**Sistema de Pagos, Gestión de Usuarios y Soporte - TurisGal**

**1. SISTEMA DE PAGOS Y FACTURACIÓN**

**Integración de Proveedores de Pago**

// src/services/PaymentService.ts

export class PaymentService {

private stripeClient: Stripe;

private paypalClient: any;

private redsysClient: any;

constructor() {

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

apiVersion: '2023-10-16'

});

this.initializePayPal();

this.initializeRedsys();

}

async processPayment(paymentData: PaymentRequest): Promise<PaymentResult> {

const { amount, currency, paymentMethod, bookingId, metadata } = paymentData;

try {

let result: PaymentResult;

switch (paymentMethod.provider) {

case 'stripe':

result = await this.processStripePayment(paymentData);

break;

case 'paypal':

result = await this.processPayPalPayment(paymentData);

break;

case 'redsys':

result = await this.processRedsysPayment(paymentData);

break;

case 'bizum':

result = await this.processBizumPayment(paymentData);

break;

default:

throw new Error(`Unsupported payment provider: ${paymentMethod.provider}`);

}

// Registrar la transacción

await this.recordTransaction(result, bookingId);

// Actualizar estado de la reserva si el pago es exitoso

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

await this.updateBookingPaymentStatus(bookingId, 'paid');

await this.generateInvoice(bookingId, result.transactionId);

}

return result;

} catch (error) {

console.error('Payment processing error:', error);

await this.recordFailedPayment(paymentData, error);

throw error;

}

}

private async processStripePayment(paymentData: PaymentRequest): Promise<PaymentResult> {

const { amount, currency, paymentMethod, customerEmail, bookingId } = paymentData;

// Crear o recuperar cliente de Stripe

const customer = await this.getOrCreateStripeCustomer(customerEmail);

// Crear PaymentIntent

const paymentIntent = await this.stripeClient.paymentIntents.create({

amount: Math.round(amount \* 100), // Stripe usa centavos

currency: currency.toLowerCase(),

customer: customer.id,

payment\_method: paymentMethod.stripePaymentMethodId,

confirmation\_method: 'manual',

confirm: true,

return\_url: `${process.env.FRONTEND\_URL}/payment/return`,

metadata: {

bookingId,

platform: 'turisgal'

},

description: `Reserva TurisGal #${bookingId}`,

statement\_descriptor: 'TURISGAL\*',

receipt\_email: customerEmail,

setup\_future\_usage: 'off\_session' // Para pagos futuros

});

return this.handleStripePaymentResult(paymentIntent);

}

private async handleStripePaymentResult(paymentIntent: Stripe.PaymentIntent): Promise<PaymentResult> {

switch (paymentIntent.status) {

case 'succeeded':

return {

status: 'succeeded',

transactionId: paymentIntent.id,

amount: paymentIntent.amount / 100,

currency: paymentIntent.currency.toUpperCase(),

provider: 'stripe',

metadata: paymentIntent.metadata

};

case 'requires\_action':

return {

status: 'requires\_action',

clientSecret: paymentIntent.client\_secret,

nextAction: paymentIntent.next\_action,

provider: 'stripe'

};

case 'requires\_payment\_method':

return {

status: 'failed',

error: 'Payment method was declined',

provider: 'stripe'

};

default:

return {

status: 'processing',

transactionId: paymentIntent.id,

provider: 'stripe'

};

}

}

private async processPayPalPayment(paymentData: PaymentRequest): Promise<PaymentResult> {

const { amount, currency, bookingId } = paymentData;

const order = await this.paypalClient.orders.create({

intent: 'CAPTURE',

purchase\_units: [{

amount: {

currency\_code: currency,

value: amount.toFixed(2)

},

description: `Reserva TurisGal #${bookingId}`,

custom\_id: bookingId,

invoice\_id: `TG-${bookingId}-${Date.now()}`

}],

payment\_source: {

paypal: {

experience\_context: {

payment\_method\_preference: 'IMMEDIATE\_PAYMENT\_REQUIRED',

return\_url: `${process.env.FRONTEND\_URL}/payment/paypal/return`,

cancel\_url: `${process.env.FRONTEND\_URL}/payment/paypal/cancel`

}

}

}

});

return {

status: 'requires\_action',

transactionId: order.id,

redirectUrl: order.links.find(link => link.rel === 'approve')?.href,

provider: 'paypal'

