// ===================================================================

// 🚀 KURZORA TRADING PLATFORM - COMPLETE INTEGRATION & DEPLOYMENT SYSTEM

// ===================================================================

// \*\*1. 🔔 ALERT & NOTIFICATION SYSTEMS\*\*

// ===================================================================

// MAKE.COM WORKFLOW AUTOMATION

// ===================================================================

// lib/notifications/makeWebhooks.ts

interface MakeWebhookPayload {

type: 'signal\_alert' | 'trade\_execution' | 'risk\_alert' | 'system\_notification';

data: {

userId: string;

ticker?: string;

message: string;

priority: 'low' | 'medium' | 'high' | 'critical';

channels: ('telegram' | 'email' | 'sms' | 'push')[];

metadata?: Record<string, any>;

};

timestamp: string;

}

class MakeWorkflowTrigger {

private webhookUrls = {

signal\_alerts: process.env.MAKE\_SIGNAL\_WEBHOOK\_URL!,

trade\_execution: process.env.MAKE\_TRADE\_WEBHOOK\_URL!,

risk\_management: process.env.MAKE\_RISK\_WEBHOOK\_URL!,

system\_notifications: process.env.MAKE\_SYSTEM\_WEBHOOK\_URL!

};

async triggerSignalAlert(signal: SignalResult, userPreferences: NotificationPreferences[]): Promise<void> {

const eligibleUsers = userPreferences.filter(pref =>

pref.signalNotifications &&

signal.finalScore >= pref.minSignalScore

);

for (const user of eligibleUsers) {

const payload: MakeWebhookPayload = {

type: 'signal\_alert',

data: {

userId: user.userId,

ticker: signal.ticker,

message: this.formatSignalMessage(signal),

priority: signal.strength === 'strong' ? 'high' : 'medium',

channels: user.enabledChannels,

metadata: {

signalScore: signal.finalScore,

signalType: signal.signalType,

entryPrice: signal.riskReward.entryPrice,

stopLoss: signal.riskReward.stopLoss,

takeProfit: signal.riskReward.takeProfit,

riskRewardRatio: signal.riskReward.riskRewardRatio

}

},

timestamp: new Date().toISOString()

};

await this.sendWebhook(this.webhookUrls.signal\_alerts, payload);

}

}

async triggerTradeExecution(trade: PaperTrade, userId: string): Promise<void> {

const payload: MakeWebhookPayload = {

type: 'trade\_execution',

data: {

userId,

ticker: trade.ticker,

message: `Trade executed: ${trade.tradeType.toUpperCase()} ${trade.shares} shares of ${trade.ticker} at $${trade.entryPrice}`,

priority: 'medium',

channels: ['telegram', 'email'],

metadata: {

tradeId: trade.id,

tradeType: trade.tradeType,

shares: trade.shares,

entryPrice: trade.entryPrice,

positionValue: trade.positionValue

}

},

timestamp: new Date().toISOString()

};

await this.sendWebhook(this.webhookUrls.trade\_execution, payload);

}

async triggerRiskAlert(alert: RiskAlert, userId: string): Promise<void> {

const payload: MakeWebhookPayload = {

type: 'risk\_alert',

data: {

userId,

message: `⚠️ Risk Alert: ${alert.message}`,

priority: alert.severity === 'high' ? 'critical' : 'high',

channels: ['telegram', 'email', 'push'],

metadata: {

alertType: alert.type,

severity: alert.severity,

recommendation: alert.recommendation

}

},

timestamp: new Date().toISOString()

};

await this.sendWebhook(this.webhookUrls.risk\_management, payload);

}

private async sendWebhook(url: string, payload: MakeWebhookPayload): Promise<void> {

try {

const response = await fetch(url, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'Authorization': `Bearer ${process.env.MAKE\_API\_KEY}`

},

body: JSON.stringify(payload)

});

if (!response.ok) {

throw new Error(`Make.com webhook failed: ${response.status} ${response.statusText}`);

}

console.log(`Make.com webhook sent successfully: ${payload.type}`);

} catch (error) {

console.error('Failed to send Make.com webhook:', error);

// Fallback to direct notification

await this.sendFallbackNotification(payload);

}

}

private formatSignalMessage(signal: SignalResult): string {

const emoji = signal.strength === 'strong' ? '💎' : signal.strength === 'valid' ? '✅' : '⚠️';

return `${emoji} ${signal.ticker} Signal (${signal.finalScore}/100)

📈 Type: ${signal.signalType.toUpperCase()}

💰 Entry: $${signal.riskReward.entryPrice}

🛡️ Stop Loss: $${signal.riskReward.stopLoss}

🎯 Take Profit: $${signal.riskReward.takeProfit}

📊 R/R: ${signal.riskReward.riskRewardRatio}:1`;

}

}

// ===================================================================

// TELEGRAM BOT INTEGRATION

// ===================================================================

// lib/notifications/telegramBot.ts

interface TelegramUser {

chatId: string;

userId: string;

username?: string;

firstName?: string;

isActive: boolean;

preferences: {

signals: boolean;

trades: boolean;

risks: boolean;

dailyDigest: boolean;

};

}

class TelegramBotService {

private botToken = process.env.TELEGRAM\_BOT\_TOKEN!;

private baseUrl = `https://api.telegram.org/bot${this.botToken}`;

private userMappings = new Map<string, TelegramUser>();

constructor() {

this.initializeBot();

}

async initializeBot(): Promise<void> {

try {

// Set webhook for receiving messages

await this.setWebhook();

// Load existing user mappings

await this.loadUserMappings();

// Set bot commands

await this.setBotCommands();

console.log('Telegram bot initialized successfully');

} catch (error) {

console.error('Failed to initialize Telegram bot:', error);

}

}

async sendSignalAlert(chatId: string, signal: SignalResult): Promise<boolean> {

const message = this.formatSignalMessage(signal);

const keyboard = this.createSignalKeyboard(signal);

return await this.sendMessage(chatId, message, {

parse\_mode: 'HTML',

reply\_markup: keyboard

});

}

async sendTradeNotification(chatId: string, trade: PaperTrade): Promise<boolean> {

const message = this.formatTradeMessage(trade);

return await this.sendMessage(chatId, message, { parse\_mode: 'HTML' });

}

async sendDailyDigest(chatId: string, digest: DailyDigest): Promise<boolean> {

const message = this.formatDailyDigest(digest);

return await this.sendMessage(chatId, message, { parse\_mode: 'HTML' });

}

// Handle incoming webhook messages

async handleWebhook(update: any): Promise<void> {

try {

if (update.message) {

await this.handleMessage(update.message);

} else if (update.callback\_query) {

await this.handleCallbackQuery(update.callback\_query);

}

} catch (error) {

console.error('Error handling Telegram webhook:', error);

