Documentación Técnica Foundation

Este es el documento que contiene toda la documentación técnica de Foundation.

Table of contents

Alcance y Limitaciones - SURA Foundation Framework

Objetivo del Proyecto

Desarrollar e implementar un framework de procesamiento centralizado (**SURA Foundation**) que permita la separación lógica y arquitectónica de los procesos de venta de seguros, proporcionando una base sólida y reutilizable para múltiples productos de la familia SURA, con especial enfoque en productos de movilidad (Motos, Autos, Viajes, Arrendamiento).

Contexto del Proyecto

SURA Foundation surge como respuesta a los problemas arquitectónicos identificados en el sistema actual, donde múltiples productos comparten código híbrido y artefactos sin separación lógica clara, generando inestabilidad, dificultades de mantenimiento y imposibilidad de medir el rendimiento real de cada proceso.

Alcance

Funcionalidades Incluidas

1. **Framework Core de Procesamiento**
   - Implementación del patrón Chain of Responsibility para procesos secuenciales
   - Sistema de observadores (Observer Pattern) para eventos de negocio
   - Adaptadores (Adapter Pattern) para integración con Vlocity y sistemas externos
   - Template Method Pattern para estandarización de procesos
2. **Gestión de Procesos por Etapas**
   - Procesamiento Pre, In y Post para cada etapa del flujo de venta
   - Manejo de eventos mediante Platform Events
   - Service Callouts integrados con lógica de reintentos configurable
   - Panel de administración centralizado para gestión de procesos
3. **Separación Lógica por Producto**
   - Procesos independientes para cada línea de negocio (Motos, Autos, Viajes, etc.)
   - Handlers específicos por producto con nomenclatura estandarizada
   - Responsabilidades claramente definidas por proceso y producto
   - Capacidad de medir desempeño individual por proceso
4. **Artefactos Reutilizables Centralizados**
   - **Conocimiento:** LeadManager y componentes base
   - **Tarificación:** Componentes de cálculo y pricing
   - **Cotización:** SURACrearNumeroPrepoliza, QuoteManager, SURAPaymentManager
   - **Emisión:** InsurancePolicyManager, SURAAccountManager, SURAOpportunityManager
   - **Legalización:** InsurancePolicyTransactionManager, SURAInvoiceListeningServiceV1
5. **Sistema de Feature Flags**

- Control dinámico de funcionalidades por ambiente
- Configuración sin deployments mediante Custom Metadata Types
- Gestión granular de permisos por funcionalidad

6. **Integración con Vlocity**
   - Adaptadores específicos para Integration Procedures
   - Reutilización del modelo estándar de Vlocity
   - Compatibilidad con Omnistudio y Salesforce Core
7. **Empaquetamiento como Managed Package**
   - Código protegido y no modificable
   - Versionado controlado (inestables: 0.x.x-NEXT, estables: 1.x-x)
   - Distribución centralizada a múltiples organizaciones
   - Actualizaciones controladas del framework
8. **Lógica Avanzada de Integraciones**
   - Configuración independiente por integración
   - Sistema de reintentos configurable (Fijo, Incremental, Fibonacci)
   - Manejo de códigos de respuesta específicos (400, 404, 500, etc.)
   - Respeto a límites de gobierno de Salesforce (< 10000ms total)

Productos Soportados

- **Seguros de Motos** - Implementación completa
- **Seguros de Autos** - Implementación completa
- **Seguros de Viajes** - Base arquitectónica preparada
- **Arrendamiento** - Base arquitectónica preparada

Procesos de Venta Cubiertos

1. **Conocimiento/Prospección**
   - Gestión de Leads
   - Captura y validación de datos iniciales
   - Integraciones con fuentes externas (RUNT, FASECOLDA)
2. **Tarificación**
   - Cálculos de pricing por producto
   - Configuración flexible de planes y coberturas
   - Validaciones de riesgos consultables
3. **Cotización**
   - Generación de prepolizas
   - Gestión de numeración automática

- Cálculos de ajustes y descuentos

4. **Emisión**
   - Creación de pólizas definitivas
   - Integración con sistemas de emisión
   - Notificaciones automáticas
5. **Legalización**
   - Gestión de pagos y transacciones
   - Integración con pasarelas de pago
   - Finalización del proceso de venta