};

}

private async processRedsysPayment(paymentData: PaymentRequest): Promise<PaymentResult> {

// Implementación para Redsys (sistema de pago español)

const { amount, currency, bookingId } = paymentData;

const redsysParams = {

Ds\_Merchant\_Amount: Math.round(amount \* 100).toString(),

Ds\_Merchant\_Currency: currency === 'EUR' ? '978' : '840',

Ds\_Merchant\_Order: this.generateRedsysOrderNumber(bookingId),

Ds\_Merchant\_MerchantCode: process.env.REDSYS\_MERCHANT\_CODE!,

Ds\_Merchant\_Terminal: process.env.REDSYS\_TERMINAL!,

Ds\_Merchant\_TransactionType: '0',

Ds\_Merchant\_MerchantURL: `${process.env.API\_URL}/payments/redsys/notification`,

Ds\_Merchant\_UrlOK: `${process.env.FRONTEND\_URL}/payment/success`,

Ds\_Merchant\_UrlKO: `${process.env.FRONTEND\_URL}/payment/error`

};

const signature = this.generateRedsysSignature(redsysParams);

const merchantParameters = Buffer.from(JSON.stringify(redsysParams)).toString('base64');

return {

status: 'requires\_action',

provider: 'redsys',

formData: {

Ds\_SignatureVersion: 'HMAC\_SHA256\_V1',

Ds\_MerchantParameters: merchantParameters,

Ds\_Signature: signature

},

redirectUrl: process.env.REDSYS\_URL

};

}

async processRefund(transactionId: string, amount?: number, reason?: string): Promise<RefundResult> {

const transaction = await prisma.transaction.findUnique({

where: { transactionId }

});

if (!transaction) {

throw new Error('Transaction not found');

}

let refundResult: RefundResult;

switch (transaction.provider) {

case 'stripe':

refundResult = await this.processStripeRefund(transactionId, amount, reason);

break;

case 'paypal':

refundResult = await this.processPayPalRefund(transactionId, amount, reason);

break;

case 'redsys':

refundResult = await this.processRedsysRefund(transactionId, amount, reason);

break;

default:

throw new Error(`Refunds not supported for provider: ${transaction.provider}`);

}

// Registrar el reembolso

await this.recordRefund(refundResult, transaction.bookingId);

return refundResult;

}

async generateInvoice(bookingId: string, transactionId: string): Promise<Invoice> {

const booking = await prisma.booking.findUnique({

where: { id: bookingId },

include: {

user: true,

property: { include: { owner: true } },

room: true

}

});

if (!booking) {

throw new Error('Booking not found');

}

const invoice: Invoice = {

id: `TG-${bookingId}-${Date.now()}`,

bookingId,

transactionId,

issueDate: new Date(),

dueDate: new Date(), // Pago inmediato

customer: {

name: `${booking.user.firstName} ${booking.user.lastName}`,

email: booking.user.email,

address: booking.guestDetails?.address,

taxId: booking.guestDetails?.taxId

},

vendor: {

name: booking.property.owner.companyName || booking.property.owner.contactName,

address: booking.property.address,

taxId: booking.property.owner.taxId,

email: booking.property.owner.email

},

items: [

{

description: `Estancia en ${booking.property.name}`,

quantity: booking.checkOutDate.getTime() - booking.checkInDate.getTime() / (1000 \* 60 \* 60 \* 24),

unitPrice: booking.totalAmount,

total: booking.totalAmount

}

],

subtotal: booking.totalAmount \* 0.9, // Sin IVA

taxes: [

{

name: 'IVA',

rate: 0.1,

amount: booking.totalAmount \* 0.1

}

],

total: booking.totalAmount,

currency: 'EUR',

status: 'paid'

};

// Generar PDF

const pdfBuffer = await this.generateInvoicePDF(invoice);

// Guardar en S3

const invoiceUrl = await this.uploadInvoicePDF(invoice.id, pdfBuffer);

// Guardar en base de datos

await prisma.invoice.create({

data: {

id: invoice.id,

bookingId,

transactionId,

customerData: invoice.customer,

vendorData: invoice.vendor,

items: invoice.items,

subtotal: invoice.subtotal,

total: invoice.total,

currency: invoice.currency,

status: invoice.status,

pdfUrl: invoiceUrl

}

});