}

}

private async handleMessage(message: any): Promise<void> {

const chatId = message.chat.id.toString();

const text = message.text;

const userId = message.from.id.toString();

// Handle commands

if (text.startsWith('/start')) {

await this.handleStartCommand(chatId, message.from);

} else if (text.startsWith('/subscribe')) {

await this.handleSubscribeCommand(chatId, userId);

} else if (text.startsWith('/unsubscribe')) {

await this.handleUnsubscribeCommand(chatId, userId);

} else if (text.startsWith('/status')) {

await this.handleStatusCommand(chatId, userId);

} else if (text.startsWith('/help')) {

await this.handleHelpCommand(chatId);

}

}

private async handleCallbackQuery(callbackQuery: any): Promise<void> {

const chatId = callbackQuery.message.chat.id.toString();

const data = callbackQuery.data;

const messageId = callbackQuery.message.message\_id;

// Parse callback data

const [action, ...params] = data.split(':');

switch (action) {

case 'paper\_trade':

await this.handlePaperTradeCallback(chatId, params, messageId);

break;

case 'view\_chart':

await this.handleViewChartCallback(chatId, params);

break;

case 'set\_alert':

await this.handleSetAlertCallback(chatId, params);

break;

}

// Answer callback query

await this.answerCallbackQuery(callbackQuery.id);

}

private async sendMessage(chatId: string, text: string, options: any = {}): Promise<boolean> {

try {

const response = await fetch(`${this.baseUrl}/sendMessage`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

chat\_id: chatId,

text,

...options

})

});

return response.ok;

} catch (error) {

console.error('Failed to send Telegram message:', error);

return false;

}

}

private formatSignalMessage(signal: SignalResult): string {

const emoji = signal.strength === 'strong' ? '💎' : '✅';

const typeEmoji = signal.signalType === 'bullish' ? '🚀' : '📉';

return `${emoji} <b>${signal.ticker} Trading Signal</b>

${typeEmoji} <b>Type:</b> ${signal.signalType.toUpperCase()}

📊 <b>Score:</b> ${signal.finalScore}/100

⭐ <b>Strength:</b> ${signal.strength.toUpperCase()}

💰 <b>Entry Price:</b> $${signal.riskReward.entryPrice}

🛡️ <b>Stop Loss:</b> $${signal.riskReward.stopLoss}

🎯 <b>Take Profit:</b> $${signal.riskReward.takeProfit}

📈 <b>Risk/Reward:</b> ${signal.riskReward.riskRewardRatio}:1

🔍 <b>Confidence:</b> ${signal.confluence.overallConfidence}%

⏰ <b>Generated:</b> ${new Date().toLocaleTimeString()}`;

}

private createSignalKeyboard(signal: SignalResult): any {

return {

inline\_keyboard: [

[

{

text: '📊 View Chart',

callback\_data: `view\_chart:${signal.ticker}`

},

{

text: '💰 Paper Trade',

callback\_data: `paper\_trade:${signal.ticker}:${signal.riskReward.entryPrice}`

}

],

[

{

text: '🔔 Set Alert',

callback\_data: `set\_alert:${signal.ticker}:${signal.finalScore}`

}

]

]

};

}

private async setWebhook(): Promise<void> {

const webhookUrl = `${process.env.VERCEL\_URL}/api/telegram/webhook`;

await fetch(`${this.baseUrl}/setWebhook`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

url: webhookUrl,

secret\_token: process.env.TELEGRAM\_WEBHOOK\_SECRET

})

});

}

private async setBotCommands(): Promise<void> {

const commands = [

{ command: 'start', description: 'Start using Kurzora Bot' },

{ command: 'subscribe', description: 'Subscribe to trading signals' },

{ command: 'unsubscribe', description: 'Unsubscribe from notifications' },

{ command: 'status', description: 'Check your subscription status' },

{ command: 'help', description: 'Show help information' }

];

await fetch(`${this.baseUrl}/setMyCommands`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ commands })

});

}

}

// ===================================================================

// EMAIL NOTIFICATION SYSTEM (SENDGRID)

// ===================================================================

// lib/notifications/emailService.ts

import sgMail from '@sendgrid/mail';

interface EmailTemplate {

templateId: string;

subject: string;

variables: Record<string, any>;

}

class EmailNotificationService {

constructor() {

sgMail.setApiKey(process.env.SENDGRID\_API\_KEY!);

}

async sendSignalAlert(userEmail: string, signal: SignalResult): Promise<boolean> {

const template: EmailTemplate = {

templateId: 'd-signal-alert-template-id',

subject: `🚀 ${signal.ticker} Trading Signal - Score: ${signal.finalScore}`,

variables: {

ticker: signal.ticker,

signalType: signal.signalType,

finalScore: signal.finalScore,

strength: signal.strength,

entryPrice: signal.riskReward.entryPrice,

stopLoss: signal.riskReward.stopLoss,

takeProfit: signal.riskReward.takeProfit,

riskRewardRatio: signal.riskReward.riskRewardRatio,

confidence: signal.confluence.overallConfidence,

timestamp: new Date().toISOString(),

chartUrl: `https://kurzora.com/chart/${signal.ticker}`,

dashboardUrl: 'https://kurzora.com/dashboard'

}

};

return await this.sendTemplatedEmail(userEmail, template);

}

async sendTradeExecuted(userEmail: string, trade: PaperTrade): Promise<boolean> {

const template: EmailTemplate = {

templateId: 'd-trade-executed-template-id',

subject: `✅ Trade Executed: ${trade.ticker}`,

variables: {

ticker: trade.ticker,

tradeType: trade.tradeType,

shares: trade.shares,

entryPrice: trade.entryPrice,

positionValue: trade.positionValue,

stopLoss: trade.stopLoss,

takeProfit: trade.takeProfit,

timestamp: trade.entryTime.toISOString(),

portfolioUrl: 'https://kurzora.com/portfolio'

}

};

return await this.sendTemplatedEmail(userEmail, template);

}

async sendDailyDigest(userEmail: string, digest: DailyDigest): Promise<boolean> {

const template: EmailTemplate = {

templateId: 'd-daily-digest-template-id',

subject: `📊 Your Daily Trading Digest - ${new Date().toLocaleDateString()}`,

variables: {

date: new Date().toLocaleDateString(),

topSignals: digest.topSignals.slice(0, 5),

portfolioPerformance: digest.portfolioPerformance,

marketSummary: digest.marketSummary,

recommendedActions: digest.recommendedActions,

dashboardUrl: 'https://kurzora.com/dashboard'

}

};

return await this.sendTemplatedEmail(userEmail, template);

