AI Broker Service - Ejemplos de Uso

El AI Broker Service es el punto central para todas las solicitudes de IA en la plataforma COMODÍN IA. Aquí se muestran ejemplos de cómo usar el servicio desde diferentes módulos del sistema.

Configuración Inicial

```
import { aiBroker, AIBrokerClient } from '@/lib/ai-broker-client';

// Usar la instancia global
const client = aiBroker;

// O crear una instancia personalizada
const customClient = new AIBrokerClient({
  baseUrl: '/api/ai/broker'
});
```

Ejemplos Básicos

1. Respuesta de Chat Automática

```
// Desde un módulo CRM
async function handleCustomerQuestion(organizationId: string, customerMessage:
string) {
  try {
    const result = await aiBroker.chat(
      organizationId,
      `Cliente pregunta: "${customerMessage}". Genera una respuesta profesional y
útil.`,
      {
        userId: 'agent-123',
        userName: 'María González',
        temperature: 0.7,
        maxTokens: 200
      }
    );

    if (result.success) {
      console.log('Respuesta AI:', result.response);
      console.log('Costo:', result.cost?.clientCost);
      console.log('Transacción:', result.transactionId);

      // Usar la respuesta en el CRM
      return {
        response: result.response,
        cost: result.cost?.clientCost,
        success: true
      };
    } else {
      // Manejar error (ej. saldo insuficiente)
      console.error('Error AI:', result.error);
      return {
        success: false,
        error: result.error
      };
    }
  } catch (error) {
    console.error('Error de conexión:', error);
    return {
      success: false,
      error: 'Error de conexión con IA'
    };
  }
}
```

2. Análisis de Sentimientos

```
async function analyzeCustomerSentiment(organizationId: string, conversation: string)
{
  const result = await aiBroker.analyzeSentiment(
    organizationId,
    conversation,
    {
      userId: 'system',
      userName: 'Sistema Automático',
      metadata: {
        source: 'whatsapp',
        conversation_id: 'conv-123'
      }
    }
  );

  if (result.success) {
    // Parsear la respuesta para extraer el sentimiento
    const sentiment = extractSentimentFromResponse(result.response);

    return {
      sentiment,
      confidence: 0.85, // Esto vendría del análisis
      cost: result.cost?.clientCost,
      processing_time: result.usage?.processingTime
    };
  }

  return null;
}
```

3. Generación de Contenido

```
async function generateEmailResponse(organizationId: string, customerEmail: string) {
  const prompt = `
    Genera una respuesta profesional por email para el siguiente mensaje del cliente:
    "${customerEmail}"

    La respuesta debe ser:
    - Profesional y cortés
    - Útil y específica
    - En español
    - Con un tono amigable pero formal
  `;

  const result = await aiBroker.generateContent(
    organizationId,
    prompt,
    {
      userId: 'email-system',
      userName: 'Sistema de Email',
      temperature: 0.8,
      maxTokens: 300,
      metadata: {
        channel: 'email',
        type: 'customer_support'
      }
    }
  );

  return result;
}
```

4. Traducción Automática

```
async function translateMessage(
  organizationId: string,
  message: string,
  targetLanguage: string
) {
  const result = await aiBroker.translate(
    organizationId,
    message,
    targetLanguage,
    {
      userId: 'translation-service',
      userName: 'Servicio de Traducción',
      metadata: {
        original_language: 'es',
        target_language: targetLanguage
      }
    }
  );

  if (result.success) {
    return {
      translated_text: result.response,
      cost: result.cost?.clientCost,
      characters_processed: message.length
    };
  }

  return null;
}
```

5. Resumen de Conversación

```typescript
async function summarizeConversation(organizationId: string, messages: string[]) {
  const fullConversation = messages.join('\n---\n');

  const result = await aiBroker.summarize(
    organizationId,
    fullConversation,
    {
      userId: 'summary-system',
      userName: 'Sistema de Resumen',
      maxTokens: 250,
      metadata: {
        message_count: messages.length,
        conversation_length: fullConversation.length
      }
    }
  );

  if (result.success) {
    return {
      summary: result.response,
      original_length: fullConversation.length,
      summary_length: result.response?.length || 0,
      compression_ratio: (result.response?.length || 0) / fullConversation.length,
      cost: result.cost?.clientCost
    };
  }

  return null;
}
```

