// SOC-APP\_DUMMY/backend/src/api/alerts.ts import { Router, Request, Response } from 'express'; import { AppDataSource } from '../data-source'; import { Source } from '../entity/Source'; import { RawAlert } from '../entity/RawAlert'; import { publish } from '../queue/producer';

const router = Router();

// POST /api/alerts router.post('/', async (req: Request, res: Response) => { try { const { sourceId, payload } = req.body as { sourceId: string; payload: Record<string, any>; };

// 1. Validate source exists  
const sourceRepo = AppDataSource.getRepository(Source);  
const source = await sourceRepo.findOneBy({ id: sourceId });  
if (!source) {  
 return res.status(404).json({ error: 'Source not found' });  
}  
  
// 2. Save raw alert to Postgres  
const rawAlertRepo = AppDataSource.getRepository(RawAlert);  
const rawAlert = rawAlertRepo.create({ source, payload });  
await rawAlertRepo.save(rawAlert);  
  
// 3. Publish to Kafka for downstream processing  
await publish('raw\_alerts', {  
 rawAlertId: rawAlert.id,  
 sourceId,  
 payload,  
 receivedAt: rawAlert.receivedAt.toISOString(),  
});  
  
return res.status(201).json({ id: rawAlert.id });

} catch (err) { console.error('Error ingesting alert:', err); return res.status(500).json({ error: 'Internal server error' }); } });

export default router;

// SOC-APP\_DUMMY/backend/src/api/feedback.ts import { Router } from 'express'; import { AppDataSource } from '../data-source'; import { Incident } from '../entity/Incident'; import { Feedback } from '../entity/Feedback'; import { User } from '../entity/User';

const feedbackRouter = Router();

// POST /api/feedback feedbackRouter.post('/', async (req, res) => { try { const { incidentId, userId, feedback } = req.body as { incidentId: string; userId: string; feedback: Record<string, any>; };

const incRepo = AppDataSource.getRepository(Incident);  
const userRepo = AppDataSource.getRepository(User);  
const incident = await incRepo.findOneBy({ id: incidentId });  
const user = await userRepo.findOneBy({ id: userId });  
if (!incident || !user) {  
 return res.status(404).json({ error: 'Incident or user not found' });  
}  
  
const fbRepo = AppDataSource.getRepository(Feedback);  
const fb = fbRepo.create({ incident, user, feedback });  
await fbRepo.save(fb);  
res.status(201).json(fb);

} catch (err) { console.error('Feedback error:', err); res.status(500).json({ error: 'Unable to submit feedback' }); } });

// GET /api/feedback feedbackRouter.get('/', async (\_req, res) => { try { const fbRepo = AppDataSource.getRepository(Feedback); const all = await fbRepo.find({ relations: ['incident', 'user'] }); res.json(all); } catch (err) { console.error('Fetch feedback error:', err); res.status(500).json({ error: 'Unable to fetch feedback' }); } });

export default feedbackRouter;

// SOC-APP\_DUMMY/backend/src/api/incidents.ts import { Router, Request, Response } from 'express'; import { AppDataSource } from '../data-source'; import { Incident } from '../entity/Incident'; import { Action } from '../entity/Action'; import { publish } from '../queue/producer';

const router = Router();

// GET /api/incidents router.get('/', async (\_req: Request, res: Response) => { try { const repo = AppDataSource.getRepository(Incident); const incidents = await repo.find({ relations: ['featureVector', 'actions', 'feedbacks'], order: { createdAt: 'DESC' }, }); return res.json(incidents); } catch (err) { console.error('Error fetching incidents:', err); return res.status(500).json({ error: 'Internal server error' }); } });

// POST /api/incidents/:id/action router.post('/:id/action', async (req: Request, res: Response) => { try { const incidentId = req.params.id; const { actionType, payload, performedBy } = req.body as { actionType: string; payload?: Record<string, any>; performedBy?: string; };

// 1. Find the incident  
const incidentRepo = AppDataSource.getRepository(Incident);  
const incident = await incidentRepo.findOneBy({ id: incidentId });  
if (!incident) {  
 return res.status(404).json({ error: 'Incident not found' });  
}  
  
// 2. Save the action  
const actionRepo = AppDataSource.getRepository(Action);  
const action = actionRepo.create({  
 incident,  
 actionType,  
 payload: payload || {},  
 performedBy: performedBy || 'system',  
});  
await actionRepo.save(action);  
  
// 3. Optionally publish feedback or follow-up event  
await publish('action\_performed', {  
 incidentId,  
 actionType,  
 performedBy,  
 timestamp: action.performedAt.toISOString(),  
});  
  
return res.status(201).json({ id: action.id });

