**Monitoreo, Rendimiento y Seguridad Avanzada - TurisGal**

**1. SISTEMA DE MONITOREO AVANZADO**

**Application Performance Monitoring (APM)**

// src/monitoring/APMService.ts

import \* as Sentry from '@sentry/node';

import { createPrometheusMetrics } from 'prometheus-api-metrics';

import { Logger } from 'winston';

export class APMService {

private metrics: any;

private logger: Logger;

constructor(logger: Logger) {

this.logger = logger;

this.initializeSentry();

this.initializePrometheus();

this.setupCustomMetrics();

}

private initializeSentry(): void {

Sentry.init({

dsn: process.env.SENTRY\_DSN,

environment: process.env.NODE\_ENV,

tracesSampleRate: process.env.NODE\_ENV === 'production' ? 0.1 : 1.0,

profilesSampleRate: 0.1,

integrations: [

new Sentry.Integrations.Http({ tracing: true }),

new Sentry.Integrations.Express({ app: require('../app') }),

new Sentry.Integrations.Postgres(),

],

beforeSend: (event) => {

// Filtrar información sensible

this.sanitizeEvent(event);

return event;

}

});

}

private setupCustomMetrics(): void {

// Métricas de negocio

this.metrics = {

checkinSuccess: new prometheus.Counter({

name: 'turisgal\_checkin\_success\_total',

help: 'Total successful check-ins',

labelNames: ['property\_id', 'user\_type']

}),

checkinFailure: new prometheus.Counter({

name: 'turisgal\_checkin\_failure\_total',

help: 'Total failed check-ins',

labelNames: ['property\_id', 'error\_type']

}),

verificationTime: new prometheus.Histogram({

name: 'turisgal\_verification\_duration\_seconds',

help: 'Time taken for identity verification',

labelNames: ['verification\_type'],

buckets: [0.5, 1, 2, 5, 10, 30, 60, 120]

}),

userSatisfaction: new prometheus.Gauge({

name: 'turisgal\_user\_satisfaction\_score',

help: 'Average user satisfaction score',

labelNames: ['property\_id']

}),

revenueGenerated: new prometheus.Counter({

name: 'turisgal\_revenue\_euros\_total',

help: 'Total revenue generated in euros',

labelNames: ['property\_id', 'payment\_method']

}),

activeUsers: new prometheus.Gauge({

name: 'turisgal\_active\_users',

help: 'Number of currently active users',

labelNames: ['user\_type']

}),

databaseConnections: new prometheus.Gauge({

name: 'turisgal\_database\_connections',

help: 'Number of active database connections'

}),

queueSize: new prometheus.Gauge({

name: 'turisgal\_queue\_size',

help: 'Size of processing queues',

labelNames: ['queue\_name']

})

};

}

// Monitoreo de transacciones críticas

async monitorCriticalTransaction<T>(

operationName: string,

operation: () => Promise<T>,

metadata?: Record<string, any>

): Promise<T> {

const transaction = Sentry.startTransaction({

name: operationName,

op: 'critical\_operation'

});

const startTime = Date.now();

let result: T;

try {

Sentry.configureScope(scope => {

scope.setTransaction(transaction);

if (metadata) {

Object.entries(metadata).forEach(([key, value]) => {

scope.setTag(key, value);

});

}

});

result = await operation();

// Métricas de éxito

this.recordOperationSuccess(operationName, Date.now() - startTime, metadata);

this.logger.info(`Critical operation ${operationName} completed successfully`, {

operation: operationName,

duration: Date.now() - startTime,

metadata

});

return result;

} catch (error) {

// Capturar error en Sentry

Sentry.captureException(error, {

tags: {

operation: operationName,

critical: true

},

extra: metadata

});

// Métricas de error

this.recordOperationFailure(operationName, error, metadata);

this.logger.error(`Critical operation ${operationName} failed`, {

operation: operationName,

error: error.message,

stack: error.stack,

metadata

});

throw error;

} finally {

transaction.finish();

}

}

// Alertas personalizadas

async checkSystemHealth(): Promise<HealthStatus> {

const healthChecks = [

this.checkDatabaseHealth(),

this.checkRedisHealth(),

this.checkS3Health(),

this.checkExternalAPIsHealth(),

this.checkQueueHealth()

];

const results = await Promise.allSettled(healthChecks);

const health: HealthStatus = {

overall: 'healthy',

services: {},

timestamp: new Date(),

uptime: process.uptime()