}

async sendWelcomeEmail(userEmail: string, userName: string): Promise<boolean> {

const template: EmailTemplate = {

templateId: 'd-welcome-template-id',

subject: '🎉 Welcome to Kurzora Trading Platform!',

variables: {

userName,

dashboardUrl: 'https://kurzora.com/dashboard',

telegramBotUrl: 'https://t.me/kurzora\_trading\_bot',

supportEmail: 'support@kurzora.com'

}

};

return await this.sendTemplatedEmail(userEmail, template);

}

private async sendTemplatedEmail(to: string, template: EmailTemplate): Promise<boolean> {

try {

const msg = {

to,

from: {

email: 'noreply@kurzora.com',

name: 'Kurzora Trading'

},

templateId: template.templateId,

dynamicTemplateData: template.variables

};

await sgMail.send(msg);

console.log(`Email sent successfully to ${to}`);

return true;

} catch (error) {

console.error('Failed to send email:', error);

return false;

}

}

async sendBulkEmails(recipients: string[], template: EmailTemplate): Promise<void> {

try {

const personalizations = recipients.map(email => ({

to: [{ email }],

dynamicTemplateData: template.variables

}));

const msg = {

from: {

email: 'noreply@kurzora.com',

name: 'Kurzora Trading'

},

templateId: template.templateId,

personalizations

};

await sgMail.sendMultiple(msg);

console.log(`Bulk email sent to ${recipients.length} recipients`);

} catch (error) {

console.error('Failed to send bulk email:', error);

}

}

}

// ===================================================================

// PUSH NOTIFICATION INFRASTRUCTURE

// ===================================================================

// lib/notifications/pushNotifications.ts

interface PushSubscription {

userId: string;

endpoint: string;

keys: {

p256dh: string;

auth: string;

};

userAgent?: string;

isActive: boolean;

}

class PushNotificationService {

private vapidKeys = {

publicKey: process.env.VAPID\_PUBLIC\_KEY!,

privateKey: process.env.VAPID\_PRIVATE\_KEY!,

subject: 'mailto:support@kurzora.com'

};

constructor() {

const webPush = require('web-push');

webPush.setVapidDetails(

this.vapidKeys.subject,

this.vapidKeys.publicKey,

this.vapidKeys.privateKey

);

}

async subscribe(userId: string, subscription: any): Promise<boolean> {

try {

const pushSubscription: PushSubscription = {

userId,

endpoint: subscription.endpoint,

keys: subscription.keys,

userAgent: subscription.userAgent,

isActive: true

};

// Store subscription in database

const { error } = await supabaseClient

.from('push\_subscriptions')

.upsert(pushSubscription, { onConflict: 'user\_id,endpoint' });

if (error) throw error;

console.log(`Push subscription registered for user ${userId}`);

return true;

} catch (error) {

console.error('Failed to register push subscription:', error);

return false;

}

}

async sendNotification(userId: string, notification: PushNotificationPayload): Promise<boolean> {

try {

// Get user's push subscriptions

const { data: subscriptions, error } = await supabaseClient

.from('push\_subscriptions')

.select('\*')

.eq('user\_id', userId)

.eq('is\_active', true);

if (error || !subscriptions?.length) {

console.log(`No active push subscriptions for user ${userId}`);

return false;

}

const webPush = require('web-push');

const payload = JSON.stringify(notification);

// Send to all user's devices

const promises = subscriptions.map(async (sub) => {

try {

await webPush.sendNotification({

endpoint: sub.endpoint,

keys: sub.keys

}, payload);

return true;

} catch (error) {

console.error(`Failed to send push notification to ${sub.endpoint}:`, error);

// Mark subscription as inactive if it's invalid

if (error.statusCode === 410) {

await this.deactivateSubscription(sub.endpoint);

}

return false;

}

});

const results = await Promise.allSettled(promises);

const successCount = results.filter(r => r.status === 'fulfilled' && r.value).length;

console.log(`Push notifications sent: ${successCount}/${subscriptions.length}`);

return successCount > 0;

} catch (error) {

console.error('Failed to send push notification:', error);

return false;

}

}

async sendBulkNotifications(userIds: string[], notification: PushNotificationPayload): Promise<void> {

const promises = userIds.map(userId => this.sendNotification(userId, notification));

await Promise.allSettled(promises);

}

private async deactivateSubscription(endpoint: string): Promise<void> {

await supabaseClient

.from('push\_subscriptions')

.update({ is\_active: false })

.eq('endpoint', endpoint);

}

}

// ===================================================================

// \*\*2. 🔗 THIRD-PARTY SERVICE INTEGRATION\*\*

// ===================================================================

// ===================================================================

// POLYGON.IO MARKET DATA SETUP

// ===================================================================

// lib/integrations/polygonSetup.ts

class PolygonIntegrationSetup {

private config = {

apiKey: process.env.POLYGON\_API\_KEY!,

baseUrl: 'https://api.polygon.io',

websocketUrl: 'wss://socket.polygon.io',

maxRequestsPerMinute: 5, // Free tier limit

retryAttempts: 3,

retryDelay: 1000

};

async validateConnection(): Promise<boolean> {

try {

const response = await fetch(`${this.config.baseUrl}/v3/reference/tickers?market=stocks&active=true&limit=1&apikey=${this.config.apiKey}`);

if (response.ok) {

console.log('✅ Polygon.io connection validated');

return true;

} else {

console.error('❌ Polygon.io connection failed:', response.status, response.statusText);

return false;

}

} catch (error) {

console.error('❌ Polygon.io connection error:', error);

return false;

}

}

async setupWebSocketConnection(): Promise<WebSocket | null> {

try {

const ws = new WebSocket(`${this.config.websocketUrl}/stocks`);

ws.onopen = () => {

console.log('🔌 Polygon WebSocket connected');

// Authenticate

ws.send(JSON.stringify({

action: 'auth',

params: this.config.apiKey

}));

};

ws.onmessage = (event) => {

const data = JSON.parse(event.data);

this.handleWebSocketMessage(data);

};

ws.onerror = (error) => {

console.error('Polygon WebSocket error:', error);

};

ws.onclose = () => {

console.log('Polygon WebSocket disconnected');

// Implement reconnection logic

setTimeout(() => this.setupWebSocketConnection(), 5000);

};

return ws;

} catch (error) {

console.error('Failed to setup Polygon WebSocket:', error);

return null;

}

}

private handleWebSocketMessage(data: any): void {

if (Array.isArray(data)) {

data.forEach(message => this.processMessage(message));

} else {

this.processMessage(data);

}

}

private processMessage(message: any): void {

switch (message.ev) {

case 'status':

console.log('Polygon status:', message.message);

break;

case 'T': // Trade

this.handleTradeMessage(message);

break;

case 'Q': // Quote

this.handleQuoteMessage(message);

break;

case 'A': // Aggregate (minute bar)

this.handleAggregateMessage(message);

break;

