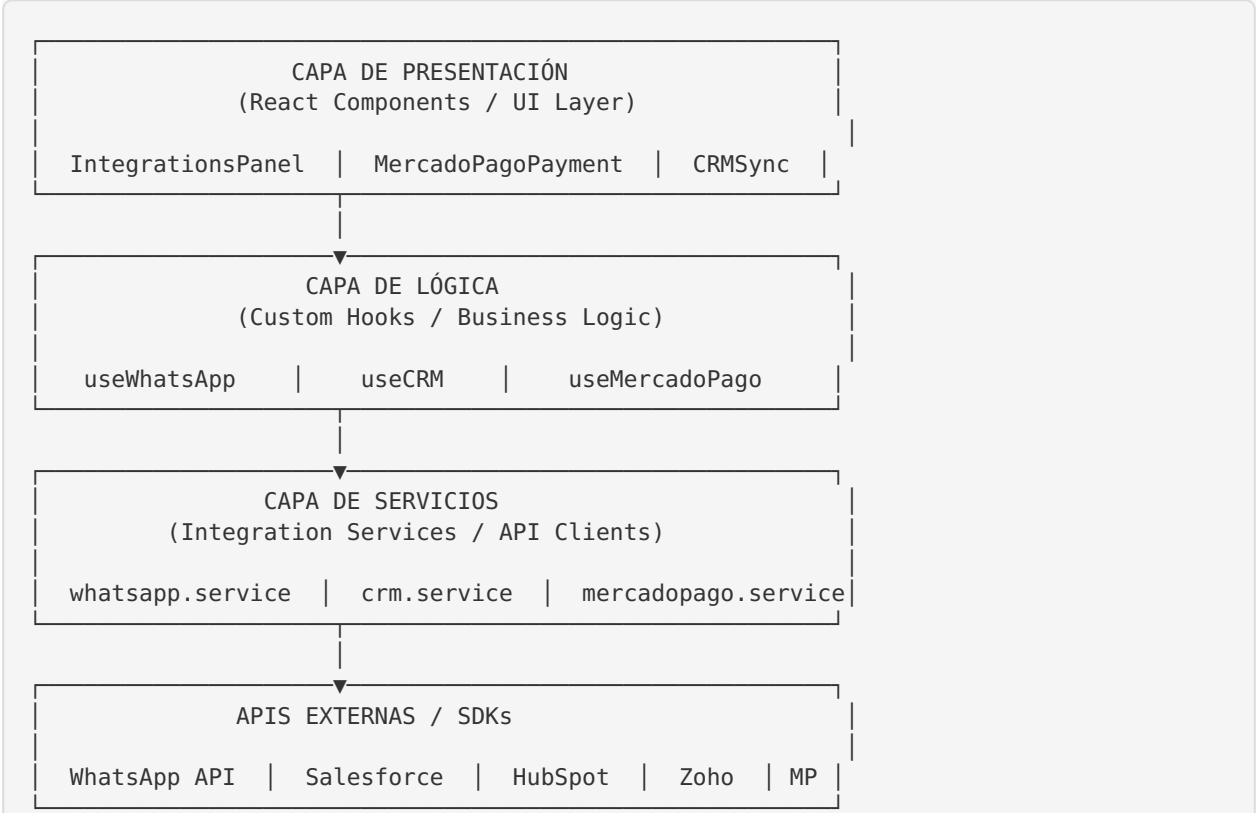
Manual Técnico de Integraciones

Índice

Arquitectura Técnica

Capas de la Aplicación



Principios de Diseño

1. **Separación de Responsabilidades**
   - Servicios: Comunicación con APIs externas

   - Hooks: Lógica de negocio y estado
   - Componentes: Presentación y UX

2. **Abstracción**
   - Los hooks abstraen la complejidad de los servicios
   - Los componentes usan hooks sin conocer detalles de implementación

3. **Reutilización**
   - Servicios singleton compartidos
   - Hooks reutilizables en múltiples componentes
   - Componentes modulares