} catch (err) { console.error('Error creating action:', err); return res.status(500).json({ error: 'Internal server error' }); } });

export default router;

// SOC-APP\_DUMMY/backend/src/api/incidents.ts import { Router, Request, Response } from 'express'; import { AppDataSource } from '../data-source'; import { Incident } from '../entity/Incident'; import { Action } from '../entity/Action'; import { publish } from '../queue/producer';

const router = Router();

// GET /api/incidents router.get('/', async (\_req: Request, res: Response) => { try { const repo = AppDataSource.getRepository(Incident); const incidents = await repo.find({ relations: ['featureVector', 'actions', 'feedbacks'], order: { createdAt: 'DESC' }, }); return res.json(incidents); } catch (err) { console.error('Error fetching incidents:', err); return res.status(500).json({ error: 'Internal server error' }); } });

// POST /api/incidents/:id/action router.post('/:id/action', async (req: Request, res: Response) => { try { const incidentId = req.params.id; const { actionType, payload, performedBy } = req.body as { actionType: string; payload?: Record<string, any>; performedBy?: string; };

// 1. Find the incident  
const incidentRepo = AppDataSource.getRepository(Incident);  
const incident = await incidentRepo.findOneBy({ id: incidentId });  
if (!incident) {  
 return res.status(404).json({ error: 'Incident not found' });  
}  
  
// 2. Save the action  
const actionRepo = AppDataSource.getRepository(Action);  
const action = actionRepo.create({  
 incident,  
 actionType,  
 payload: payload || {},  
 performedBy: performedBy || 'system',  
});  
await actionRepo.save(action);  
  
// 3. Optionally publish feedback or follow-up event  
await publish('action\_performed', {  
 incidentId,  
 actionType,  
 performedBy,  
 timestamp: action.performedAt.toISOString(),  
});  
  
return res.status(201).json({ id: action.id });

} catch (err) { console.error('Error creating action:', err); return res.status(500).json({ error: 'Internal server error' }); } });

export default router;

// SOC-APP\_DUMMY/backend/src/api/metrics.ts import { Router } from 'express'; import { AppDataSource } from '../data-source'; import { ModelMetric } from '../entity/ModelMetric';

const metricsRouter = Router();

// GET /api/metrics metricsRouter.get('/', async (\_req, res) => { try { const repo = AppDataSource.getRepository(ModelMetric); const metrics = await repo.find({ order: { runTs: 'DESC' }, take: 10, }); res.json(metrics); } catch (err) { console.error('Metrics error:', err); res.status(500).json({ error: 'Unable to fetch metrics' }); } });

export default metricsRouter;

// soc-app-dummy/backend/src/config/config.ts // This file contains configuration settings for the backend application. // It loads environment variables and exports constants for use throughout the application. import dotenv from 'dotenv'; dotenv.config();

export const KAFKA\_BROKERS = (process.env.KAFKA\_BROKERS || 'localhost:9092').split(','); export const KAFKA\_CLIENT\_ID = process.env.KAFKA\_CLIENT\_ID || 'soc-platform';

export const DB\_HOST = process.env.DB\_HOST || 'localhost'; export const DB\_PORT = parseInt(process.env.DB\_PORT || '5432', 10); export const DB\_USER = process.env.DB\_USER || 'postgres'; export const DB\_PASS = process.env.DB\_PASS || 'password'; export const DB\_NAME = process.env.DB\_NAME || 'socdb';

// SOC-APP\_DUMMY/backend/src/entity/Action.ts import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, ManyToOne } from 'typeorm'; import { Incident } from './Incident';

@Entity({ name: 'actions' }) export class Action { @PrimaryGeneratedColumn('uuid') id!: string;

@ManyToOne(() => Incident, inc => inc.actions, { onDelete: 'CASCADE' }) incident!: Incident;

@Column() actionType!: string; // AUTO\_QUARANTINE, ANALYST\_REVIEW

@Column({ type: 'jsonb', nullable: true }) payload!: Record<string, any>;

@Column({ nullable: true }) performedBy!: string;

@CreateDateColumn({ type: 'timestamptz' }) performedAt!: Date; }

// SOC-APP\_DUMMY/backend/src/entity/FeatureVector.ts // This file defines the FeatureVector entity which is used to store feature vectors extracted from normalized alerts // and associated with incidents. import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, ManyToOne, OneToOne, JoinColumn } from 'typeorm'; import { NormalizedAlert } from './NormalizedAlert'; import { Incident } from './Incident';

