfaCode, secret: u.mfa\_secret })) {

return res.status(401).json({ error: 'MFA required/invalid' });

}

}

const token = await mintAccessToken({ sub: u.id, tid: u.tenant\_id, roles: u.roles });

const { refreshToken, expiresAt } = await mintRefreshToken(u.id, u.tenant\_id);

await audit({ action:'login.success', severity:'info', actor:{tenantId: u.tenant\_id, roles:u.roles, userId:u.id}, ip, userAgent:ua });

res.json({ access\_token: token, token\_type:'Bearer', expires\_in: Number(process.env.TOKEN\_ACCESS\_TTL\_SECONDS||900), refresh\_token: refreshToken, refresh\_expires\_at: expiresAt });

} catch (e:any) {

res.status(500).json({ error: e.message });

}

});

**auth-service/src/routes/mfa.ts**

import { Router } from 'express';

import { authenticator } from 'otplib';

import QRCode from 'qrcode';

import { Pool } from 'pg';

import { authMiddleware } from '@company/common-auth';

const pool = new Pool({ connectionString: process.env.AUTH\_DB\_URL });

export const mfaRouter = Router();

mfaRouter.post('/v1/auth/mfa/enroll', authMiddleware, async (req:any, res) => {

const { sub, tid } = req.auth;

const secret = authenticator.generateSecret();

const otpauth = authenticator.keyuri(sub, `POS(${tid})`, secret);

const png = await QRCode.toDataURL(otpauth);

await pool.query('UPDATE users SET mfa\_secret=$1 WHERE id=$2 AND tenant\_id=$3', [secret, sub, tid]);

res.json({ qr\_data\_url: png, secret });

});

mfaRouter.post('/v1/auth/mfa/verify', authMiddleware, async (req:any, res) => {

const { sub, tid } = req.auth;

const { token } = req.body ?? {};

const { rows } = await pool.query('SELECT mfa\_secret FROM users WHERE id=$1 AND tenant\_id=$2', [sub, tid]);

if (!rows[0]?.mfa\_secret) return res.status(400).json({ error: 'Not enrolled' });

const ok = authenticator.verify({ token, secret: rows[0].mfa\_secret });

res.json({ verified: !!ok });

});

**auth-service/src/routes/apiKeys.ts**

import { Router } from 'express';

import crypto from 'crypto';

import { Pool } from 'pg';

import { authMiddleware, requireRole } from '@company/common-auth';

const pool = new Pool({ connectionString: process.env.AUTH\_DB\_URL });

export const apiKeysRouter = Router();

function hashKey(key:string, salt:Buffer) {

const h = crypto.createHash('sha256');

h.update(salt); h.update(key);

return h.digest();

}

apiKeysRouter.post('/v1/partners/apikeys', authMiddleware, requireRole('super\_admin','admin'), async (req:any,res) => {

const { name } = req.body ?? {};

const raw = 'pk\_' + crypto.randomBytes(24).toString('base64url');

const prefix = raw.slice(0,8);

const salt = crypto.randomBytes(16);

const keyHash = hashKey(raw, salt);

await pool.query('INSERT INTO auth\_api\_keys (tenant\_id, name, key\_prefix, key\_hash, salt, created\_by) VALUES ($1,$2,$3,$4,$5,$6)',

[req.auth.tid, name || prefix, prefix, keyHash, salt, req.auth.sub]);

res.json({ api\_key: raw, prefix });

});

apiKeysRouter.post('/v1/partners/apikeys/revoke', authMiddleware, requireRole('super\_admin','admin'), async (req:any,res) => {

const { id } = req.body ?? {};

await pool.query('UPDATE auth\_api\_keys SET revoked\_at=now() WHERE id=$1 AND tenant\_id=$2', [id, req.auth.tid]);

res.json({ ok: true });

});