};

results.forEach((result, index) => {

const serviceName = ['database', 'redis', 's3', 'external\_apis', 'queues'][index];

if (result.status === 'fulfilled') {

health.services[serviceName] = result.value;

} else {

health.services[serviceName] = {

status: 'unhealthy',

error: result.reason?.message,

lastCheck: new Date()

};

health.overall = 'degraded';

}

});

// Enviar alertas si hay problemas críticos

await this.processHealthAlerts(health);

return health;

}

private async processHealthAlerts(health: HealthStatus): Promise<void> {

const criticalServices = ['database', 'redis'];

const degradedServices = Object.entries(health.services)

.filter(([service, status]) => status.status !== 'healthy')

.map(([service]) => service);

if (degradedServices.some(service => criticalServices.includes(service))) {

await this.sendCriticalAlert({

level: 'critical',

message: `Critical services degraded: ${degradedServices.join(', ')}`,

services: degradedServices,

timestamp: new Date()

});

} else if (degradedServices.length > 0) {

await this.sendWarningAlert({

level: 'warning',

message: `Services experiencing issues: ${degradedServices.join(', ')}`,

services: degradedServices,

timestamp: new Date()

});

}

}

// Análisis de rendimiento en tiempo real

async analyzePerformanceTrends(): Promise<PerformanceAnalysis> {

const timeWindow = 24 \* 60 \* 60 \* 1000; // 24 horas

const now = new Date();

const yesterday = new Date(now.getTime() - timeWindow);

const analysis = await Promise.all([

this.getResponseTimeDistribution(yesterday, now),

this.getErrorRateAnalysis(yesterday, now),

this.getThroughputAnalysis(yesterday, now),

this.getResourceUtilization(yesterday, now)

]);

const [responseTime, errorRate, throughput, resources] = analysis;

// Detectar anomalías

const anomalies = await this.detectAnomalies({

responseTime,

errorRate,

throughput,

resources

});

return {

responseTime,

errorRate,

throughput,

resources,

anomalies,

recommendations: await this.generatePerformanceRecommendations(analysis),

timestamp: now

};

}

private async detectAnomalies(metrics: any): Promise<Anomaly[]> {

const anomalies: Anomaly[] = [];

// Detección de anomalías en tiempo de respuesta

if (metrics.responseTime.p95 > metrics.responseTime.baseline.p95 \* 1.5) {

anomalies.push({

type: 'performance',

metric: 'response\_time',

severity: 'high',

message: `P95 response time ${metrics.responseTime.p95}ms exceeds baseline by 50%`,

value: metrics.responseTime.p95,

baseline: metrics.responseTime.baseline.p95,

timestamp: new Date()

});

}

// Detección de picos de error

if (metrics.errorRate.current > metrics.errorRate.baseline \* 2) {

anomalies.push({

type: 'reliability',

metric: 'error\_rate',

severity: 'critical',

message: `Error rate ${(metrics.errorRate.current \* 100).toFixed(2)}% is ${(metrics.errorRate.current / metrics.errorRate.baseline).toFixed(1)}x higher than baseline`,

value: metrics.errorRate.current,

baseline: metrics.errorRate.baseline,

timestamp: new Date()

});

}

return anomalies;

}

}

// Sistema de alertas integrado

export class AlertingService {

private channels: Map<string, AlertChannel> = new Map();

constructor() {

this.initializeChannels();

}

private initializeChannels(): void {

// Slack

this.channels.set('slack', {

type: 'slack',

webhook: process.env.SLACK\_WEBHOOK\_URL!,

send: this.sendSlackAlert.bind(this)

});

// Email

this.channels.set('email', {

type: 'email',

config: { smtp: process.env.SMTP\_CONFIG },

send: this.sendEmailAlert.bind(this)

});

// SMS (para alertas críticas)

this.channels.set('sms', {

type: 'sms',

config: { twilio: process.env.TWILIO\_CONFIG },

send: this.sendSMSAlert.bind(this)

});

// PagerDuty

this.channels.set('pagerduty', {

type: 'pagerduty',

config: { token: process.env.PAGERDUTY\_TOKEN },

send: this.sendPagerDutyAlert.bind(this)

});

}

async sendAlert(alert: Alert): Promise<void> {

const channelsToUse = this.getChannelsForSeverity(alert.severity);

const promises = channelsToUse.map(channelName => {

const channel = this.channels.get(channelName);

return channel?.send(alert);