}

}

private async handleTradeMessage(trade: any): Promise<void> {

// Store real-time trade data

const tradeData = {

ticker: trade.sym,

price: trade.p,

volume: trade.s,

timestamp: new Date(trade.t),

conditions: trade.c

};

// Trigger real-time updates

await this.broadcastTradeUpdate(tradeData);

}

}

// ===================================================================

// TRADINGVIEW WIDGET INTEGRATION

// ===================================================================

// components/TradingViewChart.tsx

interface TradingViewConfig {

symbol: string;

interval: string;

theme: 'light' | 'dark';

style: string;

locale: string;

toolbar\_bg: string;

enable\_publishing: boolean;

container\_id: string;

}

class TradingViewIntegration {

private static defaultConfig: Partial<TradingViewConfig> = {

interval: '1H',

theme: 'dark',

style: '1',

locale: 'en',

toolbar\_bg: '#1e293b',

enable\_publishing: false

};

static createChart(containerId: string, symbol: string, customConfig: Partial<TradingViewConfig> = {}): void {

const config = {

...this.defaultConfig,

...customConfig,

symbol: `NASDAQ:${symbol}`,

container\_id: containerId

};

// Load TradingView script

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

script.src = 'https://s3.tradingview.com/external-embedding/embed-widget-advanced-chart.js';

script.type = 'text/javascript';

script.async = true;

script.innerHTML = JSON.stringify({

autosize: true,

...config

});

const container = document.getElementById(containerId);

if (container) {

container.appendChild(script);

}

}

static createMiniChart(containerId: string, symbol: string): void {

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

script.src = 'https://s3.tradingview.com/external-embedding/embed-widget-mini-symbol-overview.js';

script.async = true;

script.innerHTML = JSON.stringify({

symbol: `NASDAQ:${symbol}`,

width: '100%',

height: '400',

locale: 'en',

dateRange: '12M',

colorTheme: 'dark',

trendLineColor: 'rgba(41, 98, 255, 1)',

underLineColor: 'rgba(41, 98, 255, 0.3)',

underLineBottomColor: 'rgba(41, 98, 255, 0)',

isTransparent: false,

autosize: true,

largeChartUrl: ''

});

const container = document.getElementById(containerId);

if (container) {

container.appendChild(script);

}

}

}

// React Component for TradingView Chart

const TradingViewChart: React.FC<{

symbol: string;

interval?: string;

height?: number;

onReady?: () => void;

}> = ({ symbol, interval = '1H', height = 500, onReady }) => {

const chartId = `tradingview\_${symbol}\_${Date.now()}`;

useEffect(() => {

TradingViewIntegration.createChart(chartId, symbol, {

interval,

theme: 'dark'

});

if (onReady) {

onReady();

}

return () => {

const container = document.getElementById(chartId);

if (container) {

container.innerHTML = '';

}

};

}, [symbol, interval]);

return (

<div

id={chartId}

style={{ height: `${height}px` }}

className="w-full rounded-lg overflow-hidden"

/>

);

};

// ===================================================================

// STRIPE PAYMENT PROCESSING

// ===================================================================

// lib/integrations/stripeSetup.ts

import Stripe from 'stripe';

interface SubscriptionPlan {

id: string;

name: string;

price: number;

interval: 'month' | 'year';

features: string[];

stripePriceId: string;

}

class StripePaymentService {

private stripe: Stripe;

private plans: SubscriptionPlan[] = [

{

id: 'basic',

name: 'Basic Trader',

price: 29,

interval: 'month',

features: ['Basic signals', 'Email alerts', 'Web dashboard'],

stripePriceId: process.env.STRIPE\_BASIC\_PRICE\_ID!

},

{

id: 'pro',

name: 'Pro Trader',

price: 79,

interval: 'month',

features: ['All signals', 'Telegram alerts', 'Advanced analytics', 'Mobile app'],

stripePriceId: process.env.STRIPE\_PRO\_PRICE\_ID!

},

{

id: 'elite',

name: 'Elite Trader',

price: 199,

interval: 'month',

features: ['All features', 'SMS alerts', 'API access', 'Priority support'],

stripePriceId: process.env.STRIPE\_ELITE\_PRICE\_ID!

}

];

constructor() {

this.stripe = new Stripe(process.env.STRIPE\_SECRET\_KEY!, {

apiVersion: '2023-10-16'

});

}

async createCheckoutSession(

userId: string,

planId: string,

successUrl: string,

cancelUrl: string

): Promise<string> {

try {

const plan = this.plans.find(p => p.id === planId);

if (!plan) throw new Error('Invalid plan ID');

const session = await this.stripe.checkout.sessions.create({

customer\_email: await this.getUserEmail(userId),

line\_items: [

{

price: plan.stripePriceId,

quantity: 1

}

],

mode: 'subscription',

success\_url: `${successUrl}?session\_id={CHECKOUT\_SESSION\_ID}`,

cancel\_url: cancelUrl,

metadata: {

userId,

planId

},

subscription\_data: {

metadata: {

userId,

planId

}

}

});

return session.url!;

} catch (error) {

console.error('Failed to create checkout session:', error);

throw error;

}

}

async handleWebhook(payload: string, signature: string): Promise<void> {

try {

const event = this.stripe.webhooks.constructEvent(

payload,

signature,

process.env.STRIPE\_WEBHOOK\_SECRET!

);

switch (event.type) {

case 'checkout.session.completed':

await this.handleCheckoutCompleted(event.data.object as Stripe.Checkout.Session);

break;

case 'invoice.payment\_succeeded':

await this.handlePaymentSucceeded(event.data.object as Stripe.Invoice);

break;

case 'invoice.payment\_failed':

await this.handlePaymentFailed(event.data.object as Stripe.Invoice);

break;

case 'customer.subscription.deleted':

await this.handleSubscriptionCancelled(event.data.object as Stripe.Subscription);

break;

}

} catch (error) {

console.error('Stripe webhook error:', error);

throw error;

}

}

private async handleCheckoutCompleted(session: Stripe.Checkout.Session): Promise<void> {

const { userId, planId } = session.metadata!;

// Update user subscription in database

await supabaseClient

.from('subscriptions')

.upsert({

user\_id: userId,

stripe\_subscription\_id: session.subscription as string,

stripe\_customer\_id: session.customer as string,

plan\_id: planId,

status: 'active',

current\_period\_start: new Date(),

current\_period\_end: new Date(Date.now() + 30 \* 24 \* 60 \* 60 \* 1000) // 30 days

});

// Send welcome email

const emailService = new EmailNotificationService();

const userEmail = await this.getUserEmail(userId);

await emailService.sendWelcomeEmail(userEmail, 'Trader');

console.log(`Subscription activated for user ${userId}, plan ${planId}`);