// Enviar por email

await this.sendInvoiceEmail(booking.user.email, invoice, pdfBuffer);

return invoice;

}

private async generateInvoicePDF(invoice: Invoice): Promise<Buffer> {

const PDFDocument = require('pdfkit');

const doc = new PDFDocument();

const chunks: Buffer[] = [];

doc.on('data', chunk => chunks.push(chunk));

return new Promise((resolve, reject) => {

doc.on('end', () => {

resolve(Buffer.concat(chunks));

});

doc.on('error', reject);

// Header

doc.fontSize(20).text('FACTURA', 50, 50);

doc.fontSize(12).text(`Número: ${invoice.id}`, 50, 80);

doc.text(`Fecha: ${invoice.issueDate.toLocaleDateString('es-ES')}`, 50, 95);

// Datos del vendedor

doc.text('DATOS DEL EMISOR:', 50, 130);

doc.text(invoice.vendor.name, 50, 145);

doc.text(invoice.vendor.taxId, 50, 160);

doc.text(JSON.stringify(invoice.vendor.address), 50, 175);

// Datos del cliente

doc.text('DATOS DEL CLIENTE:', 300, 130);

doc.text(invoice.customer.name, 300, 145);

doc.text(invoice.customer.email, 300, 160);

if (invoice.customer.taxId) {

doc.text(invoice.customer.taxId, 300, 175);

}

// Línea de productos/servicios

let yPosition = 220;

doc.text('CONCEPTO', 50, yPosition);

doc.text('CANTIDAD', 200, yPosition);

doc.text('PRECIO UNIT.', 300, yPosition);

doc.text('TOTAL', 450, yPosition);

doc.moveTo(50, yPosition + 15).lineTo(550, yPosition + 15).stroke();

yPosition += 30;

invoice.items.forEach(item => {

doc.text(item.description, 50, yPosition);

doc.text(item.quantity.toString(), 200, yPosition);

doc.text(`€${item.unitPrice.toFixed(2)}`, 300, yPosition);

doc.text(`€${item.total.toFixed(2)}`, 450, yPosition);

yPosition += 20;

});

// Totales

yPosition += 20;

doc.text(`Subtotal: €${invoice.subtotal.toFixed(2)}`, 350, yPosition);

invoice.taxes.forEach(tax => {

yPosition += 15;

doc.text(`${tax.name} (${(tax.rate \* 100).toFixed(1)}%): €${tax.amount.toFixed(2)}`, 350, yPosition);

});

yPosition += 20;

doc.fontSize(14).text(`TOTAL: €${invoice.total.toFixed(2)}`, 350, yPosition);

// Pie de página

doc.fontSize(10).text('Pagado mediante TurisGal', 50, 700);

doc.text(`ID Transacción: ${invoice.transactionId}`, 50, 715);

doc.end();

});

}

// Gestión de suscripciones para propietarios

async createSubscription(propertyOwnerId: string, planId: string): Promise<Subscription> {

const owner = await prisma.propertyOwner.findUnique({

where: { id: propertyOwnerId }

});

if (!owner) {

throw new Error('Property owner not found');

}

const plan = await this.getSubscriptionPlan(planId);

// Crear cliente en Stripe si no existe

let stripeCustomerId = owner.stripeCustomerId;

if (!stripeCustomerId) {

const customer = await this.stripeClient.customers.create({

email: owner.email,

name: owner.contactName,

metadata: {

propertyOwnerId: owner.id,

platform: 'turisgal'

}

});

stripeCustomerId = customer.id;

await prisma.propertyOwner.update({

where: { id: propertyOwnerId },

data: { stripeCustomerId }

});

}

// Crear suscripción en Stripe

const stripeSubscription = await this.stripeClient.subscriptions.create({

customer: stripeCustomerId,

items: [{ price: plan.stripePriceId }],

trial\_period\_days: plan.trialDays,

metadata: {

propertyOwnerId,

planId

}

});

// Guardar en base de datos

const subscription = await prisma.subscription.create({

data: {

id: stripeSubscription.id,

propertyOwnerId,

planId,

status: stripeSubscription.status,

currentPeriodStart: new Date(stripeSubscription.current\_period\_start \* 1000),

currentPeriodEnd: new Date(stripeSubscription.current\_period\_end \* 1000),

trialEnd: stripeSubscription.trial\_end ? new Date(stripeSubscription.trial\_end \* 1000) : null