**auth-service/src/routes/refresh.ts**

import { Router } from 'express';

import { Pool } from 'pg';

import { mintAccessToken } from '../jwt';

const pool = new Pool({ connectionString: process.env.AUTH\_DB\_URL });

export const refreshRouter = Router();

refreshRouter.post('/v1/auth/refresh', async (req, res) => {

const { refresh\_token, tenantId } = req.body ?? {};

const { rows } = await pool.query(

'SELECT r.user\_id, r.tenant\_id, u.roles, r.expires\_at FROM auth\_refresh\_tokens r JOIN users u ON u.id=r.user\_id WHERE jti=$1 AND r.tenant\_id=$2 AND r.revoked\_at IS NULL',

[refresh\_token, tenantId]

);

const r = rows[0];

if (!r || new Date(r.expires\_at) < new Date()) return res.status(401).json({ error: 'Invalid refresh' });

const access = await mintAccessToken({ sub: r.user\_id, tid: r.tenant\_id, roles: r.roles });

res.json({ access\_token: access, token\_type:'Bearer', expires\_in: Number(process.env.TOKEN\_ACCESS\_TTL\_SECONDS||900) });

});

**auth-service/src/jobs/rotate-keys.ts**

import crypto from 'crypto';

import { Pool } from 'pg';

const pool = new Pool({ connectionString: process.env.AUTH\_DB\_URL });

export async function rotateKeys() {

const { privateKey, publicKey } = crypto.generateKeyPairSync('rsa', { modulusLength: 2048 });

const private\_pem = privateKey.export({ type:'pkcs8', format:'pem' }).toString();

const public\_pem = publicKey.export({ type:'spki', format:'pem' }).toString();

const kid = crypto.randomUUID();

await pool.query('UPDATE auth\_signing\_keys SET active=FALSE WHERE active=TRUE');

await pool.query('INSERT INTO auth\_signing\_keys (kid, private\_pem, public\_pem, active) VALUES ($1,$2,$3,TRUE)', [kid, private\_pem, public\_pem]);

}

// run on interval via your scheduler or process manager

if (require.main === module) {

rotateKeys().then(()=>process.exit(0)).catch(e=>{console.error(e);process.exit(1);});

}

**Product Service — enforce RBAC (diff)**

diff --git a/product-service/src/server.ts b/product-service/src/server.ts

@@

-import express from 'express';

+import express from 'express';

+import { authMiddleware, requirePermission } from '@company/common-auth';

+import { audit } from '@company/common-audit';

const app = express();

app.use(express.json());

+app.use(authMiddleware);

-app.post('/v1/products', async (req, res) => {

+app.post('/v1/products', requirePermission('product:create'), async (req:any, res) => {

const tenantId = req.header('x-tenant-id');

const body = req.body;

// ... create product in DB, scoped to tenantId

+ await audit({

+ action: 'product.created',

+ severity: 'info',

+ actor: { tenantId: req.auth.tid, roles: req.auth.roles, userId: req.auth.sub },

+ target: { type: 'product', id: '...db-id...' },

+ meta: { sku: body.sku }

+ });

res.status(201).json({ ok:true });

});

**Customer Service — PII encryption + GDPR**

**Repository hooks (write‑encrypt / read‑decrypt)**

// customer-service/src/crypto/tenantKeys.ts

import { Pool } from 'pg';

import { decryptDEK, encryptDEK, generateDEK } from '@company/common-crypto';

const pool = new Pool({ connectionString: process.env.CUSTOMER\_DB\_URL });

export async function getTenantDEK(tenantId: string) {

const { rows } = await pool.query('SELECT key\_id, enc\_dek FROM tenant\_data\_keys WHERE tenant\_id=$1 ORDER BY created\_at DESC LIMIT 1', [tenantId]);

if (rows[0]) {

const buf = Buffer.from(rows[0].enc\_dek, 'base64');

const iv = buf.subarray(0,12), tag = buf.subarray(12,28), ct = buf.subarray(28);

const dek = decryptDEK(iv, tag, ct);

return { keyId: rows[0].key\_id, dek };