@Entity({ name: 'feature\_vectors' }) export class FeatureVector { @PrimaryGeneratedColumn('uuid') id!: string;

@ManyToOne(() => NormalizedAlert, na => na.featureVectors, { onDelete: 'CASCADE' }) normalizedAlert!: NormalizedAlert;

@Column({ type: 'jsonb' }) vector!: Record<string, any>;

@CreateDateColumn({ type: 'timestamptz' }) extractedAt!: Date;

@OneToOne(() => Incident, inc => inc.featureVector) @JoinColumn({ name: 'incident\_id' }) incident!: Incident; }

// SOC-APP\_DUMMY/backend/src/entity/Feedback.ts // This file defines the Feedback entity which is used to store user feedback on incidents. import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, ManyToOne } from 'typeorm'; import { Incident } from './Incident'; import { User } from './User';

@Entity({ name: 'feedback' }) export class Feedback { @PrimaryGeneratedColumn('uuid') id!: string;

@ManyToOne(() => Incident, inc => inc.feedbacks, { onDelete: 'CASCADE' }) incident!: Incident;

@ManyToOne(() => User, user => user.feedbacks, { onDelete: 'CASCADE' }) user!: User;

@Column({ type: 'jsonb' }) feedback!: Record<string, any>;

@CreateDateColumn({ type: 'timestamptz' }) submittedAt!: Date; }

// SOC-APP\_DUMMY/backend/src/entity/Incident.ts import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, UpdateDateColumn, OneToOne, OneToMany, JoinColumn } from 'typeorm'; import { FeatureVector } from './FeatureVector'; import { Action } from './Action'; import { Feedback } from './Feedback';

@Entity({ name: 'incidents' }) export class Incident { @PrimaryGeneratedColumn('uuid') id!: string;

@OneToOne(() => FeatureVector, fv => fv.incident, { onDelete: 'CASCADE' }) @JoinColumn({ name: 'feature\_vector\_id' }) featureVector!: FeatureVector;

@Column() decision!: 'AUTO' | 'MANUAL';

@Column('float') confidence!: number;

@Column({ default: 'open' }) status!: string;

@CreateDateColumn({ type: 'timestamptz' }) createdAt!: Date;

@UpdateDateColumn({ type: 'timestamptz', nullable: true }) closedAt!: Date;

@OneToMany(() => Action, act => act.incident) actions!: Action[];

@OneToMany(() => Feedback, fb => fb.incident) feedbacks!: Feedback[]; }

// SOC-APP\_DUMMY/backend/src/entity/ModelMetric.ts // This file defines the ModelMetric entity which is used to store metrics related to model performance. import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity({ name: 'model\_metrics' }) export class ModelMetric { @PrimaryGeneratedColumn('uuid') id!: string;

@Column({ type: 'timestamptz' }) runTs!: Date;

@Column('int') alertsProcessed!: number;

@Column('int') autoActions!: number;

@Column('int') manualReviews!: number;

@Column('float') precision!: number;

@Column('float') recall!: number;

@Column('int') latencyMs!: number; }

// SOC-APP\_DUMMY/backend/src/api/NormalizedAlerts.ts import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, ManyToOne, OneToMany } from 'typeorm'; import { RawAlert } from './RawAlert'; import { FeatureVector } from './FeatureVector';

@Entity({ name: 'normalized\_alerts' }) export class NormalizedAlert { @PrimaryGeneratedColumn('uuid') id!: string;

@ManyToOne(() => RawAlert, ra => ra.id, { onDelete: 'CASCADE' }) rawAlert!: RawAlert;

@Column({ type: 'timestamptz' }) timestamp!: Date;

@Column({ nullable: true }) alertType!: string;

@Column({ nullable: true }) severity!: string;

@Column({ type: 'jsonb', nullable: true }) metadata!: Record<string, any>;

@CreateDateColumn({ type: 'timestamptz' }) normalizedAt!: Date;

@OneToMany(() => FeatureVector, fv => fv.normalizedAlert) featureVectors!: FeatureVector[]; }

// SOC-APP\_DUMMY/backend/src/entity/RawAlert.ts import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, ManyToOne } from 'typeorm'; import { Source } from './Source'; import { NormalizedAlert } from './NormalizedAlert';

@Entity({ name: 'raw\_alerts' }) export class RawAlert { @PrimaryGeneratedColumn('uuid') id!: string;

@ManyToOne(() => Source, src => src.rawAlerts, { onDelete: 'CASCADE' }) source!: Source;

@Column({ type: 'jsonb' }) payload!: Record<string, any>;