4. **Escalabilidad**
   - Fácil agregar nuevos CRMs
   - Fácil extender funcionalidades
   - Patrones consistentes

Servicios de Integración

Estructura de un Servicio

Todos los servicios siguen esta estructura:

```
class ServiceName {
  constructor() {
    // Inicializar configuración desde variables de entorno
    this.config = this.loadConfig();
  }

  isConfigured() {
    // Verificar si el servicio está correctamente configurado
    return {
      configured: boolean,
      message: string
    };
  }

  async makeRequest(method, endpoint, data) {
    // Método genérico para hacer requests HTTP
    // Manejo centralizado de errores
    // Logging automático
  }

  // Métodos específicos del servicio...
}

// Exportar instancia singleton
export default new ServiceName();
```

WhatsApp Service

Archivo: `src/services/integrations/whatsapp.service.js`

Responsabilidades:
- Envío de mensajes de texto

- Gestión de templates de mensajes
- Envío masivo de mensajes

Métodos Principales:

```
// Enviar mensaje simple
await whatsappService.sendMessage(phoneNumber, message);

// Enviar usando template predefinido
await whatsappService.sendWelcomeMessage(phoneNumber, userName);

// Envío masivo
await whatsappService.sendBulkMessage(phoneNumbers, message);
```

Configuración Requerida:

```
VITE_WHATSAPP_ACCESS_TOKEN
VITE_WHATSAPP_PHONE_NUMBER_ID
VITE_WHATSAPP_BUSINESS_ACCOUNT_ID
```

CRM Service

Archivo: `src/services/integrations/crm/crm.service.js`

Responsabilidades:
- Interfaz unificada para múltiples CRMs
- Detección automática del CRM activo
- Delegación a adaptadores específicos

Adaptadores:
- `salesforce.service.js` - Salesforce CRM
- `hubspot.service.js` - HubSpot CRM
- `zoho.service.js` - Zoho CRM

Métodos Principales:

```
// Sincronización
await crmService.syncDebtor(debtorData);
await crmService.syncDebtors(debtorsArray);

// Importación
await crmService.getDebtors(filters);
await crmService.importDebts(filters);

// Actualización
await crmService.updateDebtStatus(debtId, updateData);

// Sincronización completa/incremental
await crmService.fullSync(options);
await crmService.incrementalSync(since);
```

Patrón Adapter:

```
class CRMService {
  constructor() {
    this.adapters = {
      salesforce: salesforceService,
      hubspot: hubspotService,
      zoho: zohoService
    };
  }

  getActiveAdapter() {
    return this.adapters[this.activeCRM];
  }

  async syncDebtor(data) {
    const adapter = this.getActiveAdapter();
    return await adapter.syncContact(data);
  }
}
```

Mercado Pago Service

Archivo: `src/services/integrations/mercadopago.service.js`

Responsabilidades:
- Crear preferencias de pago
- Procesar webhooks
- Gestionar transacciones
- Otorgar incentivos automáticamente

Métodos Principales:

```
// Crear preferencia de pago
await mercadoPagoService.createPaymentPreference(paymentData);

// Procesar webhook
await mercadoPagoService.processWebhook(webhookData);

// Obtener información de pago
await mercadoPagoService.getPayment(paymentId);

// Crear reembolso
await mercadoPagoService.createRefund(paymentId, amount);
```

Flujo de Webhook:

```
async processWebhook(webhookData) {
  // 1. Validar webhook
  if (webhookData.type !== 'payment') return;

  // 2. Obtener información del pago
  const payment = await this.getPayment(webhookData.data.id);

  // 3. Guardar transacción
  await this.saveTransaction(payment);

  // 4. Si está aprobado, procesar
  if (payment.status === 'approved') {
    await this.processApprovedPayment(payment);
  }
}

async processApprovedPayment(payment) {
  // 1. Actualizar deuda
  await this.updateDebt(payment.metadata.debt_id);

  // 2. Otorgar incentivo
  await this.grantPaymentIncentive(...);

  // 3. Registrar en historial
  await this.logPaymentHistory(...);

  // 4. Notificar al usuario (opcional)
  // await whatsappService.sendPaymentConfirmation(...);
}
```