}

});

return subscription;

}

}

// Planes de suscripción

const subscriptionPlans = {

basic: {

id: 'basic',

name: 'Plan Básico',

description: 'Perfecto para empezar',

price: 29,

currency: 'EUR',

interval: 'month',

features: [

'Hasta 2 propiedades',

'Check-in digital ilimitado',

'Soporte por email',

'Dashboard básico',

'Exportación de datos'

],

limits: {

properties: 2,

checkins: Infinity,

support: 'email'

}

},

professional: {

id: 'professional',

name: 'Plan Profesional',

description: 'Para gestores de múltiples propiedades',

price: 79,

currency: 'EUR',

interval: 'month',

features: [

'Hasta 10 propiedades',

'Analytics avanzados',

'Integraciones PMS',

'Soporte prioritario',

'API access',

'White-label opcional'

],

limits: {

properties: 10,

checkins: Infinity,

support: 'priority',

api: true

}

},

enterprise: {

id: 'enterprise',

name: 'Plan Empresarial',

description: 'Para grandes cadenas hoteleras',

price: 199,

currency: 'EUR',

interval: 'month',

features: [

'Propiedades ilimitadas',

'Analytics empresariales',

'Integraciones personalizadas',

'Soporte 24/7',

'SLA garantizado',

'Implementación dedicada'

],

limits: {

properties: Infinity,

checkins: Infinity,

support: '24/7',

api: true,

customIntegrations: true

}

}

};

**2. GESTIÓN AVANZADA DE USUARIOS Y ROLES**

**Sistema de Roles y Permisos**

// src/services/UserManagementService.ts

export class UserManagementService {

async createUser(userData: CreateUserRequest): Promise<User> {

const hashedPassword = await bcrypt.hash(userData.password, 12);

const user = await prisma.user.create({

data: {

...userData,

passwordHash: hashedPassword,

emailVerified: false,

phoneVerified: false,

accountStatus: 'pending\_verification'

}

});

// Enviar email de verificación

await this.sendVerificationEmail(user);

// Enviar SMS de verificación si hay teléfono

if (user.phone) {

await this.sendVerificationSMS(user);

}

return user;

}

async createPropertyOwner(ownerData: CreatePropertyOwnerRequest, createdBy?: string): Promise<PropertyOwner> {

const hashedPassword = await bcrypt.hash(ownerData.password, 12);

const owner = await prisma.propertyOwner.create({

data: {

...ownerData,

passwordHash: hashedPassword,

role: ownerData.role || 'owner',

permissions: this.getDefaultPermissions(ownerData.role || 'owner'),

accountStatus: 'active',

createdBy

}

});

// Asignar plan básico por defecto

if (ownerData.autoSubscribe !== false) {

await this.paymentService.createSubscription(owner.id, 'basic');

}

return owner;

}

private getDefaultPermissions(role: string): UserPermissions {

const permissions: Record<string, UserPermissions> = {

super\_admin: {

users: ['create', 'read', 'update', 'delete'],

properties: ['create', 'read', 'update', 'delete'],

bookings: ['create', 'read', 'update', 'delete'],

analytics: ['read', 'export'],

system: ['read', 'update'],

billing: ['read', 'update'],

support: ['read', 'update', 'delete']

},

admin: {

users: ['create', 'read', 'update'],

properties: ['create', 'read', 'update', 'delete'],

bookings: ['read', 'update'],

analytics: ['read', 'export'],

billing: ['read']

},

owner: {

properties: ['create', 'read', 'update', 'delete'],

bookings: ['read', 'update'],

analytics: ['read'],

staff: ['create', 'read', 'update']

},

manager: {

properties: ['read', 'update'],

bookings: ['read', 'update'],

analytics: ['read']

},

staff: {

bookings: ['read', 'update'],

checkins: ['read', 'update']

}

};

return permissions[role] || permissions.staff;

}

async assignRole(userId: string, role: string, assignedBy: string): Promise<void> {

// Verificar que quien asigna tiene permisos

const assigner = await this.getUserWithPermissions(assignedBy);

if (!this.hasPermission(assigner, 'users', 'update')) {

throw new Error('Insufficient permissions to assign roles');