@CreateDateColumn({ type: 'timestamptz' }) receivedAt!: Date; }

// SOC-APP\_DUMMY/backend/src/entity/Source.ts // This file defines the Source entity which is used to represent data sources for alerts. import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, OneToMany } from 'typeorm'; import { RawAlert } from './RawAlert';

@Entity({ name: 'sources' }) export class Source { @PrimaryGeneratedColumn('uuid') id!: string;

@Column() name!: string;

@Column() type!: string; // e.g. EDR, SIEM

@Column({ type: 'jsonb', nullable: true }) config!: Record<string, any>;

@CreateDateColumn({ type: 'timestamptz' }) createdAt!: Date;

@OneToMany(() => RawAlert, alert => alert.source) rawAlerts!: RawAlert[]; } // SOC-APP\_DUMMY/backend/src/entity/User.ts // This file defines the User entity which is used to represent users in the system. import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, OneToMany } from 'typeorm'; import { Feedback } from './Feedback';

@Entity({ name: 'users' }) export class User { @PrimaryGeneratedColumn('uuid') id!: string;

@Column({ unique: true }) username!: string;

@Column({ unique: true }) email!: string;

@Column() role!: 'analyst' | 'admin';

@CreateDateColumn({ type: 'timestamptz' }) createdAt!: Date;

@OneToMany(() => Feedback, fb => fb.user) feedbacks!: Feedback[]; }

// SOC-APP\_DUMMY/backend/src/migration/0001CreateSOCSchema1754434699078.ts import { MigrationInterface, QueryRunner } from "typeorm";

export class CreateSOCSchema1754434699078 implements MigrationInterface { name = 'CreateSOCSchema1754434699078'

public async up(queryRunner: QueryRunner): Promise<void> {  
 await queryRunner.query(`ALTER TABLE "raw\_alerts" DROP CONSTRAINT "FK\_raw\_alerts\_source"`);  
 await queryRunner.query(`ALTER TABLE "actions" DROP CONSTRAINT "FK\_actions\_incident"`);  
 await queryRunner.query(`ALTER TABLE "feedback" DROP CONSTRAINT "FK\_feedback\_incident"`);  
 await queryRunner.query(`ALTER TABLE "feedback" DROP CONSTRAINT "FK\_feedback\_user"`);  
 await queryRunner.query(`ALTER TABLE "incidents" DROP CONSTRAINT "FK\_incidents\_feature\_vector"`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" DROP CONSTRAINT "FK\_feature\_vectors\_normalized\_alert"`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" DROP CONSTRAINT "FK\_feature\_vectors\_incident"`);  
 await queryRunner.query(`ALTER TABLE "normalized\_alerts" DROP CONSTRAINT "FK\_normalized\_alerts\_raw\_alert"`);  
 await queryRunner.query(`DROP INDEX "public"."IDX\_raw\_alerts\_received\_at"`);  
 await queryRunner.query(`DROP INDEX "public"."IDX\_actions\_performed\_at"`);  
 await queryRunner.query(`DROP INDEX "public"."IDX\_feedback\_submitted\_at"`);  
 await queryRunner.query(`DROP INDEX "public"."IDX\_incidents\_status"`);  
 await queryRunner.query(`DROP INDEX "public"."IDX\_incidents\_created\_at"`);  
 await queryRunner.query(`DROP INDEX "public"."IDX\_normalized\_alerts\_timestamp"`);  
 await queryRunner.query(`DROP INDEX "public"."IDX\_normalized\_alerts\_severity"`);  
 await queryRunner.query(`ALTER TABLE "incidents" RENAME COLUMN "featureVectorId" TO "feature\_vector\_id"`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" RENAME COLUMN "incidentId" TO "incident\_id"`);  
 await queryRunner.query(`ALTER TABLE "incidents" ALTER COLUMN "closedAt" SET DEFAULT now()`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" ADD CONSTRAINT "UQ\_083508f614eb553a92fdf6674e3" UNIQUE ("incident\_id")`);  
 await queryRunner.query(`ALTER TABLE "raw\_alerts" ADD CONSTRAINT "FK\_4796a7a7e7be7b509c5d1dfcd17" FOREIGN KEY ("sourceId") REFERENCES "sources"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "actions" ADD CONSTRAINT "FK\_d6628cbad25cd09af676c877814" FOREIGN KEY ("incidentId") REFERENCES "incidents"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feedback" ADD CONSTRAINT "FK\_011bc00399f1b72107ae0d0bb31" FOREIGN KEY ("incidentId") REFERENCES "incidents"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feedback" ADD CONSTRAINT "FK\_4a39e6ac0cecdf18307a365cf3c" FOREIGN KEY ("userId") REFERENCES "users"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "incidents" ADD CONSTRAINT "FK\_443d5b240eb781d67b29152f2ff" FOREIGN KEY ("feature\_vector\_id") REFERENCES "feature\_vectors"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" ADD CONSTRAINT "FK\_361cf5e3da2174db33778865457" FOREIGN KEY ("normalizedAlertId") REFERENCES "normalized\_alerts"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" ADD CONSTRAINT "FK\_083508f614eb553a92fdf6674e3" FOREIGN KEY ("incident\_id") REFERENCES "incidents"("id") ON DELETE NO ACTION ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "normalized\_alerts" ADD CONSTRAINT "FK\_2fef533e672f1f45b8985df3ba8" FOREIGN KEY ("rawAlertId") REFERENCES "raw\_alerts"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
}  
  