Manejo de Errores

```
async function handleAIRequestWithErrorHandling(organizationId: string, prompt:
string) {
  try {
    const result = await aiBroker.sendRequest(organizationId, prompt);

    if (!result.success) {
      // Verificar tipo específico de error
      if (AIBrokerUtils.isInsufficientBalanceError(result.error || '')) {
        // Redirigir a página de recarga de billetera
        return {
          error: 'insufficient_balance',
          message: 'Saldo insuficiente. Por favor recarga tu billetera.',
          redirect_url: '/wallet/recharge'
        };
      }

      return {
        error: 'ai_error',
        message: result.error || 'Error desconocido'
      };
    }

    return {
      success: true,
      data: result
    };

  } catch (error) {
    return {
      error: 'connection_error',
      message: 'No se pudo conectar con el servicio de IA'
    };
  }
}
```

Verificación de Saldo

```
async function checkBalanceBeforeAIRequest(organizationId: string, prompt: string) {
  // Estimar costo antes de la solicitud
  const estimatedTokens = AIBrokerUtils.estimateTokens(prompt);
  const estimatedCost = estimatedTokens * 0.00002 * 1.30; // Con margen

  // Verificar si se puede usar IA (implementación futura)
  const canUse = await AIBrokerUtils.canUseAI(organizationId, prompt);

  if (!canUse) {
    return {
      can_proceed: false,
      estimated_cost: estimatedCost,
      message: 'Saldo insuficiente para procesar esta solicitud'
    };
  }

  return {
    can_proceed: true,
    estimated_cost: estimatedCost
  };
}
```

Uso desde Componentes React

```
// Hook personalizado para usar el AI Broker
import { useState } from 'react';

export function useAIBroker() {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const sendRequest = async (organizationId: string, prompt: string, options = {}) =>
{
    setLoading(true);
    setError(null);

    try {
      const result = await aiBroker.sendRequest(organizationId, prompt, options);

      if (!result.success) {
        setError(result.error || 'Error desconocido');
        return null;
      }

      return result;
    } catch (err: any) {
      setError(err.message || 'Error de conexión');
      return null;
    } finally {
      setLoading(false);
    }
  };

  return {
    sendRequest,
    loading,
    error,
    clearError: () => setError(null)
  };
}

// Uso en componente
function ChatComponent({ organizationId }: { organizationId: string }) {
  const { sendRequest, loading, error } = useAIBroker();

  const handleSendMessage = async (message: string) => {
    const result = await sendRequest(organizationId, message, {
      usageType: AIUsageType.CHAT_RESPONSE,
      temperature: 0.7
    });

    if (result) {
      console.log('Respuesta:', result.response);
      console.log('Costo:', AIBrokerUtils.formatCost(result.cost?.clientCost || 0));
    }
  };

  return (
    <div>
      {loading && <div>Procesando...</div>}
      {error && <div className="error">Error: {error}</div>}
      {/* UI del chat */}
    </div>
  );
}
```

Obtener Estadísticas

```
async function getAIUsageStats() {
  try {
    const stats = await aiBroker.getStats();

    if (stats.success && stats.data) {
      return {
        total_requests: stats.data.totalRequests,
        total_cost: stats.data.totalCost,
        avg_response_time: stats.data.averageResponseTime,
        top_providers: stats.data.topProviders,
        formatted_cost: AIBrokerUtils.formatCost(stats.data.totalCost)
      };
    }
  } catch (error) {
    console.error('Error obteniendo estadísticas:', error);
  }

  return null;
}
```

Testing del AI Broker

```
async function testAIBroker(organizationId: string) {
  const testTypes = ['simple', 'chat', 'analysis', 'content', 'translation',
'summary'];
  const results = [];

  for (const testType of testTypes) {
    console.log(`Ejecutando test: ${testType}`);

    const result = await aiBroker.runTest(organizationId, testType as any);

    results.push({
      test: testType,
      success: result.success,
      processing_time: result.processingTime,
      error: result.error,
      cost: result.result?.cost?.clientCost
    });
  }

  return results;
}
```

Notas Importantes

1. **Verificación de Saldo**: El AI Broker siempre verifica el saldo antes de procesar solicitudes.
2. **Cálculo de Costos**: Los costos se calculan como `costo_proveedor * 1.30` (margen del 30%).
3. **Registro de Transacciones**: Cada uso se registra automáticamente en la base de datos.
4. **Manejo de Errores**: Siempre verificar `result.success` antes de usar la respuesta.
5. **Límites de Rate**: Los proveedores pueden tener límites de velocidad configurados.

6. **Seguridad**: Las claves API están encriptadas y solo las puede gestionar el Super Admin.