Hooks Personalizados

Estructura de un Hook

```
export const useServiceName = () => {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  const { user } = useAuth();
  const { addNotification } = useNotification();

  const methodName = useCallback(async (params) => {
    setLoading(true);
    setError(null);

    try {
      const result = await service.method(params);

      if (result.success) {
        addNotification({
          type: 'success',
          message: 'Operación exitosa'
        });
      }

      return result;
    } catch (err) {
      setError(err.message);
      addNotification({
        type: 'error',
        message: err.message
      });
      return { success: false, error: err.message };
    } finally {
      setLoading(false);
    }
  }, [addNotification]);

  return {
    loading,
    error,
    methodName
  };
};
```

useWhatsApp

Archivo: `src/hooks/integrations/useWhatsApp.js`

Estado:

```
{
  loading: boolean,
  error: string | null
}
```

Métodos:

```
{
  isConfigured,
  sendMessage,
  sendWelcome,
  sendPaymentReminder,
  sendAgreementConfirmation,
  sendPaymentConfirmation,
  sendIncentiveAlert,
  sendNewOfferNotification,
  sendOfferExpiringAlert,
  sendAchievementNotification,
  sendLevelUpNotification,
  sendBulkMessages
}
```

useCRM

Archivo: `src/hooks/integrations/useCRM.js`

Estado:

```
{
  loading: boolean,
  error: string | null,
  activeCRM: string | null,
  availableCRMs: Array<{name, configured, active}>
}
```

Métodos:

```
{
  changeCRM,
  syncDebtor,
  syncDebtors,
  getDebtors,
  getDebtor,
  importDebts,
  updateDebtStatus,
  logActivity,
  logPayment,
  createPaymentAgreement,
  updatePaymentAgreement,
  getDebtorHistory,
  searchDebtors,
  fullSync,
  incrementalSync
}
```

useMercadoPago

Archivo: `src/hooks/integrations/useMercadoPago.js`

Estado:

```
{
  loading: boolean,
  error: string | null,
  isConfigured: boolean
}
```

Métodos:

```
{
  createPaymentPreference,
  createInstallmentPayment,
  getPayment,
  processWebhook,
  createRefund,
  searchPayments,
  getPaymentStats,
  // Helpers
  createDebtPayment,
  createInstallmentPaymentForAgreement,
  checkPaymentStatus
}
```

Componentes UI

IntegrationsPanel

Propósito: Panel de administración para ver estado de integraciones

Props: Ninguno

Uso:

```
import { IntegrationsPanel } from './components/integrations';

<IntegrationsPanel />
```

Características:
- Muestra estado de todas las integraciones
- Indica cuáles están configuradas
- Proporciona links a documentación
- Botones de acción rápida

MercadoPagoPayment

Propósito: Componente de pago con Mercado Pago

Props:

```
{
  debt: {
    id: string,
    amount: number,
    company_name: string,
    // ...
  },
  onPaymentCreated?: (result) => void,
  installment?: {
    number: number,
    total: number
  }
}
```

Uso:

```
<MercadoPagoPayment
  debt={selectedDebt}
  onPaymentCreated={(result) => {
    console.log('Pago creado:', result);
  }}
/>
```

CRMSyncStatus

Propósito: Muestra estado de sincronización con CRM

Props: Ninguno

Uso:

```
import { CRMSyncStatus } from './components/integrations';

<CRMSyncStatus />
```

Características:
- Muestra última sincronización
- Permite sincronización manual
- Muestra estadísticas de sync
- Selector de CRM activo

WhatsAppNotificationSettings

Propósito: Configuración de notificaciones WhatsApp para usuarios

Props: Ninguno

Uso:

```
<WhatsAppNotificationSettings />
```

Características:
- Input de número de teléfono
- Toggles para tipos de notificación