});

await Promise.allSettled(promises);

// Registrar la alerta

await this.logAlert(alert);

}

private getChannelsForSeverity(severity: AlertSeverity): string[] {

switch (severity) {

case 'critical':

return ['slack', 'email', 'sms', 'pagerduty'];

case 'high':

return ['slack', 'email'];

case 'medium':

return ['slack'];

case 'low':

return ['email'];

default:

return ['slack'];

}

}

private async sendSlackAlert(alert: Alert): Promise<void> {

const webhook = this.channels.get('slack')?.webhook;

if (!webhook) return;

const color = this.getSeverityColor(alert.severity);

const payload = {

attachments: [{

color,

title: `🚨 ${alert.title}`,

text: alert.message,

fields: [

{

title: 'Severity',

value: alert.severity.toUpperCase(),

short: true

},

{

title: 'Service',

value: alert.service || 'TurisGal API',

short: true

},

{

title: 'Timestamp',

value: alert.timestamp.toISOString(),

short: true

}

],

footer: 'TurisGal Monitoring',

ts: Math.floor(alert.timestamp.getTime() / 1000)

}]

};

await axios.post(webhook, payload);

}

}

**2. OPTIMIZACIÓN DE RENDIMIENTO**

**Cache Strategy Avanzado**

// src/cache/CacheService.ts

export class CacheService {

private redisClient: Redis.Redis;

private localCache: NodeCache;

constructor() {

this.redisClient = new Redis(process.env.REDIS\_URL);

this.localCache = new NodeCache({

stdTTL: 300, // 5 minutos por defecto

checkperiod: 60 // Limpiar cada minuto

});

}

// Cache multi-nivel (Local -> Redis -> Database)

async get<T>(key: string, fetchFn?: () => Promise<T>, ttl = 3600): Promise<T | null> {

// Nivel 1: Cache local (más rápido)

let value = this.localCache.get<T>(key);

if (value !== undefined) {

return value;

}

// Nivel 2: Redis (rápido, distribuido)

const redisValue = await this.redisClient.get(key);

if (redisValue) {

try {

value = JSON.parse(redisValue);

// Guardar en cache local para próximas consultas

this.localCache.set(key, value, Math.min(ttl, 300)); // Max 5 min en local

return value;

} catch (error) {

console.error('Error parsing Redis value:', error);

}

}

// Nivel 3: Fetch desde fuente (base de datos, API, etc.)

if (fetchFn) {

value = await fetchFn();

if (value !== null && value !== undefined) {

// Guardar en ambos niveles

await this.set(key, value, ttl);

}

return value;

}

return null;

}

async set<T>(key: string, value: T, ttl = 3600): Promise<void> {

// Guardar en Redis

await this.redisClient.setex(key, ttl, JSON.stringify(value));

// Guardar en cache local (con TTL menor)

this.localCache.set(key, value, Math.min(ttl, 300));

}

// Cache con compresión para objetos grandes

async setCompressed<T>(key: string, value: T, ttl = 3600): Promise<void> {

const compressed = await this.compress(JSON.stringify(value));

await this.redisClient.setex(`${key}:compressed`, ttl, compressed);

}

async getCompressed<T>(key: string): Promise<T | null> {

const compressed = await this.redisClient.get(`${key}:compressed`);

if (!compressed) return null;

const decompressed = await this.decompress(compressed);

return JSON.parse(decompressed);

}

// Cache con invalidación inteligente

async invalidatePattern(pattern: string): Promise<void> {

const keys = await this.redisClient.keys(pattern);

if (keys.length > 0) {

await this.redisClient.del(...keys);

}

// Invalidar cache local también

this.localCache.flushAll();

}

// Cache warming para datos críticos

async warmCache(): Promise<void> {

const warmingTasks = [

this.warmPropertyCache(),

this.warmUserPreferencesCache(),

this.warmConfigurationCache(),

this.warmPopularDataCache()

];

await Promise.allSettled(warmingTasks);

}

private async warmPropertyCache(): Promise<void> {

// Pre-cargar propiedades más populares

const popularProperties = await prisma.property.findMany({

take: 100,

orderBy: { bookings: { \_count: 'desc' } },

include: {

rooms: true,

owner: { select: { contactName: true, email: true } }

}

});

const promises = popularProperties.map(property =>

this.set(`property:${property.id}`, property, 7200) // 2 horas

);

await Promise.all(promises);