Fuera del Alcance

Limitaciones Actuales

1. **Procesos Internos Específicos de Productos**
   - El framework no desarrolla la lógica de negocio específica de cada producto
   - Solo proporciona la base arquitectónica para integrarlos
2. **Modificación de Procesos Neurálgicos**
   - Los procesos core de PreVenta no pueden ser duplicados o modificados
   - Implementación obligatoria del modelo estándar de Vlocity
3. **Productos No Incluidos**
   - Seguros de Vida
   - Seguros de Hogar
   - Seguros Empresariales
   - Otros productos fuera de la familia de Movilidad
4. **Mejoras en Lógica Existente**
   - No se incluyen optimizaciones de procesos legacy
   - El enfoque es arquitectónico, no funcional
5. **Interfaces de Usuario**
   - No incluye componentes de frontend específicos
   - Se enfoca en la capa de servicios y lógica de negocio

Dependencias Externas

1. **Salesforce Platform**
   - Requiere Salesforce Enterprise o superior
   - Dependiente de límites de gobierno de la plataforma
2. **Vlocity/OmniStudio**
   - Integración obligatoria con Vlocity framework

- Dependiente de versiones compatibles de OmniStudio

3. **Sistemas Externos**
   - RUNT (Registro Único Nacional de Tránsito)
   - FASECOLDA (Federación de Aseguradores Colombianos)
   - Sistemas de pagos y SARLAFT

Roadmap de Implementación

Fase 1: Foundation Core (Completada)

- ✅ Implementación de patrones base
- ✅ Managed Package inicial
- ✅ Procesos para Motos y Autos

Fase 2: Expansión de Productos

- 🔄 Implementación para Seguros de Viajes
- 🔄 Implementación para Arrendamiento
- 🔄 Optimización de performance

Fase 3: Funcionalidades Avanzadas

- ⏳ Dashboard de monitoreo en tiempo real
- ⏳ Analytics de performance por proceso
- ⏳ Integración con herramientas de CI/CD

Fase 4: Escalabilidad

- ⏳ Soporte para nuevas familias de productos
- ⏳ Integración con sistemas de terceros
- ⏳ Arquitectura multi-tenant avanzada

Retos a Afrontar

Técnicos

1. **Mantenimiento del Core**
   - Evolución continua del framework base
   - Compatibilidad con nuevas versiones de Salesforce/Vlocity
2. **Migración de Productos Existentes**
   - Implementación del nuevo enfoque en productos actuales
   - Minimización del impacto en procesos en producción
3. **Gestión de Versiones**

- Mantenimiento de múltiples versiones del package
- Estrategia de deprecación de versiones antiguas

Organizacionales

1. **Cambio de Mentalidad**
   - Adopción del nuevo modelo de desarrollo
   - Capacitación en patrones de diseño y mejores prácticas
2. **Coordinación entre Equipos**
   - Sincronización entre equipos de diferentes productos
   - Establecimiento de estándares de desarrollo
3. **Puntos de Integración**
   - Mantenimiento de interfaces con sistemas externos
   - Gestión de dependencias entre componentes

Beneficios Esperados

Técnicos

- **Separación lógica clara** entre procesos de diferentes productos
- **Facilidad para incluir nuevos productos** y procesos
- **Versionado y obsolescencia controlada** de componentes
- **Mejoras en el manejo de eventos** y notificaciones
- **Independencia lógica** entre productos

Operacionales

- **Distribución controlada** mediante managed packages
- **Disponibilidad desde Salesforce Core y Omnistudio**
- **Capacidad de medir desempeño** real por proceso
- **Múltiples lógicas simultáneas** de tarificación, cotización y emisión

Estratégicos

- **Base escalable** para futuros productos
- **Arquitectura enterprise** robusta y mantenible
- **Reducción de riesgos** por acoplamiento de código
- **Velocidad de implementación** de nuevos productos

Consideraciones de Implementación

Prerrequisitos

- Salesforce Enterprise Edition o superior

- Vlocity Industries CPQ instalado
- Permisos de instalación de managed packages
- Configuración de Named Credentials para integraciones