- Botón de prueba
- Guarda preferencias en BD

Flujos de Datos

Flujo 1: Pago de Deuda con Mercado Pago

```
[Usuario hace clic en "Pagar"]
    ↓
[useMercadoPago.createDebtPayment()]
    ↓
[mercadoPagoService.createPaymentPreference()]
    ↓
[POST /checkout/preferences a Mercado Pago API]
    ↓
[Respuesta: preferenceId, initPoint]
    ↓
[window.location.href = initPoint]
    ↓
[Usuario completa pago en Mercado Pago]
    ↓
[Mercado Pago envía webhook a nuestra URL]
    ↓
[/api/webhooks/mercadopago recibe notificación]
    ↓
[mercadoPagoService.processWebhook()]
    ↓
[getPayment() para obtener detalles]
    ↓
[saveTransaction() en Supabase]
    ↓
[Si approved: processApprovedPayment()]
    ↓
[Actualizar deuda en BD]
    ↓
[grantPaymentIncentive()]
    ↓
[logPaymentHistory()]
    ↓
[Opcional: sendPaymentConfirmation() vía WhatsApp]
    ↓
[Opcional: logPayment() en CRM]
```

Flujo 2: Sincronización CRM

```
[Admin hace clic en "Sincronizar"]
    ↓
[useCRM.fullSync()]
    ↓
[crmService.fullSync()]
    ↓
[getActiveAdapter()]
    ↓
[adapter.getContacts()]
    ↓
[GET /contacts desde API del CRM]
    ↓
[Mapear campos del CRM a formato de plataforma]
    ↓
[Guardar en Supabase (upsert)]
    ↓
[adapter.importDebts()]
    ↓
[GET /debts desde API del CRM]
    ↓
[Mapear y guardar en Supabase]
    ↓
[Retornar resumen: {debtors: X, debts: Y}]
    ↓
[Actualizar UI con notificación]
```

Flujo 3: Envío de Notificación WhatsApp

```
[Evento: Pago aprobado]
    ↓
[useWhatsApp.sendPaymentConfirmation()]
    ↓
[whatsappService.sendPaymentConfirmation()]
    ↓
[Generar mensaje desde template]
    ↓
[sendMessage(phone, message)]
    ↓
[POST /{phone_number_id}/messages a WhatsApp API]
    ↓
[Respuesta: messageId]
    ↓
[Retornar resultado]
    ↓
[Hook actualiza estado y muestra notificación]
```

Manejo de Errores

Estrategia de 3 Capas

1. **Capa de Servicio**: Captura errores de API
2. **Capa de Hook**: Maneja errores y actualiza estado
3. **Capa de Componente**: Muestra errores al usuario

Ejemplo Completo

```
// 1. SERVICIO
async makeRequest(method, endpoint, data) {
  try {
    const response = await axios({...});
    return {
      success: true,
      data: response.data
    };
  } catch (error) {
    console.error('❌ Error en API:', error);
    return {
      success: false,
      error: error.response?.data || error.message
    };
  }
}

// 2. HOOK
const sendMessage = useCallback(async (phone, message) => {
  setLoading(true);
  setError(null);

  try {
    const result = await whatsappService.sendMessage(phone, message);

    if (!result.success) {
      throw new Error(result.error);
    }

    addNotification({
      type: 'success',
      message: 'Mensaje enviado'
    });

    return result;
  } catch (err) {
    setError(err.message);
    addNotification({
      type: 'error',
      message: err.message
    });
    return { success: false, error: err.message };
  } finally {
    setLoading(false);
  }
}, []);

// 3. COMPONENTE
function MyComponent() {
  const { sendMessage, loading, error } = useWhatsApp();

  const handleSend = async () => {
    const result = await sendMessage(phone, message);

    if (result.success) {
      // Éxito
    } else {
      // Error ya fue manejado por el hook
    }
  };

  return (
```

```
    <div>
      {error && <ErrorAlert message={error} />}
      <button onClick={handleSend} disabled={loading}>
        {loading ? 'Enviando...' : 'Enviar'}
      </button>
    </div>
  );
}
```