}

// Circuit breaker para cache

async getWithCircuitBreaker<T>(

key: string,

fetchFn: () => Promise<T>,

options: CircuitBreakerOptions = {}

): Promise<T> {

const circuitKey = `circuit:${key}`;

const failureCount = await this.redisClient.get(`${circuitKey}:failures`) || '0';

if (parseInt(failureCount) >= (options.threshold || 5)) {

const lastFailure = await this.redisClient.get(`${circuitKey}:last\_failure`);

const timeSinceFailure = Date.now() - parseInt(lastFailure || '0');

if (timeSinceFailure < (options.timeout || 60000)) {

// Circuit abierto, devolver cache stale si existe

const staleValue = await this.get<T>(`${key}:stale`);

if (staleValue) {

return staleValue;

}

throw new Error('Circuit breaker open and no stale data available');

}

}

try {

const value = await fetchFn();

// Reset circuit breaker en caso de éxito

await this.redisClient.del(`${circuitKey}:failures`, `${circuitKey}:last\_failure`);

// Guardar valor y copia stale

await this.set(key, value);

await this.set(`${key}:stale`, value, 86400); // 24h para emergencias

return value;

} catch (error) {

// Incrementar contador de fallos

await this.redisClient.incr(`${circuitKey}:failures`);

await this.redisClient.set(`${circuitKey}:last\_failure`, Date.now().toString());

throw error;

}

}

}

// Optimización de base de datos

export class DatabaseOptimizationService {

async optimizeQueries(): Promise<OptimizationReport> {

const report: OptimizationReport = {

slowQueries: [],

indexRecommendations: [],

statisticsUpdated: false,

vacuumPerformed: false

};

// Identificar consultas lentas

const slowQueries = await this.identifySlowQueries();

report.slowQueries = slowQueries;

// Recomendar índices

const indexRecommendations = await this.analyzeIndexUsage();

report.indexRecommendations = indexRecommendations;

// Actualizar estadísticas de PostgreSQL

await this.updateTableStatistics();

report.statisticsUpdated = true;

// Vacuum automático para tablas grandes

await this.performMaintenanceTasks();

report.vacuumPerformed = true;

return report;

}

private async identifySlowQueries(): Promise<SlowQuery[]> {

const slowQueries = await prisma.$queryRaw<SlowQuery[]>`

SELECT

query,

calls,

total\_time,

mean\_time,

rows

FROM pg\_stat\_statements

WHERE mean\_time > 100 -- Consultas que toman más de 100ms en promedio

ORDER BY mean\_time DESC

LIMIT 20

`;

return slowQueries;

}

private async analyzeIndexUsage(): Promise<IndexRecommendation[]> {

// Analizar uso de índices existentes

const unusedIndexes = await prisma.$queryRaw<any[]>`

SELECT

schemaname,

tablename,

indexname,

idx\_tup\_read,

idx\_tup\_fetch

FROM pg\_stat\_user\_indexes

WHERE idx\_tup\_read = 0

AND idx\_tup\_fetch = 0

`;

// Buscar consultas que podrían beneficiarse de índices

const missingIndexes = await this.findMissingIndexOpportunities();

return [

...unusedIndexes.map(idx => ({

type: 'drop\_unused',

table: idx.tablename,

index: idx.indexname,

reason: 'Index never used'

})),

...missingIndexes

];

}

// Connection pooling optimizado

async optimizeConnectionPool(): Promise<void> {

const poolConfig = {

min: 5,

max: 20,

acquire: 30000,

idle: 10000,

evict: 5000,

handleDisconnects: true

};

// Monitorear uso del pool

setInterval(async () => {

const stats = await this.getPoolStats();

if (stats.waiting > 0) {

console.warn(`Database pool saturation: ${stats.waiting} connections waiting`);

}

if (stats.idle / stats.total > 0.7) {

console.info('Consider reducing max pool size');

}

}, 30000);

}

}

// Optimización frontend

export class FrontendOptimizationService {

// Code splitting automático

static setupCodeSplitting() {

return {

// Lazy loading de rutas

routes: [

{

path: '/dashboard',

component: lazy(() => import('../pages/Dashboard'))

},

{

path: '/checkin',

component: lazy(() => import('../pages/CheckIn'))

},

{

path: '/profile',

component: lazy(() => import('../pages/Profile'))