Recursos Requeridos

- Desarrolladores con experiencia en Salesforce y Vlocity
- Arquitectos familiarizados con patrones de diseño
- Administradores para configuración de procesos
- Equipo de QA para validación de integraciones

Métricas de Éxito

- Reducción en tiempo de implementación de nuevos productos
- Mejora en la estabilidad de procesos existentes
- Disminución de incidentes por acoplamiento de código
- Aumento en la velocidad de resolución de issues

Fecha de Última Actualización: Mayo 2025**Versión del Documento:** 1.0**Responsables:** Soulberto Lorenzo, Jean Carlos Melendez

Arquitectura General - SURA Foundation Framework

Resumen Ejecutivo

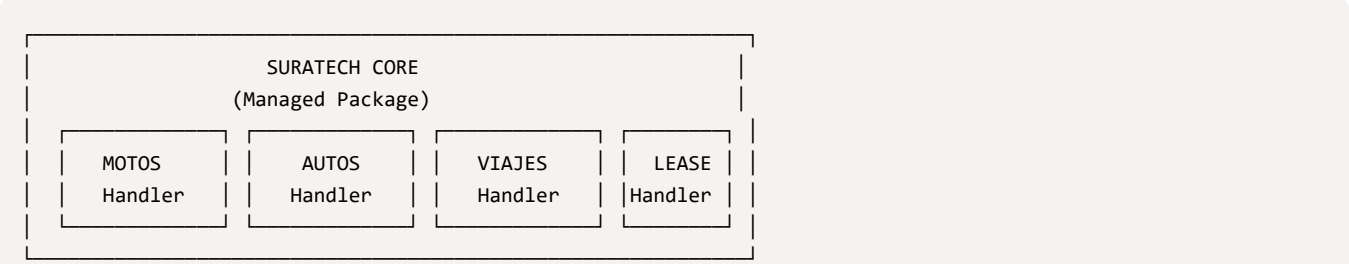
SURA Foundation es un framework de procesamiento centralizado diseñado como un **Managed Package** de Salesforce que proporciona una arquitectura modular y escalable para los procesos de venta de seguros. El framework implementa patrones de diseño enterprise para garantizar la separación lógica, reutilización de componentes y facilitar el mantenimiento de múltiples productos de seguros.

📐 Visión Arquitectónica

Concepto Central: Suratech Core

El corazón del framework es **Suratech Core**, un managed package que encapsula:

- Lógica de negocio común a todos los productos
- Patrones de procesamiento estandarizados
- Interfaces de integración con sistemas externos
- Componentes reutilizables para el ciclo de vida de ventas

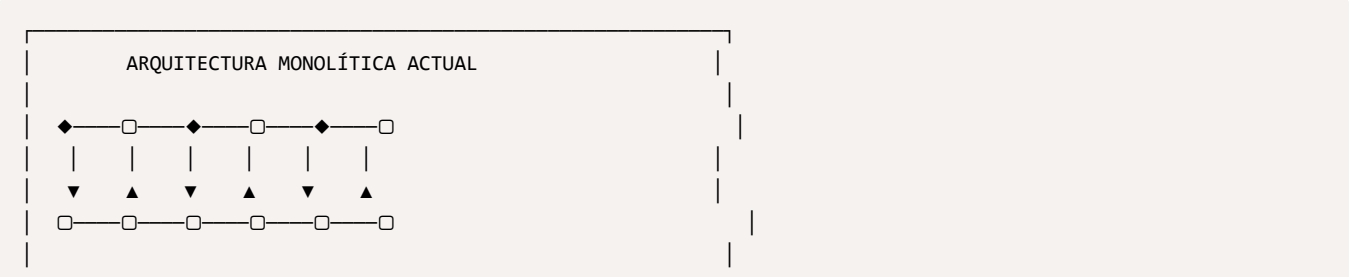


🔄 Evolución Arquitectónica

Arquitectura Actual (Problemática)

Problemas Identificados:

- Código híbrido compartido entre múltiples productos
- Lógica mezclada sin separación clara
- Modificaciones por diferentes células sin control
- Imposibilidad de medir desempeño individual
- Difícil mantenimiento y extensibilidad
- Responsabilidades no definidas por producto

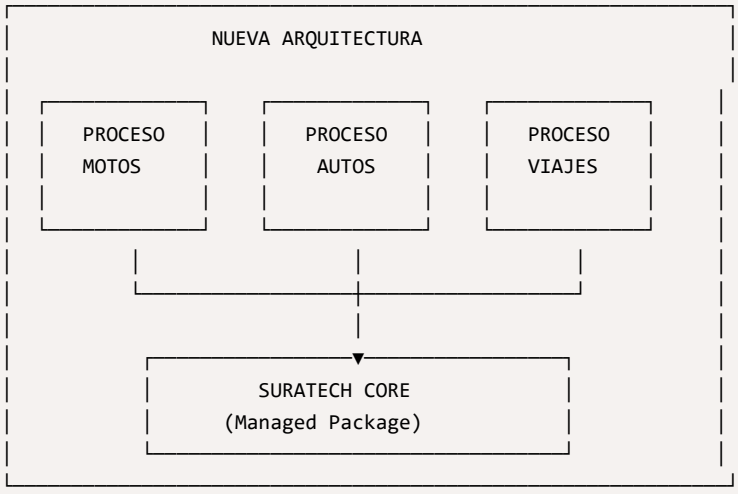


```
│  ⚠️ Código compartido y mezclado                          │
│  ⚠️ Sin separación de responsabilidades                   │
└──────────────────────────────────────────────────────────┘
```

Nueva Arquitectura (Solución)

Beneficios del Nuevo Enfoque:

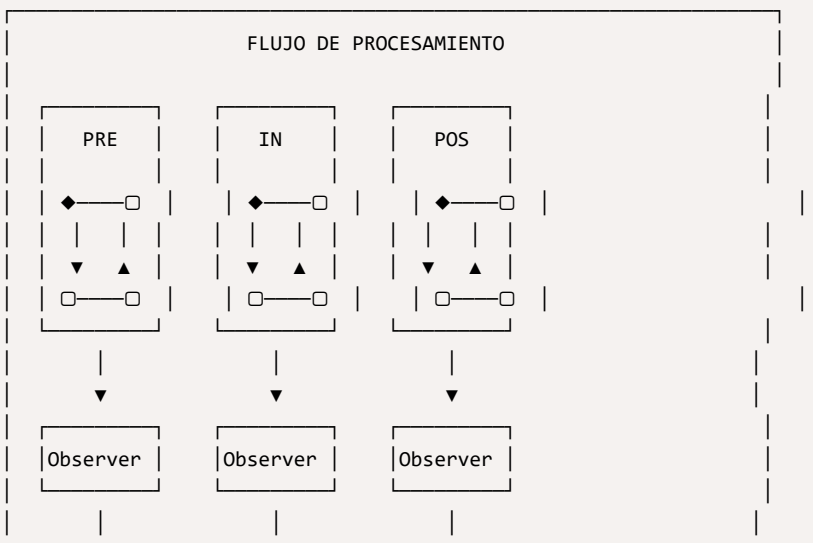
- Separación lógica clara por proceso y producto
- Componentes encapsulados y reutilizables
- Facilidad para agregar nuevos productos
- Medición individual de desempeño
- Versionado y distribución controlada