}

async createCustomerPortalSession(customerId: string, returnUrl: string): Promise<string> {

const session = await this.stripe.billingPortal.sessions.create({

customer: customerId,

return\_url: returnUrl

});

return session.url;

}

private async getUserEmail(userId: string): Promise<string> {

const { data } = await supabaseClient

.from('users')

.select('email')

.eq('id', userId)

.single();

return data?.email || '';

}

}

// ===================================================================

// \*\*3. 🔄 REAL-TIME COMMUNICATION\*\*

// ===================================================================

// ===================================================================

// WEBSOCKET IMPLEMENTATION

// ===================================================================

// lib/realtime/websocketServer.ts

interface WebSocketClient {

id: string;

userId?: string;

socket: WebSocket;

subscriptions: Set<string>;

lastPing: number;

isAlive: boolean;

}

interface WebSocketMessage {

type: 'subscribe' | 'unsubscribe' | 'ping' | 'pong' | 'signal\_update' | 'portfolio\_update';

data?: any;

timestamp: number;

}

class WebSocketManager {

private clients = new Map<string, WebSocketClient>();

private channels = new Map<string, Set<string>>(); // channel -> client IDs

private heartbeatInterval: NodeJS.Timeout;

constructor() {

this.heartbeatInterval = setInterval(() => this.performHeartbeat(), 30000);

}

addClient(socket: WebSocket, userId?: string): string {

const clientId = this.generateClientId();

const client: WebSocketClient = {

id: clientId,

userId,

socket,

subscriptions: new Set(),

lastPing: Date.now(),

isAlive: true

};

this.clients.set(clientId, client);

this.setupSocketHandlers(client);

console.log(`WebSocket client connected: ${clientId} (user: ${userId})`);

// Send connection confirmation

this.sendToClient(clientId, {

type: 'connection\_confirmed',

data: { clientId, userId },

timestamp: Date.now()

});

return clientId;

}

removeClient(clientId: string): void {

const client = this.clients.get(clientId);

if (!client) return;

// Unsubscribe from all channels

client.subscriptions.forEach(channel => {

this.unsubscribeFromChannel(clientId, channel);

});

this.clients.delete(clientId);

console.log(`WebSocket client disconnected: ${clientId}`);

}

subscribeToChannel(clientId: string, channel: string): boolean {

const client = this.clients.get(clientId);

if (!client) return false;

client.subscriptions.add(channel);

if (!this.channels.has(channel)) {

this.channels.set(channel, new Set());

}

this.channels.get(channel)!.add(clientId);

console.log(`Client ${clientId} subscribed to ${channel}`);

return true;

}

unsubscribeFromChannel(clientId: string, channel: string): boolean {

const client = this.clients.get(clientId);

if (!client) return false;

client.subscriptions.delete(channel);

this.channels.get(channel)?.delete(clientId);

console.log(`Client ${clientId} unsubscribed from ${channel}`);

return true;

}

broadcastToChannel(channel: string, message: WebSocketMessage): void {

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

if (!subscribers) return;

subscribers.forEach(clientId => {

this.sendToClient(clientId, message);

});

console.log(`Broadcasted to ${subscribers.size} clients on channel ${channel}`);

}

sendToUser(userId: string, message: WebSocketMessage): void {

for (const client of this.clients.values()) {

if (client.userId === userId) {

this.sendToClient(client.id, message);

}

}

}

sendToClient(clientId: string, message: WebSocketMessage): boolean {

const client = this.clients.get(clientId);

if (!client || !client.isAlive) return false;

try {

client.socket.send(JSON.stringify(message));

return true;

} catch (error) {

console.error(`Failed to send message to client ${clientId}:`, error);

this.removeClient(clientId);

return false;

}

}

private setupSocketHandlers(client: WebSocketClient): void {

client.socket.onmessage = (event) => {

try {

const message: WebSocketMessage = JSON.parse(event.data);

this.handleMessage(client, message);

} catch (error) {

console.error('Invalid WebSocket message:', error);

}

};

client.socket.onclose = () => {

this.removeClient(client.id);

};

client.socket.onerror = (error) => {

console.error(`WebSocket error for client ${client.id}:`, error);

this.removeClient(client.id);

};

client.socket.onpong = () => {

client.lastPing = Date.now();

client.isAlive = true;

};

}

private handleMessage(client: WebSocketClient, message: WebSocketMessage): void {

switch (message.type) {

case 'subscribe':

if (message.data?.channel) {

this.subscribeToChannel(client.id, message.data.channel);

}

break;

case 'unsubscribe':

if (message.data?.channel) {

this.unsubscribeFromChannel(client.id, message.data.channel);

}

break;

case 'ping':

this.sendToClient(client.id, {

type: 'pong',

timestamp: Date.now()

});

break;

}

}

private performHeartbeat(): void {

const now = Date.now();

for (const [clientId, client] of this.clients) {

if (now - client.lastPing > 60000) { // 60 seconds timeout

console.log(`Client ${clientId} timed out`);

this.removeClient(clientId);

} else {

// Send ping

try {

client.socket.ping();

} catch (error) {

this.removeClient(clientId);

}

}

}

}

private generateClientId(): string {

return `client\_${Date.now()}\_${Math.random().toString(36).substr(2, 9)}`;

}

// Public methods for application use

broadcastSignalUpdate(signal: SignalResult): void {

this.broadcastToChannel('signals', {

type: 'signal\_update',

data: signal,

timestamp: Date.now()

});

}

broadcastPortfolioUpdate(userId: string, portfolio: Portfolio): void {

this.sendToUser(userId, {

type: 'portfolio\_update',

data: portfolio,

timestamp: Date.now()

});

}

getActiveConnections(): number {

return this.clients.size;

}

getChannelSubscribers(channel: string): number {

return this.channels.get(channel)?.size || 0;

}

}

// ===================================================================

// SERVER-SENT EVENTS (SSE)

// ===================================================================

// lib/realtime/sseManager.ts

interface SSEConnection {

id: string;

userId?: string;

response: any; // Express Response object

subscriptions: Set<string>;

lastActivity: number;

}

class SSEManager {

private connections = new Map<string, SSEConnection>();

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

createConnection(req: any, res: any, userId?: string): string {

const connectionId = this.generateConnectionId();

// Set SSE headers

res.writeHead(200, {

'Content-Type': 'text/event-stream',

'Cache-Control': 'no-cache',

'Connection': 'keep-alive',

'Access-Control-Allow-Origin': '\*',

'Access-Control-Allow-Headers': 'Cache-Control'

});

const connection: SSEConnection = {

id: connectionId,

userId,

response: res,

subscriptions: new Set(),

lastActivity: Date.now()

};

this.connections.set(connectionId, connection);

// Send initial connection event

this.sendEvent(connectionId, 'connected', {

connectionId,

userId,

timestamp: Date.now()