public async down(queryRunner: QueryRunner): Promise<void> {  
 await queryRunner.query(`ALTER TABLE "normalized\_alerts" DROP CONSTRAINT "FK\_2fef533e672f1f45b8985df3ba8"`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" DROP CONSTRAINT "FK\_083508f614eb553a92fdf6674e3"`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" DROP CONSTRAINT "FK\_361cf5e3da2174db33778865457"`);  
 await queryRunner.query(`ALTER TABLE "incidents" DROP CONSTRAINT "FK\_443d5b240eb781d67b29152f2ff"`);  
 await queryRunner.query(`ALTER TABLE "feedback" DROP CONSTRAINT "FK\_4a39e6ac0cecdf18307a365cf3c"`);  
 await queryRunner.query(`ALTER TABLE "feedback" DROP CONSTRAINT "FK\_011bc00399f1b72107ae0d0bb31"`);  
 await queryRunner.query(`ALTER TABLE "actions" DROP CONSTRAINT "FK\_d6628cbad25cd09af676c877814"`);  
 await queryRunner.query(`ALTER TABLE "raw\_alerts" DROP CONSTRAINT "FK\_4796a7a7e7be7b509c5d1dfcd17"`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" DROP CONSTRAINT "UQ\_083508f614eb553a92fdf6674e3"`);  
 await queryRunner.query(`ALTER TABLE "incidents" ALTER COLUMN "closedAt" DROP DEFAULT`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" RENAME COLUMN "incident\_id" TO "incidentId"`);  
 await queryRunner.query(`ALTER TABLE "incidents" RENAME COLUMN "feature\_vector\_id" TO "featureVectorId"`);  
 await queryRunner.query(`CREATE INDEX "IDX\_normalized\_alerts\_severity" ON "normalized\_alerts" ("severity") `);  
 await queryRunner.query(`CREATE INDEX "IDX\_normalized\_alerts\_timestamp" ON "normalized\_alerts" ("timestamp") `);  
 await queryRunner.query(`CREATE INDEX "IDX\_incidents\_created\_at" ON "incidents" ("createdAt") `);  
 await queryRunner.query(`CREATE INDEX "IDX\_incidents\_status" ON "incidents" ("status") `);  
 await queryRunner.query(`CREATE INDEX "IDX\_feedback\_submitted\_at" ON "feedback" ("submittedAt") `);  
 await queryRunner.query(`CREATE INDEX "IDX\_actions\_performed\_at" ON "actions" ("performedAt") `);  
 await queryRunner.query(`CREATE INDEX "IDX\_raw\_alerts\_received\_at" ON "raw\_alerts" ("receivedAt") `);  
 await queryRunner.query(`ALTER TABLE "normalized\_alerts" ADD CONSTRAINT "FK\_normalized\_alerts\_raw\_alert" FOREIGN KEY ("rawAlertId") REFERENCES "raw\_alerts"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" ADD CONSTRAINT "FK\_feature\_vectors\_incident" FOREIGN KEY ("incidentId") REFERENCES "incidents"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feature\_vectors" ADD CONSTRAINT "FK\_feature\_vectors\_normalized\_alert" FOREIGN KEY ("normalizedAlertId") REFERENCES "normalized\_alerts"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "incidents" ADD CONSTRAINT "FK\_incidents\_feature\_vector" FOREIGN KEY ("featureVectorId") REFERENCES "feature\_vectors"("id") ON DELETE SET NULL ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feedback" ADD CONSTRAINT "FK\_feedback\_user" FOREIGN KEY ("userId") REFERENCES "users"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "feedback" ADD CONSTRAINT "FK\_feedback\_incident" FOREIGN KEY ("incidentId") REFERENCES "incidents"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "actions" ADD CONSTRAINT "FK\_actions\_incident" FOREIGN KEY ("incidentId") REFERENCES "incidents"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
 await queryRunner.query(`ALTER TABLE "raw\_alerts" ADD CONSTRAINT "FK\_raw\_alerts\_source" FOREIGN KEY ("sourceId") REFERENCES "sources"("id") ON DELETE CASCADE ON UPDATE NO ACTION`);  
}