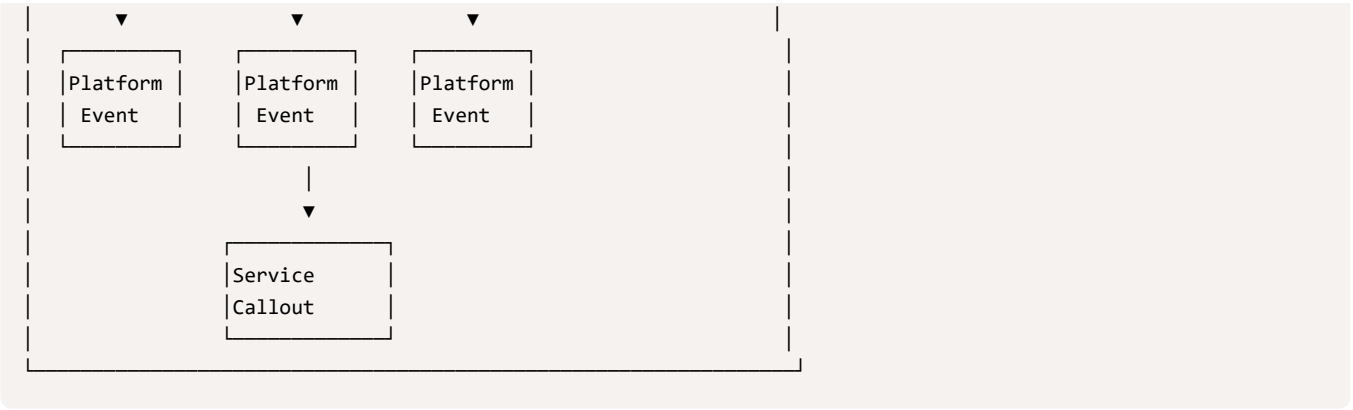


🔧 Estructura de Procesamiento

Modelo Pre-In-Pos

El framework implementa un modelo de procesamiento en tres fases para cada etapa del flujo de ventas:





Fases de Procesamiento:

1. **PRE**: Validaciones y preparación de datos
2. **IN**: Lógica de negocio principal
3. **POS**: Notificaciones y procesos posteriores

Componentes de Integración:

- **Observer**: Monitoreo de eventos en cada fase
- **Platform Events**: Comunicación asíncrona entre componentes
- **Service Callout**: Integración con servicios externos

📦 Arquitectura de Componentes

Artefactos Principales por Proceso

1. Conocimiento (Lead Management)

```
┌──────────────────────────┐
│     CONOCIMIENTO         │
├──────────────────────────┤
│ • LeadManager            │
│ • SURA_ConsultarVehiculo │
│ • SURA_ConsultaFasecolda │
│ • SURA_RiesgosConsultables│
└──────────────────────────┘
```

2. Tarificación (Pricing)

```
┌──────────────────────────┐
│     TARIFICACIÓN         │
├──────────────────────────┤
│ • RateProductHandler     │
│ • PricingEngine          │
│ • ValidationComponents   │
└──────────────────────────┘
```

3. Cotización (Quoting)

```
┌──────────────────────────┐
│      COTIZACIÓN          │
├──────────────────────────┤
```

```
| • SURACrearNumeroPrepoliza|
| • SURAQuotePricingAdjustmentTrigger|
| • SURAPaymentManager     |
| • QuoteManager           |
| • PAGOS_GenerarUrlPago   |
| • CCM_EnviarBienvenida   |
| • PAGOS_ValidacionSARLAFT|
└──────────────────────────┘
```

4. Emisión (Issuance)

```
┌──────────────────────────┐
│       EMISIÓN            │
├──────────────────────────┤
| • SURANotificatorToExternalServices|
| • SURAHomologacionPreEmision|
| • InsurancePolicyManager|
| • SURAAccountManager     |
| • SURAOpportunityManager|
└──────────────────────────┘
```

5. Legalización (Legalization)

```
┌──────────────────────────┐
│     LEGALIZACIÓN         │
├──────────────────────────┤
| • SURAInvoiceListeningServiceV1|
| • InsurancePolicyTransactionServicesV1|
| • InsurancePolicyTransactionManager|
└──────────────────────────┘
```

🎯 Patrones de Diseño Implementados

1. Chain of Responsibility (Cadena de Responsabilidad)

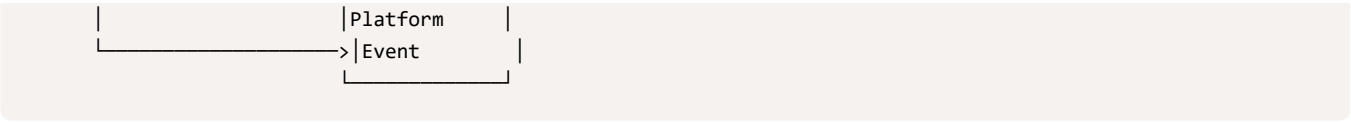
```
Request ──┬──> [Motos Handler] ──┬──> [Autos Handler] ──┬──> [Viajes Handler]
          │                      │                      │
          ▼                      ▼                      ▼
       🙁 Reject              🙁 Reject              ✅ Process
```