}

const dek = generateDEK();

const { iv, tag, ct } = encryptDEK(dek);

const keyId = `dek\_${Date.now()}`;

const packed = Buffer.concat([iv, tag, ct]).toString('base64');

await pool.query('INSERT INTO tenant\_data\_keys (tenant\_id, key\_id, enc\_dek) VALUES ($1,$2,$3)', [tenantId, keyId, packed]);

return { keyId, dek };

}

// customer-service/src/repository/customers.ts

import { Pool } from 'pg';

import { aesGcmEncrypt, aesGcmDecrypt, stableHash } from '@company/common-crypto';

import { getTenantDEK } from '../crypto/tenantKeys';

import crypto from 'crypto';

const pool = new Pool({ connectionString: process.env.CUSTOMER\_DB\_URL });

const EMAIL\_SALT = Buffer.from(process.env.MASTER\_KEY\_HEX!, 'hex').subarray(0,16); // reuse derivation or separate env

type CustomerInput = { firstName: string; lastName: string; email: string; phone?: string; };

export async function createCustomer(tenantId: string, input: CustomerInput) {

const { keyId, dek } = await getTenantDEK(tenantId);

const emailHash = stableHash(input.email, EMAIL\_SALT);

const firstEnc = aesGcmEncrypt(Buffer.from(input.firstName), dek);

const lastEnc = aesGcmEncrypt(Buffer.from(input.lastName), dek);

const emailEnc = aesGcmEncrypt(Buffer.from(input.email), dek);

const phoneEnc = input.phone ? aesGcmEncrypt(Buffer.from(input.phone), dek) : null;

const { rows } = await pool.query(

`INSERT INTO customers (tenant\_id, enc\_key\_id, email\_hash, first\_name\_enc, last\_name\_enc, email\_enc, phone\_enc)

VALUES ($1,$2,$3,$4,$5,$6,$7)

RETURNING id`,

[

tenantId, keyId, emailHash,

Buffer.from(JSON.stringify(firstEnc)), Buffer.from(JSON.stringify(lastEnc)),

Buffer.from(JSON.stringify(emailEnc)), phoneEnc ? Buffer.from(JSON.stringify(phoneEnc)) : null

]

);

return rows[0].id;

}

export async function getByEmail(tenantId: string, email: string) {

const { keyId, dek } = await getTenantDEK(tenantId);

const emailHash = stableHash(email, EMAIL\_SALT);

const { rows } = await pool.query('SELECT id, first\_name\_enc, last\_name\_enc, email\_enc, phone\_enc FROM customers WHERE tenant\_id=$1 AND email\_hash=$2 LIMIT 1', [tenantId, emailHash]);

if (!rows[0]) return null;

const dec = (buf: Buffer | null) => {

if (!buf) return null;

const enc = JSON.parse(buf.toString());

return aesGcmDecrypt(enc, dek).toString('utf8');

};

return {

id: rows[0].id,

firstName: dec(rows[0].first\_name\_enc),

lastName: dec(rows[0].last\_name\_enc),

email: dec(rows[0].email\_enc),

phone: dec(rows[0].phone\_enc)

};

}

**GDPR endpoints**

// customer-service/src/routes/gdpr.ts

import { Router } from 'express';

import { authMiddleware, requireRole } from '@company/common-auth';

import { audit } from '@company/common-audit';

import { getByEmail } from '../repository/customers';

import { Pool } from 'pg';

const pool = new Pool({ connectionString: process.env.CUSTOMER\_DB\_URL });

export const gdprRouter = Router();

gdprRouter.use(authMiddleware);

// Export by email (admin/manager)

gdprRouter.post('/v1/gdpr/export', requireRole('super\_admin','admin','manager'), async (req:any,res) => {

const { email } = req.body ?? {};

const data = await getByEmail(req.auth.tid, email);

await audit({ action:'gdpr.export', severity:'info', actor:{tenantId:req.auth.tid, roles:req.auth.roles, userId:req.auth.sub}, meta:{ email } });

res.json({ data });

});

// Delete user account (admin)

gdprRouter.post('/v1/gdpr/delete', requireRole('super\_admin','admin'), async (req:any,res) => {

const { userId } = req.body ?? {};