}

// Verificar jerarquía de roles

if (!this.canAssignRole(assigner.role, role)) {

throw new Error('Cannot assign higher or equal role');

}

await prisma.propertyOwner.update({

where: { id: userId },

data: {

role,

permissions: this.getDefaultPermissions(role),

updatedBy: assignedBy

}

});

// Log de auditoría

await this.logRoleChange(userId, role, assignedBy);

}

async updatePermissions(userId: string, permissions: Partial<UserPermissions>, updatedBy: string): Promise<void> {

const user = await prisma.propertyOwner.findUnique({

where: { id: userId }

});

if (!user) {

throw new Error('User not found');

}

const updatedPermissions = {

...user.permissions,

...permissions

};

await prisma.propertyOwner.update({

where: { id: userId },

data: {

permissions: updatedPermissions,

updatedBy

}

});

await this.logPermissionChange(userId, permissions, updatedBy);

}

async getUsersWithFilters(filters: UserFilters): Promise<PaginatedUsers> {

const {

page = 1,

limit = 20,

role,

status,

search,

sortBy = 'createdAt',

sortOrder = 'desc'

} = filters;

const where: any = {};

if (role) where.role = role;

if (status) where.accountStatus = status;

if (search) {

where.OR = [

{ contactName: { contains: search, mode: 'insensitive' } },

{ email: { contains: search, mode: 'insensitive' } },

{ companyName: { contains: search, mode: 'insensitive' } }

];

}

const [users, total] = await Promise.all([

prisma.propertyOwner.findMany({

where,

skip: (page - 1) \* limit,

take: limit,

orderBy: { [sortBy]: sortOrder },

include: {

properties: { select: { id: true, name: true } },

\_count: { select: { properties: true } }

}

}),

prisma.propertyOwner.count({ where })

]);

return {

users,

pagination: {

page,

limit,

total,

totalPages: Math.ceil(total / limit)

}

};

}

async deactivateUser(userId: string, reason: string, deactivatedBy: string): Promise<void> {

await prisma.propertyOwner.update({

where: { id: userId },

data: {

accountStatus: 'deactivated',

deactivatedAt: new Date(),

deactivatedBy,

deactivationReason: reason

}

});

// Suspender suscripciones activas

const activeSubscriptions = await prisma.subscription.findMany({

where: {

propertyOwnerId: userId,

status: 'active'

}

});

for (const subscription of activeSubscriptions) {

await this.paymentService.cancelSubscription(subscription.id, reason);

}

// Notificar al usuario

await this.notificationService.sendAccountDeactivationEmail(userId, reason);

// Log de auditoría

await this.logAccountAction(userId, 'deactivated', { reason, deactivatedBy });

}

async impersonateUser(targetUserId: string, adminUserId: string): Promise<ImpersonationSession> {

const admin = await this.getUserWithPermissions(adminUserId);

if (!this.hasPermission(admin, 'users', 'update')) {

throw new Error('Insufficient permissions for impersonation');

}

const targetUser = await prisma.propertyOwner.findUnique({

where: { id: targetUserId }

});

if (!targetUser) {

throw new Error('Target user not found');

}

// Crear sesión de impersonación

const session = await prisma.impersonationSession.create({

data: {

adminUserId,

targetUserId,

startedAt: new Date(),

expiresAt: new Date(Date.now() + 2 \* 60 \* 60 \* 1000) // 2 horas

}

});

// Log de auditoría

await this.logImpersonation(adminUserId, targetUserId, 'started');

return session;

}

async getBulkActions(): Promise<BulkAction[]> {

return [

{

id: 'activate\_users',

name: 'Activar usuarios',

description: 'Activar usuarios seleccionados',

requiredPermission: 'users.update',

confirmationRequired: false

},

{

id: 'deactivate\_users',

name: 'Desactivar usuarios',

description: 'Desactivar usuarios seleccionados',

requiredPermission: 'users.update',

confirmationRequired: true

},

{

id: 'reset\_passwords',

name: 'Reset contraseñas',

description: 'Enviar emails de reset de contraseña',

requiredPermission: 'users.update',

confirmationRequired: true

},

{

id: 'export\_users',

name: 'Exportar usuarios',

description: 'Exportar datos de usuarios seleccionados',

requiredPermission: 'users.read',

confirmationRequired: false

}

];