Aplicación:

- Procesamiento secuencial por tipo de producto
- Cada handler determina si puede procesar la solicitud
- Si no puede, la pasa al siguiente en la cadena

2. Observer Pattern (Observador)

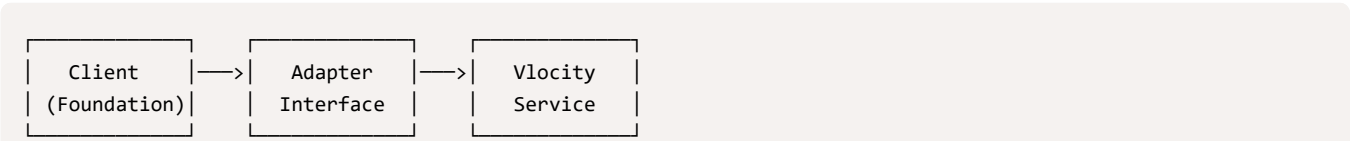
```
┌──────────────┐    notify    ┌──────────────┐
│   Subject    │ ────────────>│  Observer    │
│ (Process)    │              │ (Listener)   │
└──────────────┘              └──────────────┘
      │                             │
      │                             ▼
      │                    ┌──────────────┐
```



Aplicación:

- Monitoreo de cambios en procesos de negocio
- Desacoplamiento entre emisores y receptores de eventos
- Notificaciones automáticas de estados

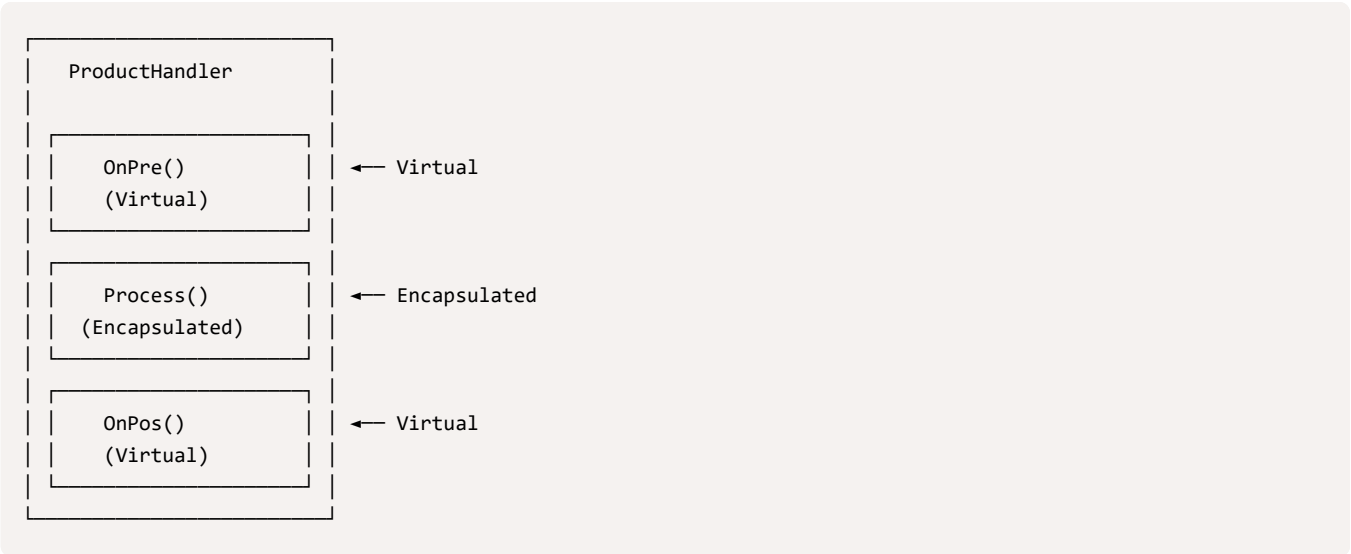
3. Adapter Pattern (Adaptador)