} // SOC-APP\_DUMMY/backend/src/queue/consumer.ts // This file defines the Kafka consumer for processing messages from the queue. // It connects to the Kafka broker and subscribes to a specified topic to handle incoming messages. import { Kafka, Consumer } from 'kafkajs'; import { KAFKA\_BROKERS, KAFKA\_CLIENT\_ID } from '../config/config';

export async function createConsumer(groupId: string): Promise { const kafka = new Kafka({ clientId: KAFKA\_CLIENT\_ID, brokers: KAFKA\_BROKERS, });

const consumer = kafka.consumer({ groupId }); await consumer.connect(); console.log(✅ Kafka consumer connected (group: ${groupId})); return consumer; }

/\*\*

* Subscribe to a Kafka topic and process each message.
* @param topic Name of the Kafka topic
* @param groupId Consumer group ID
* @param handler Function to process each message

\*/ export async function subscribe( topic: string, groupId: string, handler: (message: any) => Promise ): Promise { const consumer = await createConsumer(groupId); await consumer.subscribe({ topic, fromBeginning: false });

await consumer.run({ eachMessage: async ({ message }) => { if (message.value) { try { const payload = JSON.parse(message.value.toString()); await handler(payload); } catch (err) { console.error('❌ Error processing message', err); } } }, }); }

// SOC-APP\_DUMMY/backend/src/queue/producer.ts // This file defines the Kafka producer for sending messages to the queue. // It connects to the Kafka broker and provides a method to publish messages to a specified topic. import { Kafka, Producer, RecordMetadata } from 'kafkajs'; import { KAFKA\_BROKERS, KAFKA\_CLIENT\_ID } from '../config/config';

let producer: Producer;

export async function initProducer(): Promise { if (producer) return producer;

const kafka = new Kafka({ clientId: KAFKA\_CLIENT\_ID, brokers: KAFKA\_BROKERS, });

producer = kafka.producer(); await producer.connect(); console.log('✅ Kafka producer connected'); return producer; }

/\*\*

* Publish JSON data to
* a Kafka topic.
* @param topic Name of the Kafka topic
* @param message A JS object (will be JSON-stringified)

\*/ export async function publish(topic: string, message: Record<string, any>): Promise<RecordMetadata[]> { const prod = await initProducer(); const payload = { topic, messages: [ { value: JSON.stringify(message) } ] }; return await prod.send(payload); }

// src/workers/featureWorker.ts // This file defines the FeatureWorker which processes normalized alerts to extract features. // It listens for messages from the queue, extracts features from normalized alerts, // and publishes feature vectors to another Kafka topic. import 'reflect-metadata'; import { AppDataSource } from '../data-source'; import { NormalizedAlert } from '../entity/NormalizedAlert'; import { FeatureVector } from '../entity/FeatureVector'; import { subscribe } from '../queue/consumer'; import { publish } from '../queue/producer';

interface NormalizedMsg { normalizedAlertId: string; timestamp: string; alertType: string; severity: string; metadata: any; }

async function featureHandler(msg: NormalizedMsg) { try { const na = await AppDataSource.getRepository(NormalizedAlert).findOneBy({ id: msg.normalizedAlertId, }); if (!na) return console.warn('NormalizedAlert not found:', msg.normalizedAlertId);

// Example feature extraction logic  
const vector = {  
 severityScore: msg.severity === 'high' ? 1 : 0,  
 typeHash: msg.alertType.length,  
 metaCount: Object.keys(msg.metadata || {}).length,  
};  
  
const fvRepo = AppDataSource.getRepository(FeatureVector);  
const fv = fvRepo.create({ normalizedAlert: na, vector });  
const saved = await fvRepo.save(fv);  
console.log(`FeatureVector saved: ${saved.id}`);  
  
await publish('feature\_vectors', {  
 featureVectorId: saved.id,  
 vector,  
 extractedAt: saved.extractedAt.toISOString(),  
});