// Soft-delete + scramble PII; keep minimal order links if needed for financial retention policies

await pool.query('UPDATE customers SET first\_name\_enc=NULL, last\_name\_enc=NULL, email\_enc=NULL, phone\_enc=NULL WHERE tenant\_id=$1 AND id=$2', [req.auth.tid, userId]);

await pool.query('INSERT INTO gdpr\_tombstones (tenant\_id, user\_id, reason) VALUES ($1,$2,$3) ON CONFLICT DO NOTHING', [req.auth.tid, userId, 'right-to-be-forgotten']);

await audit({ action:'gdpr.delete', severity:'high', actor:{tenantId:req.auth.tid, roles:req.auth.roles, userId:req.auth.sub}, target:{type:'customer', id:userId} });

res.json({ ok:true });

});

**Integration Gateway**

**API key auth + audit + Redis rate limiter + Coinbase verify**

// integration-gateway/src/middleware/apiKeyAuth.ts

import { Pool } from 'pg';

import crypto from 'crypto';

import { audit } from '@company/common-audit';

const pool = new Pool({ connectionString: process.env.AUTH\_DB\_URL });

function hashKey(key:string, salt:Buffer) {

return crypto.createHash('sha256').update(salt).update(key).digest();

}

export async function apiKeyAuth(req:any,res:any,next:any) {

const key = req.header('x-api-key');

const tenantId = req.header('x-tenant-id');

if (!key || !tenantId) return res.status(401).json({ error: 'Missing key/tenant' });

const prefix = key.slice(0,8);

const { rows } = await pool.query('SELECT id, key\_hash, salt, revoked\_at FROM auth\_api\_keys WHERE tenant\_id=$1 AND key\_prefix=$2', [tenantId, prefix]);

if (!rows[0] || rows[0].revoked\_at) return res.status(401).json({ error:'Invalid key' });

const ok = crypto.timingSafeEqual(rows[0].key\_hash, hashKey(key, rows[0].salt));

if (!ok) return res.status(401).json({ error:'Invalid key' });

req.partner = { tenantId, keyId: rows[0].id, prefix };

await audit({

action: 'apikey.used',

severity: 'info',

actor: { tenantId, roles:['partner'], userId: undefined },

meta: { route: req.originalUrl, method: req.method, keyPrefix: prefix }

});

next();

}

// integration-gateway/src/middleware/rateLimiter.ts

import { createClient } from 'redis';

const client = createClient({ url: process.env.REDIS\_URL });

client.connect();

const RPM = Number(process.env.GATEWAY\_RATE\_LIMIT\_RPM || 60);

export async function rateLimiter(req:any,res:any,next:any) {

const key = `rl:${req.partner?.tenantId || req.ip}`;

const now = Math.floor(Date.now() / 1000);

const window = Math.floor(now / 60);

const redisKey = `${key}:${window}`;

const cnt = await client.incr(redisKey);

if (cnt === 1) await client.expire(redisKey, 60);

if (cnt > RPM) return res.status(429).json({ error: 'Rate limit exceeded' });

next();

}

// integration-gateway/src/routes/webhooks/coinbase.ts

import { Router } from 'express';

import crypto from 'crypto';

import { audit } from '@company/common-audit';

const SECRET = process.env.COINBASE\_WEBHOOK\_SECRET!;

export const coinbaseRouter = Router();

function verify(bodyRaw: string, signature: string) {

const h = crypto.createHmac('sha256', SECRET);

h.update(bodyRaw);

const expected = h.digest('hex');

return crypto.timingSafeEqual(Buffer.from(signature, 'hex'), Buffer.from(expected, 'hex'));

}

coinbaseRouter.post('/webhooks/coinbase', express.raw({ type: '\*/\*' }), async (req:any,res) => {

const sig = req.header('X-CC-Webhook-Signature') || '';

const ok = verify(req.body.toString('utf8'), sig);

if (!ok) return res.status(400).json({ error:'Invalid signature' });

const event = JSON.parse(req.body.toString('utf8'));