}

async executeBulkAction(

actionId: string,

userIds: string[],

executedBy: string,

parameters?: Record<string, any>

): Promise<BulkActionResult> {

const results: BulkActionResult = {

successful: [],

failed: [],

total: userIds.length

};

for (const userId of userIds) {

try {

switch (actionId) {

case 'activate\_users':

await this.activateUser(userId, executedBy);

results.successful.push(userId);

break;

case 'deactivate\_users':

await this.deactivateUser(userId, parameters?.reason || 'Bulk action', executedBy);

results.successful.push(userId);

break;

case 'reset\_passwords':

await this.sendPasswordResetEmail(userId);

results.successful.push(userId);

break;

default:

results.failed.push({ userId, error: 'Unknown action' });

}

} catch (error) {

results.failed.push({

userId,

error: error instanceof Error ? error.message : 'Unknown error'

});

}

}

// Log de auditoría para acción masiva

await this.logBulkAction(actionId, userIds, executedBy, results);

return results;

}

}

// Componente de gestión de usuarios

// src/components/admin/UserManagement.tsx

export const UserManagement: React.FC = () => {

const [users, setUsers] = useState<PaginatedUsers | null>(null);

const [filters, setFilters] = useState<UserFilters>({

page: 1,

limit: 20,

sortBy: 'createdAt',

sortOrder: 'desc'

});

const [selectedUsers, setSelectedUsers] = useState<string[]>([]);

const [bulkActionDialog, setBulkActionDialog] = useState<{

open: boolean;

action?: BulkAction;

}>({ open: false });

useEffect(() => {

loadUsers();

}, [filters]);

const loadUsers = async () => {

try {

const userData = await userManagementService.getUsersWithFilters(filters);

setUsers(userData);

} catch (error) {

console.error('Error loading users:', error);

}

};

const handleBulkAction = async (actionId: string, parameters?: Record<string, any>) => {

try {

const result = await userManagementService.executeBulkAction(

actionId,

selectedUsers,

currentUser.id,

parameters

);

// Mostrar resultados

if (result.failed.length > 0) {

// Mostrar errores

console.error('Some actions failed:', result.failed);

}

// Recargar datos

await loadUsers();

setSelectedUsers([]);

setBulkActionDialog({ open: false });

} catch (error) {

console.error('Bulk action failed:', error);

}

};

return (

<Box sx={{ p: 3 }}>

<Typography variant="h4" sx={{ mb: 3 }}>Gestión de Usuarios</Typography>

{/\* Filtros y búsqueda \*/}

<Card sx={{ mb: 3 }}>

<CardContent>

<Grid container spacing={2} alignItems="center">

<Grid item xs={12} md={3}>

<TextField

fullWidth

label="Buscar usuarios"

value={filters.search || ''}

onChange={(e) => setFilters({ ...filters, search: e.target.value, page: 1 })}

placeholder="Nombre, email, empresa..."

/>

</Grid>

<Grid item xs={12} md={2}>

<FormControl fullWidth>

<InputLabel>Rol</InputLabel>

<Select

value={filters.role || ''}

label="Rol"

onChange={(e) => setFilters({ ...filters, role: e.target.value || undefined, page: 1 })}

>

<MenuItem value="">Todos</MenuItem>

<MenuItem value="owner">Propietario</MenuItem>

<MenuItem value="manager">Gestor</MenuItem>

<MenuItem value="staff">Staff</MenuItem>

<MenuItem value="admin">Admin</MenuItem>

</Select>

</FormControl>

</Grid>

<Grid item xs={12} md={2}>

<FormControl fullWidth>

<InputLabel>Estado</InputLabel>

<Select

value={filters.status || ''}

label="Estado"

onChange={(e) => setFilters({ ...filters, status: e.target.value || undefined, page: 1 })}