} catch (err) { console.error('FeatureWorker error:', err); } }

async function startFeatureWorker() { await AppDataSource.initialize(); console.log('🔄 Starting FeatureWorker'); await subscribe('normalized\_alerts', 'feature-group', featureHandler); }

startFeatureWorker().catch(err => { console.error('FeatureWorker failed to start:', err); process.exit(1); });

// src/workers/inferenceWorker.ts // This file defines the InferenceWorker which processes feature vectors to create incidents. // It listens for messages from the queue, calls an AI inference endpoint, import 'reflect-metadata'; import axios from 'axios'; import { AppDataSource } from '../data-source'; import { FeatureVector } from '../entity/FeatureVector'; import { Incident } from '../entity/Incident'; import { subscribe } from '../queue/consumer'; import { publish } from '../queue/producer';

interface FeatureMsg { featureVectorId: string; vector: any; extractedAt: string; }

async function inferenceHandler(msg: FeatureMsg) { try { const fvRepo = AppDataSource.getRepository(FeatureVector); const fv = await fvRepo.findOneBy({ id: msg.featureVectorId }); if (!fv) return console.warn('FeatureVector not found:', msg.featureVectorId);

// Call your AI inference endpoint  
const response = await axios.post(process.env.AI\_ENDPOINT!, { vector: msg.vector });  
const { decision, confidence } = response.data;  
  
const incRepo = AppDataSource.getRepository(Incident);  
const inc = incRepo.create({ featureVector: fv, decision, confidence });  
const saved = await incRepo.save(inc);  
console.log(`Incident created: ${saved.id}`);  
  
await publish('ai\_predictions', {  
 incidentId: saved.id,  
 decision,  
 confidence,  
 createdAt: saved.createdAt.toISOString(),  
});

} catch (err) { console.error('InferenceWorker error:', err); } }

async function startInferenceWorker() { await AppDataSource.initialize(); console.log('🔄 Starting InferenceWorker'); await subscribe('feature\_vectors', 'inference-group', inferenceHandler); }

startInferenceWorker().catch(err => { console.error('InferenceWorker failed to start:', err); process.exit(1); });

// src/workers/normalizeWorker.ts import 'reflect-metadata'; import { AppDataSource } from '../data-source'; import { RawAlert } from '../entity/RawAlert'; import { NormalizedAlert } from '../entity/NormalizedAlert'; import { subscribe } from '../queue/consumer'; import { publish } from '../queue/producer';

interface RawAlertMessage { rawAlertId: string; sourceId: string; payload: Record<string, any>; receivedAt: string; }

async function normalizeHandler(msg: RawAlertMessage) { const { rawAlertId, payload, receivedAt } = msg; try { // 1. Load the raw alert entity const rawRepo = AppDataSource.getRepository(RawAlert); const rawAlert = await rawRepo.findOneBy({ id: rawAlertId }); if (!rawAlert) { console.warn(RawAlert ${rawAlertId} not found); return; }

// 2. Perform normalization logic  
// (Example: map vendor fields to your schema)  
const timestamp = new Date(payload.eventTime || receivedAt);  
const alertType = payload.alertType || payload.type || 'unknown';  
const severity = payload.severity || 'medium';  
const metadata = {  
 host: payload.host,  
 user: payload.user,  
 ip: payload.sourceIp,  
 raw: payload,  
};  
  
// 3. Persist to normalized\_alerts table  
const normRepo = AppDataSource.getRepository(NormalizedAlert);  
const norm = normRepo.create({  
 rawAlert,  
 timestamp,  
 alertType,  
 severity,  
 metadata,  
});  
const saved = await normRepo.save(norm);  
console.log(`✅ NormalizedAlert saved: ${saved.id}`);  
  
// 4. Publish to next Kafka topic  
await publish('normalized\_alerts', {  
 normalizedAlertId: saved.id,  
 rawAlertId,  
 timestamp: saved.timestamp.toISOString(),  
 alertType,  
 severity,  
 metadata,  
});

} catch (err) { console.error('❌ Error in normalizeHandler:', err); } }

async function start() { // Ensure DB is initialized await AppDataSource.initialize();

console.log('🔄 Starting Normalization Worker...'); subscribe('raw\_alerts', 'normalize-group', normalizeHandler) .catch(err => { console.error('Failed to subscribe to raw\_alerts:', err); process.exit(1); }); }

start();