});

// Handle client disconnect

req.on('close', () => {

this.removeConnection(connectionId);

});

console.log(`SSE connection established: ${connectionId} (user: ${userId})`);

return connectionId;

}

removeConnection(connectionId: string): void {

const connection = this.connections.get(connectionId);

if (!connection) return;

// Unsubscribe from all channels

connection.subscriptions.forEach(channel => {

this.unsubscribeFromChannel(connectionId, channel);

});

this.connections.delete(connectionId);

console.log(`SSE connection closed: ${connectionId}`);

}

subscribeToChannel(connectionId: string, channel: string): boolean {

const connection = this.connections.get(connectionId);

if (!connection) return false;

connection.subscriptions.add(channel);

if (!this.channels.has(channel)) {

this.channels.set(channel, new Set());

}

this.channels.get(channel)!.add(connectionId);

return true;

}

unsubscribeFromChannel(connectionId: string, channel: string): boolean {

const connection = this.connections.get(connectionId);

if (!connection) return false;

connection.subscriptions.delete(channel);

this.channels.get(channel)?.delete(connectionId);

return true;

}

sendEvent(connectionId: string, event: string, data: any): boolean {

const connection = this.connections.get(connectionId);

if (!connection) return false;

try {

const message = `event: ${event}\ndata: ${JSON.stringify(data)}\n\n`;

connection.response.write(message);

connection.lastActivity = Date.now();

return true;

} catch (error) {

console.error(`Failed to send SSE to ${connectionId}:`, error);

this.removeConnection(connectionId);

return false;

}

}

broadcastToChannel(channel: string, event: string, data: any): void {

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

if (!subscribers) return;

subscribers.forEach(connectionId => {

this.sendEvent(connectionId, event, data);

});

}

sendToUser(userId: string, event: string, data: any): void {

for (const connection of this.connections.values()) {

if (connection.userId === userId) {

this.sendEvent(connection.id, event, data);

}

}

}

// Heartbeat to keep connections alive

sendHeartbeat(): void {

for (const [connectionId, connection] of this.connections) {

if (Date.now() - connection.lastActivity > 300000) { // 5 minutes

this.removeConnection(connectionId);

} else {

this.sendEvent(connectionId, 'heartbeat', { timestamp: Date.now() });

}

}

}

private generateConnectionId(): string {

return `sse\_${Date.now()}\_${Math.random().toString(36).substr(2, 9)}`;

}

}

// ===================================================================

// \*\*4. 🚀 DEPLOYMENT ARCHITECTURE\*\*

// ===================================================================

// ===================================================================

// VERCEL DEPLOYMENT CONFIGURATION

// ===================================================================

// vercel.json

const vercelConfig = {

"version": 2,

"name": "kurzora-platform",

"builds": [

{

"src": "package.json",

"use": "@vercel/next"

}

],

"routes": [

{

"src": "/api/webhooks/(.\*)",

"dest": "/api/webhooks/$1"

},

{

"src": "/api/realtime/(.\*)",

"dest": "/api/realtime/$1"

}

],

"env": {

"NODE\_ENV": "production",

"NEXT\_PUBLIC\_APP\_URL": "https://kurzora.com"

},

"functions": {

"app/api/\*\*/\*.ts": {

"maxDuration": 30

}

},

"headers": [

{

"source": "/api/(.\*)",

"headers": [

{

"key": "Access-Control-Allow-Origin",

"value": "\*"

},

{

"key": "Access-Control-Allow-Methods",

"value": "GET, POST, PUT, DELETE, OPTIONS"

},

{

"key": "Access-Control-Allow-Headers",

"value": "Content-Type, Authorization"

}

]

}

]

};

// ===================================================================

// ENVIRONMENT CONFIGURATION

// ===================================================================

// .env.production

const productionEnv = `

# Database

NEXT\_PUBLIC\_SUPABASE\_URL=https://your-project.supabase.co

NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY=your\_supabase\_anon\_key

SUPABASE\_SERVICE\_ROLE\_KEY=your\_supabase\_service\_role\_key

# Market Data

POLYGON\_API\_KEY=your\_polygon\_api\_key

POLYGON\_WEBHOOK\_SECRET=your\_polygon\_webhook\_secret

# Notifications

SENDGRID\_API\_KEY=your\_sendgrid\_api\_key

TELEGRAM\_BOT\_TOKEN=your\_telegram\_bot\_token

TELEGRAM\_WEBHOOK\_SECRET=your\_telegram\_webhook\_secret

VAPID\_PUBLIC\_KEY=your\_vapid\_public\_key

VAPID\_PRIVATE\_KEY=your\_vapid\_private\_key

# Payments

STRIPE\_PUBLISHABLE\_KEY=pk\_live\_your\_stripe\_publishable\_key

STRIPE\_SECRET\_KEY=sk\_live\_your\_stripe\_secret\_key

STRIPE\_WEBHOOK\_SECRET=whsec\_your\_stripe\_webhook\_secret

STRIPE\_BASIC\_PRICE\_ID=price\_your\_basic\_price\_id

STRIPE\_PRO\_PRICE\_ID=price\_your\_pro\_price\_id

STRIPE\_ELITE\_PRICE\_ID=price\_your\_elite\_price\_id

# Make.com

MAKE\_API\_KEY=your\_make\_api\_key

MAKE\_SIGNAL\_WEBHOOK\_URL=https://hook.eu1.make.com/your\_signal\_webhook

MAKE\_TRADE\_WEBHOOK\_URL=https://hook.eu1.make.com/your\_trade\_webhook

MAKE\_RISK\_WEBHOOK\_URL=https://hook.eu1.make.com/your\_risk\_webhook

MAKE\_SYSTEM\_WEBHOOK\_URL=https://hook.eu1.make.com/your\_system\_webhook

# Monitoring

SENTRY\_DSN=your\_sentry\_dsn

MIXPANEL\_TOKEN=your\_mixpanel\_token

# Security

JWT\_SECRET=your\_jwt\_secret\_key

ENCRYPTION\_KEY=your\_encryption\_key

API\_RATE\_LIMIT\_WINDOW=900000

API\_RATE\_LIMIT\_MAX=100

# App

NEXT\_PUBLIC\_APP\_URL=https://kurzora.com

NEXT\_PUBLIC\_API\_URL=https://kurzora.com/api

`;

// ===================================================================

// FIREBASE FUNCTIONS DEPLOYMENT

// ===================================================================

// firebase.json

const firebaseConfig = {

"functions": {

"source": "functions",

"runtime": "nodejs18",

"predeploy": ["npm --prefix \"$RESOURCE\_DIR\" run build"],

"postdeploy": ["npm --prefix \"$RESOURCE\_DIR\" run logs"]