Aplicación:

- Integración transparente con Vlocity/OmniStudio
- Abstracción de servicios externos
- Reutilización de componentes existentes

4. Template Method (Método Plantilla)

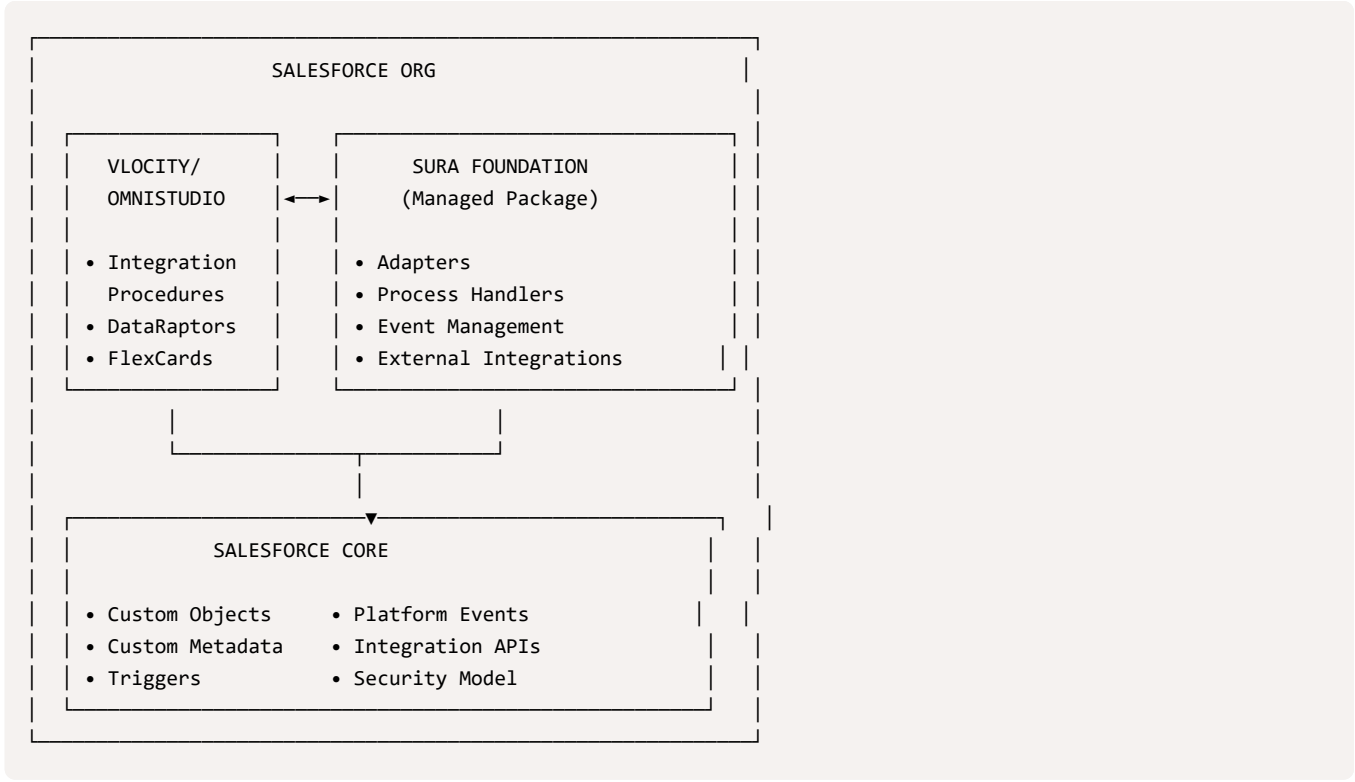


Aplicación:

- Estructura común para todos los handlers de producto
- Puntos de extensión predefinidos (OnPre, OnPos)
- Lógica core protegida y reutilizable