// ... process event safely

await audit({ action:'webhook.coinbase', severity:'info', actor:{tenantId: event?.data?.metadata?.tenantId ?? 'unknown', roles:['webhook']}, meta:{ id: event.id, type: event.type } });

res.json({ received: true });

});

Wire the middlewares on partner routes:

diff --git a/integration-gateway/src/server.ts b/integration-gateway/src/server.ts

@@

-import express from 'express';

+import express from 'express';

import cors from 'cors';

import helmet from 'helmet';

+import { apiKeyAuth } from './middleware/apiKeyAuth';

+import { rateLimiter } from './middleware/rateLimiter';

+import { coinbaseRouter } from './routes/webhooks/coinbase';

const app = express();

app.use(helmet());

app.use(cors({ origin: [/pos\.yourdomain\.com$/, /admin\.yourdomain\.com$/], credentials: true }));

app.use(express.json());

-// Partner routes:

-app.post('/v1/ext/orders', ..., handler);

+app.use('/v1/ext', apiKeyAuth, rateLimiter);

+app.post('/v1/ext/orders', /\* handler \*/ (req,res)=>res.json({ok:true}));

+// Webhooks

+app.use(coinbaseRouter);

app.get('/health', (req,res)=>res.json({ok:true}));

**Audit Service (consumer)**

// audit-service/src/index.ts

import { Kafka } from 'kafkajs';

import { Pool } from 'pg';

import crypto from 'crypto';

const kafka = new Kafka({ clientId: process.env.KAFKA\_CLIENT\_ID || 'audit', brokers: (process.env.KAFKA\_BROKERS || '').split(',') });

const consumer = kafka.consumer({ groupId: 'audit-service-v1' });

const pool = new Pool({ connectionString: process.env.AUDIT\_DB\_URL });

const MASTER = Buffer.from(process.env.MASTER\_KEY\_HEX!, 'hex');

function signRow(row:any) {

const h = crypto.createHmac('sha256', MASTER);

const s = JSON.stringify({ occurred\_at: row.occurred\_at, actor\_tenant\_id: row.actor\_tenant\_id, action: row.action, target\_type: row.target\_type, target\_id: row.target\_id, meta: row.meta });

h.update(s);

return h.digest();

}

async function getPrevSig(tenantId: string) {

const { rows } = await pool.query('SELECT row\_sig FROM audit\_logs WHERE actor\_tenant\_id=$1 ORDER BY occurred\_at DESC, id DESC LIMIT 1', [tenantId]);

return rows[0]?.row\_sig || null;

}

async function run() {

await consumer.connect();

await consumer.subscribe({ topic: process.env.AUDIT\_TOPIC || 'audit.events.v1' });

await consumer.run({

eachMessage: async ({ message }) => {

const ev = JSON.parse(String(message.value));

const prev = await getPrevSig(ev.actor.tenantId);

const res = await pool.query(

`INSERT INTO audit\_logs

(occurred\_at, actor\_user\_id, actor\_tenant\_id, actor\_roles, action, target\_type, target\_id, severity, ip, user\_agent, meta, event\_id, prev\_row\_sig, row\_sig)

VALUES ($1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14)

RETURNING id, occurred\_at, actor\_tenant\_id, action, target\_type, target\_id, meta`,

[ev.occurredAt, ev.actor.userId || null, ev.actor.tenantId, ev.actor.roles, ev.action, ev.target?.type || null, ev.target?.id || null, ev.severity, ev.ip || null, ev.userAgent || null, ev.meta || {}, ev.eventId, prev, Buffer.alloc(0)]

);

const row = res.rows[0];

const sig = signRow(row);

await pool.query('UPDATE audit\_logs SET row\_sig=$1 WHERE id=$2', [sig, row.id]);

}

});

}

run().catch(e => { console.error(e); process.exit(1); });