}

],

// Preloading de recursos críticos

preloadCriticalRoutes: () => {

const preloadRoutes = ['/dashboard', '/checkin'];

preloadRoutes.forEach(route => {

import(`../pages${route}`);

});

}

};

}

// Service Worker para cache offline

static registerServiceWorker() {

if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/sw.js')

.then(registration => {

console.log('SW registered: ', registration);

// Actualizar cache cuando hay nueva versión

registration.addEventListener('updatefound', () => {

const newWorker = registration.installing;

newWorker?.addEventListener('statechange', () => {

if (newWorker.state === 'installed' && navigator.serviceWorker.controller) {

// Mostrar notificación de actualización disponible

this.showUpdateNotification();

}

});

});

})

.catch(error => {

console.log('SW registration failed: ', error);

});

}

}

// Optimización de imágenes

static optimizeImages() {

// Lazy loading de imágenes

const imageObserver = new IntersectionObserver((entries, observer) => {

entries.forEach(entry => {

if (entry.isIntersecting) {

const img = entry.target as HTMLImageElement;

img.src = img.dataset.src!;

img.classList.remove('lazy');

observer.unobserve(img);

}

});

});

document.querySelectorAll('img[data-src]').forEach(img => {

imageObserver.observe(img);

});

// WebP detection y fallback

const supportsWebP = () => {

const canvas = document.createElement('canvas');

return canvas.toDataURL('image/webp').indexOf('data:image/webp') === 0;

};

if (supportsWebP()) {

document.documentElement.classList.add('webp-support');

}

}

// Bundle size analysis

static analyzeBundleSize() {

if (process.env.NODE\_ENV === 'development') {

import('webpack-bundle-analyzer').then(({ BundleAnalyzerPlugin }) => {

// Configurar analyzer para desarrollo

});

}

}

}

**3. SEGURIDAD AVANZADA**

**Security Service Comprehensive**

// src/security/SecurityService.ts

export class SecurityService {

private rateLimiter: Map<string, RateLimitData> = new Map();

private suspiciousActivityTracker = new Map<string, SuspiciousActivity>();

// Rate limiting avanzado con diferentes niveles

async checkRateLimit(

identifier: string,

action: string,

customLimits?: RateLimitConfig

): Promise<RateLimitResult> {

const limits = customLimits || this.getDefaultLimits(action);

const key = `${identifier}:${action}`;

const now = Date.now();

let data = this.rateLimiter.get(key) || {

requests: [],

blocked: false,

blockedUntil: 0

};

// Limpiar requests antiguos

data.requests = data.requests.filter(timestamp =>

now - timestamp < limits.windowMs

);

// Verificar si está bloqueado

if (data.blocked && now < data.blockedUntil) {

return {

allowed: false,

resetTime: data.blockedUntil,

remaining: 0,

reason: 'Rate limit exceeded'

};

}

// Verificar límite

if (data.requests.length >= limits.max) {

data.blocked = true;

data.blockedUntil = now + (limits.blockDurationMs || 300000); // 5 min default

this.rateLimiter.set(key, data);

// Log de actividad sospechosa

await this.logSuspiciousActivity(identifier, 'rate\_limit\_exceeded', {

action,

requests: data.requests.length,

windowMs: limits.windowMs

});

return {

allowed: false,

resetTime: data.blockedUntil,

remaining: 0,

reason: 'Rate limit exceeded'

};

}

// Permitir request

data.requests.push(now);

data.blocked = false;

this.rateLimiter.set(key, data);

return {

allowed: true,

remaining: limits.max - data.requests.length,

resetTime: now + limits.windowMs

};

}

private getDefaultLimits(action: string): RateLimitConfig {

const limits: Record<string, RateLimitConfig> = {

'login': { max: 5, windowMs: 15 \* 60 \* 1000, blockDurationMs: 15 \* 60 \* 1000 }, // 5 intentos por 15 min

'register': { max: 3, windowMs: 60 \* 60 \* 1000, blockDurationMs: 60 \* 60 \* 1000 }, // 3 registros por hora

'password\_reset': { max: 3, windowMs: 60 \* 60 \* 1000 },

'checkin': { max: 10, windowMs: 60 \* 1000 }, // 10 check-ins por minuto

'upload': { max: 20, windowMs: 60 \* 1000 }, // 20 uploads por minuto

'api\_general': { max: 100, windowMs: 60 \* 1000 } // 100 requests por minuto

};

return limits[action] || limits['api\_general'];