>

<MenuItem value="">Todos</MenuItem>

<MenuItem value="active">Activo</MenuItem>

<MenuItem value="pending\_verification">Pendiente</MenuItem>

<MenuItem value="deactivated">Desactivado</MenuItem>

<MenuItem value="suspended">Suspendido</MenuItem>

</Select>

</FormControl>

</Grid>

<Grid item xs={12} md={2}>

<Button

variant="contained"

startIcon={<Add />}

onClick={() => setCreateUserDialog(true)}

>

Nuevo Usuario

</Button>

</Grid>

<Grid item xs={12} md={3}>

<Box sx={{ display: 'flex', gap: 1 }}>

<Button

variant="outlined"

startIcon={<Download />}

onClick={() => handleExport()}

>

Exportar

</Button>

{selectedUsers.length > 0 && (

<Button

variant="outlined"

startIcon={<MoreVert />}

onClick={() => setBulkActionDialog({ open: true })}

>

Acciones ({selectedUsers.length})

</Button>

)}

</Box>

</Grid>

</Grid>

</CardContent>

</Card>

{/\* Tabla de usuarios \*/}

<Card>

<CardContent>

<TableContainer>

<Table>

<TableHead>

<TableRow>

<TableCell padding="checkbox">

<Checkbox

checked={selectedUsers.length === users?.users.length && users.users.length > 0}

indeterminate={selectedUsers.length > 0 && selectedUsers.length < (users?.users.length || 0)}

onChange={(e) => {

if (e.target.checked) {

setSelectedUsers(users?.users.map(u => u.id) || []);

} else {

setSelectedUsers([]);

}

}}

/>

</TableCell>

<TableCell>Usuario</TableCell>

<TableCell>Rol</TableCell>

<TableCell>Propiedades</TableCell>

<TableCell>Estado</TableCell>

<TableCell>Último acceso</TableCell>

<TableCell>Acciones</TableCell>

</TableRow>

</TableHead>

<TableBody>

{users?.users.map((user) => (

<TableRow key={user.id}>

<TableCell padding="checkbox">

<Checkbox

checked={selectedUsers.includes(user.id)}

onChange={(e) => {

if (e.target.checked) {

setSelectedUsers([...selectedUsers, user.id]);

} else {

setSelectedUsers(selectedUsers.filter(id => id !== user.id));

}

}}

/>

</TableCell>

<TableCell>

<Box sx={{ display: 'flex', alignItems: 'center', gap: 2 }}>

<Avatar sx={{ width: 40, height: 40 }}>

{user.contactName.charAt(0)}

</Avatar>

<Box>

<Typography variant="body2" fontWeight="bold">

{user.contactName}

</Typography>

<Typography variant="caption" color="text.secondary">

{user.email}

</Typography>

{user.companyName && (

<Typography variant="caption" display="block" color="text.secondary">

{user.companyName}

</Typography>

)}

</Box>

</Box>

</TableCell>

<TableCell>

<Chip

label={user.role}

size="small"

color={getRoleColor(user.role)}

/>

</TableCell>

<TableCell>

<Box sx={{ display: 'flex', alignItems: 'center', gap: 1 }}>

<Typography variant="body2">

{user.\_count.properties}

</Typography>

{user.properties.length > 0 && (

<Tooltip title={user.properties.map(p => p.name).join(', ')}>

<IconButton size="small">

<Info fontSize="small" />

</IconButton>

</Tooltip>

)}

</Box>

</TableCell>

<TableCell>

<Chip

label={getStatusLabel(user.accountStatus)}

size="small"

color={getStatusColor(user.accountStatus)}

/>

</TableCell>

<TableCell>

<Typography variant="body2">

{user.lastLoginAt ?

formatDistanceToNow(new Date(user.lastLoginAt), { addSuffix: true, locale: es }) :

'Nunca'

}

</Typography>

</TableCell>

<TableCell>

<UserActionsMenu user={user} onAction={handleUserAction} />

</TableCell>

</TableRow>

))}

</TableBody>

</Table>

</TableContainer>

{users && (

<Box sx={{ display: 'flex', justifyContent: 'center', mt: 3 }}>

<Pagination

count={users.pagination.totalPages}

page={users.pagination.page}

onChange={(\_, page) => setFilters({ ...filters, page })}

color="primary"

/>

</Box>

)}

</CardContent>

</Card>

{/\* Dialog de acciones masivas \*/}

<BulkActionsDialog

open={bulkActionDialog.open}

selectedCount={selectedUsers.length}

onClose={() => setBulkActionDialog({ open: false })}

onExecute={handleBulkAction}

/>

</Box>

);

};