},

"firestore": {

"rules": "firestore.rules",

"indexes": "firestore.indexes.json"

},

"hosting": [

{

"target": "kurzora-platform",

"public": "out",

"ignore": ["firebase.json", "\*\*/.\*", "\*\*/node\_modules/\*\*"],

"rewrites": [

{

"source": "\*\*",

"destination": "/index.html"

}

]

}

],

"emulators": {

"functions": {

"port": 5001

},

"firestore": {

"port": 8080

},

"hosting": {

"port": 5000

},

"ui": {

"enabled": true

}

}

};

// functions/src/index.ts

import \* as functions from 'firebase-functions';

import { MarketDataPipeline } from './lib/marketDataPipeline';

import { SignalProcessor } from './lib/signalProcessor';

// Market data processing function

export const processMarketData = functions

.runWith({

timeoutSeconds: 540,

memory: '2GB'

})

.pubsub

.schedule('every 5 minutes')

.timeZone('America/New\_York')

.onRun(async (context) => {

console.log('Starting market data processing...');

const pipeline = new MarketDataPipeline();

await pipeline.processLatestData();

console.log('Market data processing completed');

});

// Signal generation function

export const generateSignals = functions

.runWith({

timeoutSeconds: 300,

memory: '1GB'

})

.pubsub

.schedule('every 5 minutes')

.timeZone('America/New\_York')

.onRun(async (context) => {

console.log('Starting signal generation...');

const signalProcessor = new SignalProcessor();

await signalProcessor.scanAndGenerateSignals();

console.log('Signal generation completed');

});

// Portfolio update function

export const updatePortfolios = functions

.runWith({

timeoutSeconds: 120,

memory: '512MB'

})

.pubsub

.schedule('every 1 minutes')

.timeZone('America/New\_York')

.onRun(async (context) => {

console.log('Updating portfolios...');

// Update all active portfolios

await updateAllPortfolios();

console.log('Portfolio updates completed');

});

// ===================================================================

// \*\*5. 📊 MONITORING & ANALYTICS\*\*

// ===================================================================

// lib/monitoring/analytics.ts

import mixpanel from 'mixpanel';

interface AnalyticsEvent {

userId?: string;

event: string;

properties: Record<string, any>;

timestamp?: Date;

}

class AnalyticsService {

private mixpanel: any;

private isProduction = process.env.NODE\_ENV === 'production';

constructor() {

if (this.isProduction) {

this.mixpanel = mixpanel.init(process.env.MIXPANEL\_TOKEN!);

}

}

track(event: AnalyticsEvent): void {

if (!this.isProduction) {

console.log('Analytics Event:', event);

return;

}

try {

this.mixpanel.track(event.event, {

distinct\_id: event.userId,

...event.properties,

timestamp: event.timestamp || new Date(),

platform: 'web',

environment: 'production'

});

} catch (error) {

console.error('Analytics tracking failed:', error);

}

}

trackSignalGenerated(signal: SignalResult): void {

this.track({

event: 'Signal Generated',

properties: {

ticker: signal.ticker,

score: signal.finalScore,

strength: signal.strength,

signalType: signal.signalType,

confidence: signal.confluence.overallConfidence

}

});

}

trackTradeExecuted(trade: PaperTrade, userId: string): void {

this.track({

userId,

event: 'Trade Executed',

properties: {

ticker: trade.ticker,

tradeType: trade.tradeType,

shares: trade.shares,

entryPrice: trade.entryPrice,

positionValue: trade.positionValue,

signalScore: trade.metadata.signalScore

}

});

}

trackUserSubscription(userId: string, planId: string, revenue: number): void {

this.track({

userId,

event: 'Subscription Created',

properties: {

planId,

revenue,

currency: 'USD'

}

});

if (this.isProduction) {

this.mixpanel.people.track\_charge(userId, revenue);

}

}

}

// ===================================================================

// SYSTEM HEALTH MONITORING

// ===================================================================

// lib/monitoring/healthCheck.ts

interface SystemHealth {

status: 'healthy' | 'degraded' | 'unhealthy';

timestamp: Date;

services: ServiceHealth[];

metrics: SystemMetrics;

}

interface ServiceHealth {

name: string;

status: 'up' | 'down' | 'slow';

responseTime?: number;

lastCheck: Date;

error?: string;

}

interface SystemMetrics {

uptime: number;

memory: {

used: number;

total: number;

percentage: number;

};

database: {

connections: number;

avgQueryTime: number;

};

api: {

requestsPerMinute: number;

errorRate: number;

avgResponseTime: number;

};

}

class HealthMonitor {

private services = [

{ name: 'supabase', url: process.env.NEXT\_PUBLIC\_SUPABASE\_URL! },

{ name: 'polygon', url: 'https://api.polygon.io' },

{ name: 'sendgrid', url: 'https://api.sendgrid.com' },

{ name: 'stripe', url: 'https://api.stripe.com' }

];

async checkHealth(): Promise<SystemHealth> {

const serviceChecks = await Promise.all(

this.services.map(service => this.checkService(service))

);

const overallStatus = this.determineOverallStatus(serviceChecks);

const metrics = await this.collectMetrics();

return {

status: overallStatus,

timestamp: new Date(),

services: serviceChecks,

metrics

};

}

private async checkService(service: { name: string; url: string }): Promise<ServiceHealth> {

const startTime = Date.now();

try {

const response = await fetch(`${service.url}/health`, {

method: 'GET',

timeout: 5000

});

const responseTime = Date.now() - startTime;

return {

name: service.name,

status: response.ok ? (responseTime > 2000 ? 'slow' : 'up') : 'down',

responseTime,

lastCheck: new Date()

};

} catch (error) {

return {

name: service.name,

status: 'down',

responseTime: Date.now() - startTime,

lastCheck: new Date(),

error: error.message

};

}

}

private determineOverallStatus(services: ServiceHealth[]): 'healthy' | 'degraded' | 'unhealthy' {

const downServices = services.filter(s => s.status === 'down').length;

const slowServices = services.filter(s => s.status === 'slow').length;

if (downServices > 0) return 'unhealthy';

if (slowServices > 1) return 'degraded';

return 'healthy';

}

private async collectMetrics(): Promise<SystemMetrics> {

const process = await import('process');

const memUsage = process.memoryUsage();

return {

uptime: process.uptime(),

memory: {

used: memUsage.heapUsed,

total: memUsage.heapTotal,

percentage: (memUsage.heapUsed / memUsage.heapTotal) \* 100

},

database: {

connections: await this.getDatabaseConnections(),

avgQueryTime: await this.getAverageQueryTime()

},

api: {

requestsPerMinute: await this.getRequestsPerMinute(),

errorRate: await this.getErrorRate(),

avgResponseTime: await this.getAverageResponseTime()

}

};

}

}

// ===================================================================

// \*\*6. 💾 BACKUP & RECOVERY\*\*

// ===================================================================

// lib/backup/backupManager.ts

class BackupManager {

private supabase = createClient(

process.env.NEXT\_PUBLIC\_SUPABASE\_URL!,

process.env.SUPABASE\_SERVICE\_ROLE\_KEY!