}

// Detección de actividad sospechosa

async detectSuspiciousActivity(userId: string, request: any): Promise<void> {

const indicators = [

this.checkUnusualLoginTimes(userId, request),

this.checkGeolocationAnomalies(userId, request),

this.checkDeviceFingerprinting(userId, request),

this.checkBehaviorPatterns(userId, request),

this.checkMultipleAccountAttempts(request.ip)

];

const suspiciousSignals = (await Promise.allSettled(indicators))

.filter(result => result.status === 'fulfilled' && result.value)

.map(result => (result as any).value);

if (suspiciousSignals.length >= 2) {

await this.handleSuspiciousActivity(userId, suspiciousSignals, request);

}

}

private async checkGeolocationAnomalies(userId: string, request: any): Promise<SuspiciousSignal | null> {

const currentLocation = await this.getLocationFromIP(request.ip);

const recentLocations = await this.getRecentUserLocations(userId, 24 \* 60 \* 60 \* 1000); // 24h

if (recentLocations.length === 0) {

return null; // Primera vez, no podemos comparar

}

const lastLocation = recentLocations[0];

const distance = this.calculateDistance(currentLocation, lastLocation.location);

const timeDiff = Date.now() - lastLocation.timestamp;

// Velocidad imposible (>1000 km/h)

const maxSpeed = 1000; // km/h

const requiredTime = (distance / maxSpeed) \* 60 \* 60 \* 1000; // ms

if (distance > 500 && timeDiff < requiredTime) {

return {

type: 'impossible\_travel',

score: 0.8,

details: {

distance,

timeDiff,

locations: [lastLocation.location, currentLocation]

}

};

}

return null;

}

private async checkDeviceFingerprinting(userId: string, request: any): Promise<SuspiciousSignal | null> {

const currentFingerprint = this.generateDeviceFingerprint(request);

const knownDevices = await this.getUserKnownDevices(userId);

const isKnownDevice = knownDevices.some(device =>

this.compareFingerprints(device.fingerprint, currentFingerprint) > 0.8

);

if (!isKnownDevice && knownDevices.length > 0) {

return {

type: 'unknown\_device',

score: 0.6,

details: { fingerprint: currentFingerprint }

};

}

return null;

}

// Input validation y sanitización

validateAndSanitizeInput(input: any, schema: ValidationSchema): ValidationResult {

const errors: ValidationError[] = [];

const sanitized: any = {};

for (const [field, rules] of Object.entries(schema)) {

const value = input[field];

// Verificar si es requerido

if (rules.required && (value === undefined || value === null || value === '')) {

errors.push({

field,

message: `${field} is required`,

code: 'REQUIRED'

});

continue;

}

if (value === undefined || value === null) {

sanitized[field] = rules.default;

continue;

}

// Validación de tipo

if (rules.type && typeof value !== rules.type) {

errors.push({

field,

message: `${field} must be of type ${rules.type}`,

code: 'INVALID\_TYPE'

});

continue;

}

let sanitizedValue = value;

// Sanitización específica por tipo

switch (rules.type) {

case 'string':

sanitizedValue = this.sanitizeString(value, rules.stringOptions);

break;

case 'number':

sanitizedValue = this.sanitizeNumber(value, rules.numberOptions);

break;

case 'email':

sanitizedValue = this.sanitizeEmail(value);

break;

case 'phone':

sanitizedValue = this.sanitizePhone(value);

break;

}

// Validación personalizada

if (rules.validator) {

const validationResult = rules.validator(sanitizedValue);

if (!validationResult.isValid) {

errors.push({

field,

message: validationResult.message,

code: 'CUSTOM\_VALIDATION'

});

continue;

}

}

sanitized[field] = sanitizedValue;

}

return {

isValid: errors.length === 0,

errors,

sanitized

};

}

private sanitizeString(value: string, options?: StringSanitizationOptions): string {

let sanitized = value.toString().trim();

if (options?.removeHTML) {

sanitized = sanitized.replace(/<[^>]\*>/g, '');

}

if (options?.escapeHTML) {

sanitized = sanitized

.replace(/&/g, '&amp;')

.replace(/</g, '&lt;')

.replace(/>/g, '&gt;')

.replace(/"/g, '&quot;')

.replace(/'/g, '&#x27;');

}

if (options?.maxLength) {

sanitized = sanitized.substring(0, options.maxLength);

}

if (options?.removeSpecialChars) {

sanitized = sanitized.replace(/[^a-zA-Z0-9\s]/g, '');