// src/workers/inferenceWorker.ts // This file defines the InferenceWorker which processes feature vectors to create incidents. // It listens for messages from the queue, calls an AI inference endpoint, // and publishes incident data to another Kafka topic. import 'reflect-metadata'; import { DataSource } from 'typeorm'; import { Source } from './entity/Source'; import { RawAlert } from './entity/RawAlert'; import { NormalizedAlert } from './entity/NormalizedAlert'; import { FeatureVector } from './entity/FeatureVector'; import { Incident } from './entity/Incident'; import { Action } from './entity/Action'; import { User } from './entity/User'; import { Feedback } from './entity/Feedback'; import { ModelMetric } from './entity/ModelMetric'; import 'reflect-metadata'; import \* as dotenv from 'dotenv'; dotenv.config();

export const AppDataSource = new DataSource({ type: 'postgres', host: process.env.DB\_HOST, port: Number(process.env.DB\_PORT), username: process.env.DB\_USER, password: process.env.DB\_PASS, database: process.env.DB\_NAME, synchronize: false, // use migrations in production logging: true, entities: [ Source, RawAlert, NormalizedAlert, FeatureVector, Incident, Action, User, Feedback, ModelMetric ], migrations: ['src/migration/\*\*/\*.ts'], });

// src/server.ts // This file sets up the Express server, connects to the database, and initializes the API routes. // It also subscribes to Kafka topics for real-time data streaming and handles Server-Sent Events import 'reflect-metadata'; import express from 'express'; import cors from 'cors'; import { AppDataSource } from './data-source'; import alertsRouter from './api/alerts'; import incidentsRouter from './api/incidents'; import metricsRouter from './api/metrics'; import feedbackRouter from './api/feedback'; import { subscribe } from './queue/consumer'; import { Kafka } from 'kafkajs';

type SSEClient = express.Response; const sseClients: SSEClient[] = [];

async function start() { await AppDataSource.initialize(); console.log('✅ Database connected');

const kafka = new Kafka({ clientId: process.env.KAFKA\_CLIENT\_ID!, brokers: process.env.KAFKA\_BROKERS!.split(','), }); // Create topics before subscribing const admin = kafka.admin(); await admin.connect(); await admin.createTopics({ waitForLeaders: true, topics: [ { topic: 'ai\_predictions', numPartitions: 1, replicationFactor: 1 }, // Add other topics you consume or produce ], }); await admin.disconnect();

// Then subscribe await subscribeKafka(kafka);

// ... existing Express + SSE setup ... } async function subscribeKafka(kafka: Kafka) { await subscribe('ai\_predictions', 'stream-group', async msg => { const dataStr = data: ${JSON.stringify(msg)}\n\n; sseClients.forEach(client => client.write(dataStr)); }); console.log('✅ Subscribed to ai\_predictions topic');

const app = express(); app.use(cors()); app.use(express.json());

app.use('/api/alerts', alertsRouter); app.use('/api/incidents', incidentsRouter); app.use('/api/metrics', metricsRouter); app.use('/api/feedback', feedbackRouter);

// Threat Intel API (stub) app.get('/api/threatintel', (\_req, res) => { const timestamp = new Date(); res.json([ { id: 'ioc1', type: 'IP', value: '198.51.100.23', firstSeen: timestamp.toISOString() }, { id: 'ioc2', type: 'Domain', value: 'malware.example.com', firstSeen: timestamp.toISOString() }, ]); });

// SSE Endpoint for real-time ai\_predictions app.get('/api/stream/ai\_predictions', (req, res) => { res.setHeader('Content-Type', 'text/event-stream'); res.setHeader('Cache-Control', 'no-cache'); res.setHeader('Connection', 'keep-alive'); res.flushHeaders();

// send initial greeting  
res.write(`data: ${JSON.stringify({ message: 'connected' })}\n\n`);  
  
sseClients.push(res);  
req.on('close', () => {  
 const idx = sseClients.indexOf(res);  
 if (idx >= 0) sseClients.splice(idx, 1);  
});

});

// Subscribe to ai\_predictions Kafka topic and broadcast subscribe('ai\_predictions', 'stream-group', async (msg: any) => { const dataStr = data: ${JSON.stringify(msg)}\n\n; sseClients.forEach(client => client.write(dataStr)); }).catch(err => console.error('SSE subscribe error', err));

const port = process.env.PORT || 3004; app.listen(port, () => console.log(🚀 API listening on http://localhost:${port})); }

start().catch(err => { console.error('Startup error:', err); process.exit(1); });

{ "compilerOptions": { "target": "ES2020", "module": "CommonJS", "lib": ["ES2020"], "outDir": "dist", "rootDir": "src", "strict": true, "moduleResolution": "node", "esModuleInterop": true, "emitDecoratorMetadata": true, "experimentalDecorators": true, "skipLibCheck": true }, "include": ["src/\*\*/\*.ts"], "exclude": ["node\_modules", "dist"] }