);

async createDatabaseBackup(): Promise<boolean> {

try {

// Export critical tables

const tables = [

'users',

'subscriptions',

'trading\_signals',

'user\_trades',

'portfolios',

'market\_data'

];

const backupData: Record<string, any[]> = {};

for (const table of tables) {

const { data, error } = await this.supabase

.from(table)

.select('\*');

if (error) throw error;

backupData[table] = data || [];

}

// Store backup

const backupFileName = `backup\_${new Date().toISOString().split('T')[0]}.json`;

const backupContent = JSON.stringify(backupData, null, 2);

// Upload to storage

await this.uploadBackup(backupFileName, backupContent);

console.log(`Database backup created: ${backupFileName}`);

return true;

} catch (error) {

console.error('Database backup failed:', error);

return false;

}

}

async restoreFromBackup(backupFileName: string): Promise<boolean> {

try {

// Download backup file

const backupContent = await this.downloadBackup(backupFileName);

const backupData = JSON.parse(backupContent);

// Restore each table

for (const [tableName, tableData] of Object.entries(backupData)) {

if (Array.isArray(tableData) && tableData.length > 0) {

const { error } = await this.supabase

.from(tableName)

.upsert(tableData);

if (error) throw error;

console.log(`Restored ${tableData.length} records to ${tableName}`);

}

}

console.log(`Database restored from: ${backupFileName}`);

return true;

} catch (error) {

console.error('Database restore failed:', error);

return false;

}

}

private async uploadBackup(fileName: string, content: string): Promise<void> {

// Upload to Supabase Storage or AWS S3

const { error } = await this.supabase.storage

.from('backups')

.upload(fileName, content, {

contentType: 'application/json'

});

if (error) throw error;

}

private async downloadBackup(fileName: string): Promise<string> {

const { data, error } = await this.supabase.storage

.from('backups')

.download(fileName);

if (error) throw error;

return await data.text();

}

}

// ===================================================================

// \*\*7. 🔒 SECURITY & COMPLIANCE\*\*

// ===================================================================

// lib/security/encryption.ts

import crypto from 'crypto';

class EncryptionService {

private algorithm = 'aes-256-gcm';

private key = Buffer.from(process.env.ENCRYPTION\_KEY!, 'hex');

encrypt(text: string): { encrypted: string; iv: string; tag: string } {

const iv = crypto.randomBytes(16);

const cipher = crypto.createCipher(this.algorithm, this.key);

cipher.setAAD(Buffer.from('kurzora-platform'));

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

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

const tag = cipher.getAuthTag();

return {

encrypted,

iv: iv.toString('hex'),

tag: tag.toString('hex')

};

}

decrypt(encryptedData: { encrypted: string; iv: string; tag: string }): string {

const decipher = crypto.createDecipher(this.algorithm, this.key);

decipher.setAAD(Buffer.from('kurzora-platform'));

decipher.setAuthTag(Buffer.from(encryptedData.tag, 'hex'));

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

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

return decrypted;

}

}

// lib/security/compliance.ts

interface ComplianceReport {

gdprCompliance: boolean;

dataRetentionPolicy: boolean;

userConsentTracking: boolean;

dataEncryption: boolean;

auditTrails: boolean;

accessControls: boolean;

lastAssessment: Date;

recommendations: string[];

}

class ComplianceManager {

async generateComplianceReport(): Promise<ComplianceReport> {

return {

gdprCompliance: await this.checkGDPRCompliance(),

dataRetentionPolicy: await this.checkDataRetention(),

userConsentTracking: await this.checkConsentTracking(),

dataEncryption: await this.checkEncryption(),

auditTrails: await this.checkAuditTrails(),

accessControls: await this.checkAccessControls(),

lastAssessment: new Date(),

recommendations: await this.generateRecommendations()

};

}

private async checkGDPRCompliance(): Promise<boolean> {

// Check GDPR compliance requirements

// - Right to access

// - Right to rectification

// - Right to erasure

// - Right to data portability

return true; // Implement actual checks

}

}

// ===================================================================

// \*\*8. 💰 COST OPTIMIZATION\*\*

// ===================================================================

// lib/optimization/costMonitor.ts

interface CostMetrics {

polygonApiCalls: { count: number; cost: number };

supabaseCosts: { storage: number; bandwidth: number; requests: number };

vercelBandwidth: { amount: number; cost: number };

sendgridEmails: { count: number; cost: number };

totalMonthlyCost: number;

costPerUser: number;

recommendations: string[];

}

class CostOptimizer {

private pricing = {

polygonPerCall: 0.004, // $0.004 per API call

supabasePerGB: 0.125, // $0.125 per GB storage

vercelPerGB: 0.40, // $0.40 per GB bandwidth

sendgridPer1000: 14.95 // $14.95 per 1000 emails

};

async analyzeCosts(): Promise<CostMetrics> {

const polygonCalls = await this.getPolygonUsage();

const supabaseUsage = await this.getSupabaseUsage();

const vercelUsage = await this.getVercelUsage();

const sendgridUsage = await this.getSendgridUsage();

const totalCost =

polygonCalls.cost +

supabaseUsage.storage +

supabaseUsage.bandwidth +

vercelUsage.cost +

sendgridUsage.cost;

const activeUsers = await this.getActiveUserCount();

return {

polygonApiCalls: polygonCalls,

supabaseCosts: supabaseUsage,

vercelBandwidth: vercelUsage,

sendgridEmails: sendgridUsage,

totalMonthlyCost: totalCost,

costPerUser: activeUsers > 0 ? totalCost / activeUsers : 0,

recommendations: this.generateCostRecommendations(totalCost, activeUsers)

};

}

private generateCostRecommendations(totalCost: number, userCount: number): string[] {

const recommendations: string[] = [];

if (totalCost > 1000) {

recommendations.push('Consider upgrading to higher-tier plans for better pricing');

}

if (userCount > 0 && totalCost / userCount > 50) {

recommendations.push('Cost per user is high - optimize API usage');

}

recommendations.push('Implement API response caching to reduce external calls');

recommendations.push('Use database connection pooling to reduce costs');

return recommendations;

}

}

export {

MakeWorkflowTrigger,

TelegramBotService,

EmailNotificationService,

PushNotificationService,

WebSocketManager,

SSEManager,

AnalyticsService,

HealthMonitor,

BackupManager,

EncryptionService,

ComplianceManager,

CostOptimizer

};