Tipos de Errores

```
// Error de configuración
if (!this.accessToken) {
  throw new Error('WhatsApp no está configurado');
}

// Error de validación
if (!phoneNumber || phoneNumber.length < 10) {
  throw new Error('Número de teléfono inválido');
}

// Error de API
if (error.response?.status === 401) {
  throw new Error('Token de acceso inválido o expirado');
}

// Error de red
if (error.code === 'ECONNREFUSED') {
  throw new Error('No se puede conectar al servicio');
}
```

Logging y Monitoreo

Estrategia de Logging

```
// ✅ Log inicial de operación
console.log('🔄 Iniciando sincronización CRM...');

// ✅ Log de éxito
console.log('✅ WhatsApp enviado a', phone);

// ✅ Log de error
console.error('❌ Error al sincronizar:', error);

// ✅ Log de advertencia
console.warn('⚠️ Token próximo a expirar');

// ✅ Log con datos (solo en desarrollo)
if (process.env.NODE_ENV === 'development') {
  console.log('📊 Datos:', data);
}
```

Niveles de Log

```
const LOG_LEVELS = {
  ERROR: '❌',
  WARN: '⚠️',
  INFO: 'ℹ️',
  SUCCESS: '✅',
  DEBUG: '🐛'
};

function log(level, message, data = null) {
  const timestamp = new Date().toISOString();
  const logMessage = `${LOG_LEVELS[level]} [${timestamp}] ${message}`;

  console.log(logMessage);

  if (data) {
    console.log('Data:', data);
  }

  // Enviar a servicio de monitoreo (opcional)
  // sendToMonitoring(level, message, data);
}
```

Monitoreo de Métricas

```
// Tiempo de ejecución
const startTime = Date.now();
await operation();
const duration = Date.now() - startTime;
console.log(`⏱️ Operación completada en ${duration}ms`);

// Contadores
let successCount = 0;
let errorCount = 0;

results.forEach(r => {
  r.success ? successCount++ : errorCount++;
});

console.log(`📊 Resultados: ${successCount} éxitos, ${errorCount} errores`);
```

Testing

Unit Tests

```
// whatsapp.service.test.js
describe('WhatsAppService', () => {
  test('isConfigured retorna true cuando está configurado', () => {
    const result = whatsappService.isConfigured();
    expect(result.configured).toBe(true);
  });

  test('sendMessage envía mensaje correctamente', async () => {
    const result = await whatsappService.sendMessage(
      '56912345678',
      'Test message'
    );

    expect(result.success).toBe(true);
    expect(result.messageId).toBeDefined();
  });
});
```

Integration Tests

```
// mercadopago.integration.test.js
describe('Mercado Pago Integration', () => {
  test('Flujo completo de pago', async () => {
    // 1. Crear preferencia
    const preference = await mercadoPagoService.createPaymentPreference({
      amount: 10000,
      description: 'Test payment'
    });

    expect(preference.success).toBe(true);

    // 2. Simular webhook
    const webhook = {
      type: 'payment',
      data: { id: 'test-payment-123' }
    };

    const result = await mercadoPagoService.processWebhook(webhook);
    expect(result.success).toBe(true);
  });
});
```

E2E Tests

```
// payment-flow.e2e.test.js
describe('Payment Flow E2E', () => {
  test('Usuario completa pago exitosamente', async () => {
    // 1. Login
    await page.goto('/login');
    await page.fill('[name=email]', 'test@example.com');
    await page.fill('[name=password]', 'password');
    await page.click('button[type=submit]');

    // 2. Seleccionar deuda
    await page.click('[data-testid=debt-card]');

    // 3. Iniciar pago
    await page.click('[data-testid=pay-button]');

    // 4. Verificar redirección a Mercado Pago
    await page.waitForURL(/mercadopago\.com/);
  });
});
```

Optimización y Performance

1. Lazy Loading

```
// Cargar servicios solo cuando se necesitan
const loadWhatsAppService = async () => {
  const { default: service } = await import('./services/integrations/whats-
app.service');
  return service;
};
```