}

return sanitized;

}

// Encriptación de datos sensibles

async encryptSensitiveData(data: string, keyId = 'default'): Promise<string> {

const key = await this.getEncryptionKey(keyId);

const iv = crypto.randomBytes(16);

const cipher = crypto.createCipherGCM('aes-256-gcm', key);

let encrypted = cipher.update(data, 'utf8', 'hex');

encrypted += cipher.final('hex');

const authTag = cipher.getAuthTag();

return `${keyId}:${iv.toString('hex')}:${authTag.toString('hex')}:${encrypted}`;

}

async decryptSensitiveData(encryptedData: string): Promise<string> {

const [keyId, ivHex, authTagHex, encrypted] = encryptedData.split(':');

const key = await this.getEncryptionKey(keyId);

const iv = Buffer.from(ivHex, 'hex');

const authTag = Buffer.from(authTagHex, 'hex');

const decipher = crypto.createDecipherGCM('aes-256-gcm', key);

decipher.setAuthTag(authTag);

let decrypted = decipher.update(encrypted, 'hex', 'utf8');

decrypted += decipher.final('utf8');

return decrypted;

}

// Audit logging de seguridad

async logSecurityEvent(event: SecurityEvent): Promise<void> {

const logEntry = {

timestamp: new Date(),

type: event.type,

severity: event.severity,

userId: event.userId,

ip: event.ip,

userAgent: event.userAgent,

details: event.details,

id: crypto.randomUUID()

};

// Log local

console.log(`[SECURITY] ${event.type}:`, logEntry);

// Enviar a SIEM si es crítico

if (event.severity === 'critical' || event.severity === 'high') {

await this.sendToSIEM(logEntry);

}

// Guardar en base de datos

await prisma.securityEvent.create({

data: logEntry

});

// Alertas automáticas

await this.processSecurityAlert(logEntry);

}

// Compliance y auditoría

async generateComplianceReport(startDate: Date, endDate: Date): Promise<ComplianceReport> {

const [

userAccesses,

dataModifications,

securityEvents,

dataRetentionStatus

] = await Promise.all([

this.getAccessLogs(startDate, endDate),

this.getDataModificationLogs(startDate, endDate),

this.getSecurityEvents(startDate, endDate),

this.checkDataRetentionCompliance()

]);

return {

period: { start: startDate, end: endDate },

userAccesses: {

total: userAccesses.length,

unique\_users: new Set(userAccesses.map(a => a.userId)).size,

failed\_attempts: userAccesses.filter(a => !a.success).length

},

dataModifications: {

total: dataModifications.length,

by\_type: this.groupBy(dataModifications, 'type'),

unauthorized\_attempts: dataModifications.filter(m => m.unauthorized).length

},

securityIncidents: {

total: securityEvents.length,

by\_severity: this.groupBy(securityEvents, 'severity'),

resolved: securityEvents.filter(e => e.resolved).length

},

dataRetention: dataRetentionStatus,

gdprCompliance: await this.checkGDPRCompliance(),

recommendations: await this.generateSecurityRecommendations(securityEvents)

};

}

}

// Middleware de seguridad para Express

export const securityMiddleware = {

// Rate limiting

rateLimitMiddleware: (action: string) => {

return async (req: Request, res: Response, next: NextFunction) => {

const identifier = req.ip + (req.user?.id || '');

const result = await securityService.checkRateLimit(identifier, action);

if (!result.allowed) {

return res.status(429).json({

error: 'Rate limit exceeded',

resetTime: result.resetTime,

reason: result.reason

});

}

// Headers informativos

res.setHeader('X-RateLimit-Remaining', result.remaining);

res.setHeader('X-RateLimit-Reset', result.resetTime);

next();

};

},

// Validación de input

validateInput: (schema: ValidationSchema) => {

return (req: Request, res: Response, next: NextFunction) => {

const validation = securityService.validateAndSanitizeInput(req.body, schema);

if (!validation.isValid) {

return res.status(400).json({

error: 'Validation failed',

details: validation.errors

});

}

req.body = validation.sanitized;

next();

};

},

// Detección de actividad sospechosa

suspiciousActivityDetection: async (req: Request, res: Response, next: NextFunction) => {

if (req.user) {

try {

await securityService.detectSuspiciousActivity(req.user.id, req);

} catch (error) {

console.error('Suspicious activity detection failed:', error);

}

}

next();

}

};