**Observability (add to each service)**

// <service>/src/metrics.ts

import client from 'prom-client';

const registry = new client.Registry();

client.collectDefaultMetrics({ register: registry });

export const requestCounter = new client.Counter({ name:'http\_requests\_total', help:'count', labelNames:['route','method','status'] });

registry.registerMetric(requestCounter);

export function metricsHandler(req:any,res:any){ res.set('Content-Type', registry.contentType); registry.metrics().then(m=>res.send(m)); }

diff --git a/<service>/src/server.ts b/<service>/src/server.ts

@@

+import { metricsHandler, requestCounter } from './metrics';

+app.use((req,res,next)=>{ const end = res.end; res.end = function(...args:any){ requestCounter.inc({ route:req.path, method:req.method, status: res.statusCode }); return end.apply(this,args); }; next(); });

+app.get('/metrics', metricsHandler);

**Tests (illustrative)**

// product-service/test/authz.spec.ts

import request from 'supertest';

import { mintAccessToken } from '../../auth-service/src/jwt';

it('rejects product create for cashier', async () => {

const token = await mintAccessToken({ sub:'u1', tid:'t1', roles:['cashier'] });

await request(app)

.post('/v1/products')

.set('Authorization', `Bearer ${token}`)

.set('X-Tenant-ID','t1')

.send({ name:'Test' })

.expect(403);

});

**Rollout guide (no downtime)**

1. **Deploy shared libs** and add as dependency in each service.
2. **Auth Service**
   * Run migrations 001–003.
   * Generate first signing key (rotate-keys.ts once).
   * Deploy JWKS + login/refresh/MFA/api-keys routes.
3. **Service adoption**
   * Add authMiddleware & role checks to write endpoints (start with highest‑risk: refunds, user mgmt, product create/update).
   * Keep old UUID token verification in gateway temporarily if you must bridge; **but** disable after clients move to JWT.
4. **Audit**
   * Deploy audit-service and direct all services to emit audit events via @company/common-audit.
5. **Customer PII**
   * Run migrations 020–021.
   * Deploy encryption hooks; backfill existing rows by re‑writing PII columns through the repository or one‑time script.
6. **GDPR**
   * Deploy routes; wire Admin UI to trigger export/delete.
7. **Integration Gateway**
   * Deploy API key audit + rate limiter; deploy Coinbase verify with secret; test with Coinbase webhook test utility.
8. **Metrics + Alerts**
   * Scrape /metrics; create dashboards & alerts for 401/403 spikes, rate limit hits, login.failed.

**PCI & Compliance checklists (skeleton docs to add to your repo)**

/docs/security/README.md

/docs/security/jwt-architecture.md

/docs/security/audit-model.md

/docs/security/pii-encryption.md

/docs/compliance/pci-dss-attestation-scope.md

/docs/compliance/gdpr-processes.md

Each doc should cover: data flows, components in scope/out, key handling, retention, incident response, and how the **integration with external payment providers** keeps cardholder data out of your environment (scope reduction).

**Threat modeling (quick highlights)**

* **Token replay / header spoofing**: Mitigated by server‑side JWT verify, aud/iss checks, X-Tenant-ID⇔tid match.
* **Privilege escalation**: Mitigated by backend RBAC checks (no front‑end trust).
* **Webhook forgery**: Mitigated by HMAC verification and (optional) timestamp tolerance.
* **PII exposure at rest**: Mitigated by per‑tenant DEKs + AES‑GCM, hashed lookup for emails.
* **Audit tampering**: Mitigated by append‑only table + chained HMAC signatures; Kafka provides durability.

**What to change in your code today**

* Add the **three shared packages** and import them across services.
* Swap any UUID token checks for authMiddleware.
* Gate sensitive routes with requireRole/requirePermission.
* Emit audit(...) events on sensitive actions.
* Merge the **PII repo** changes into Customer Service and run migrations.
* Integrate **Integration Gateway** middlewares for API key usage + Redis limiter + Coinbase verify.

If you paste these files into the indicated paths and run the migrations, you’ll have immediate, concrete security/compliance improvements with minimal churn. I can tailor the snippets to your actual repo structure if you share your service folders or a few representative files.