2. Caching

```
// Cache de resultados
const cache = new Map();

async function getCachedData(key, fetcher, ttl = 5 * 60 * 1000) {
  const cached = cache.get(key);

  if (cached && Date.now() - cached.timestamp < ttl) {
    return cached.data;
  }

  const data = await fetcher();
  cache.set(key, { data, timestamp: Date.now() });

  return data;
}

// Uso
const debtors = await getCachedData('crm-debtors', () => {
  return crmService.getDebtors();
});
```

3. Batch Operations

```
// Procesar en lotes
async function processBatch(items, batchSize = 100) {
  const results = [];

  for (let i = 0; i < items.length; i += batchSize) {
    const batch = items.slice(i, i + batchSize);
    const batchResults = await Promise.all(
      batch.map(item => processItem(item))
    );
    results.push(...batchResults);

    // Pausa entre lotes
    await sleep(1000);
  }

  return results;
}
```

4. Debouncing

```
// Debounce para búsquedas
import { debounce } from 'lodash';

const debouncedSearch = debounce(async (query) => {
  const results = await crmService.searchDebtors(query);
  setSearchResults(results);
}, 300);
```

Seguridad

1. Validación de Entrada

```
function validatePhoneNumber(phone) {
  // Solo dígitos
  const cleaned = phone.replace(/\D/g, '');

  // Longitud mínima
  if (cleaned.length < 10) {
    throw new Error('Número inválido');
  }

  return cleaned;
}
```

2. Sanitización

```
function sanitizeInput(input) {
  return input
    .trim()
    .replace(/<script[^>]*>.*?<\/script>/gi, '')
    .replace(/[<>]/g, '');
}
```

3. Rate Limiting

```
const rateLimiter = {
  requests: new Map(),
  limit: 10, // requests por minuto

  async checkLimit(key) {
    const now = Date.now();
    const requests = this.requests.get(key) || [];

    // Filtrar requests del último minuto
    const recentRequests = requests.filter(
      time => now - time < 60000
    );

    if (recentRequests.length >= this.limit) {
      throw new Error('Rate limit exceeded');
    }

    recentRequests.push(now);
    this.requests.set(key, recentRequests);
  }
};
```

4. Validación de Webhooks

```
function validateWebhookSignature(payload, signature, secret) {
  const hmac = crypto
    .createHmac('sha256', secret)
    .update(JSON.stringify(payload))
    .digest('hex');

  return hmac === signature;
}
```

Mejores Prácticas

1. Código Limpio

```
// ❌ Mal
async function f(d) {
  const r = await api.get('/data', d);
  if (r.ok) return r.data;
  throw new Error(r.error);
}

// ✅ Bien
async function fetchDebtorData(debtorId) {
  const response = await crmService.getDebtor(debtorId);

  if (response.success) {
    return response.data;
  }

  throw new Error(`Failed to fetch debtor: ${response.error}`);
}
```

2. Documentación

```
/**
 * Sincroniza un deudor con el CRM activo
 *
 * @param {Object} debtorData - Datos del deudor
 * @param {string} debtorData.email - Email del deudor
 * @param {string} debtorData.name - Nombre completo
 * @param {number} debtorData.totalDebt - Deuda total
 * @returns {Promise<{success: boolean, contactId?: string, error?: string}>}
 *
 * @example
 * const result = await syncDebtor({
 *   email: 'juan@example.com',
 *   name: 'Juan Pérez',
 *   totalDebt: 500000
 * });
 */
async function syncDebtor(debtorData) {
  // ...
}
```

3. Manejo de Promises

```
// ✅ Usar Promise.allSettled para operaciones paralelas
const results = await Promise.allSettled([
  sendWhatsApp(phone1, message),
  sendWhatsApp(phone2, message),
  sendWhatsApp(phone3, message)
]);

// Procesar resultados
results.forEach((result, index) => {
  if (result.status === 'fulfilled') {
    console.log(`✅ Mensaje ${index + 1} enviado`);
  } else {
    console.error(`❌ Mensaje ${index + 1} falló:`, result.reason);
  }
});
```

Versión: 1.0.0
Última actualización: Octubre 2025