🔗 Integración con Ecosistema Salesforce

Vlocity/OmniStudio Integration

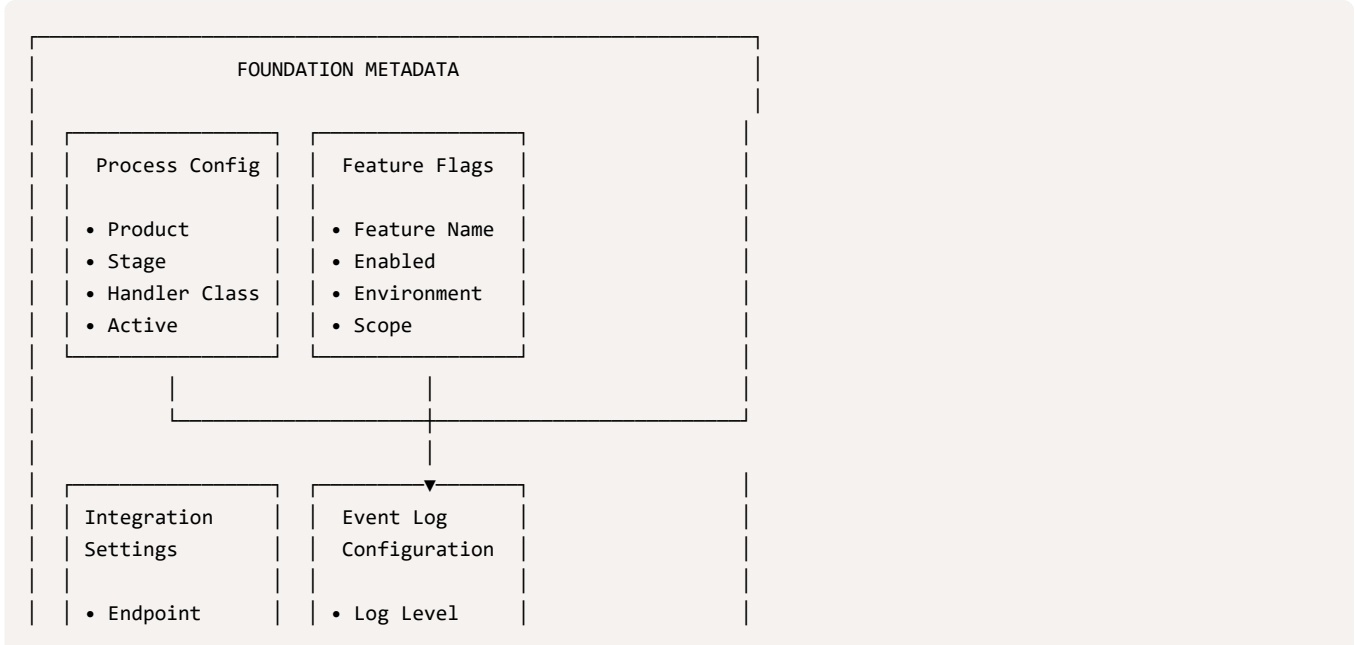


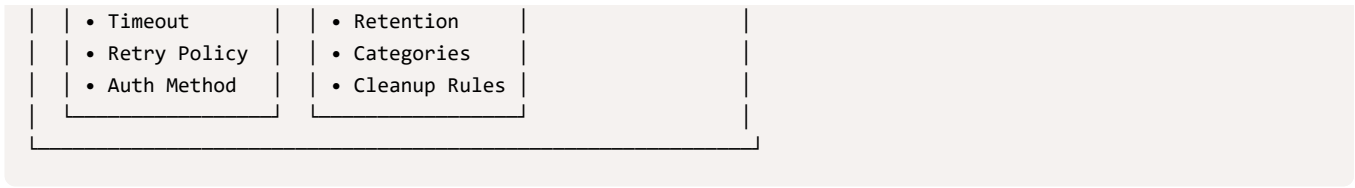
Puntos de Integración:

1. **Integration Procedures**: Llamadas desde Vlocity hacia Foundation
2. **Platform Events**: Comunicación asíncrona bidireccional
3. **Custom Metadata Types**: Configuración compartida
4. **Apex Classes**: Lógica de negocio expuesta via interfaces

📊 Arquitectura de Datos

Modelo de Metadatos

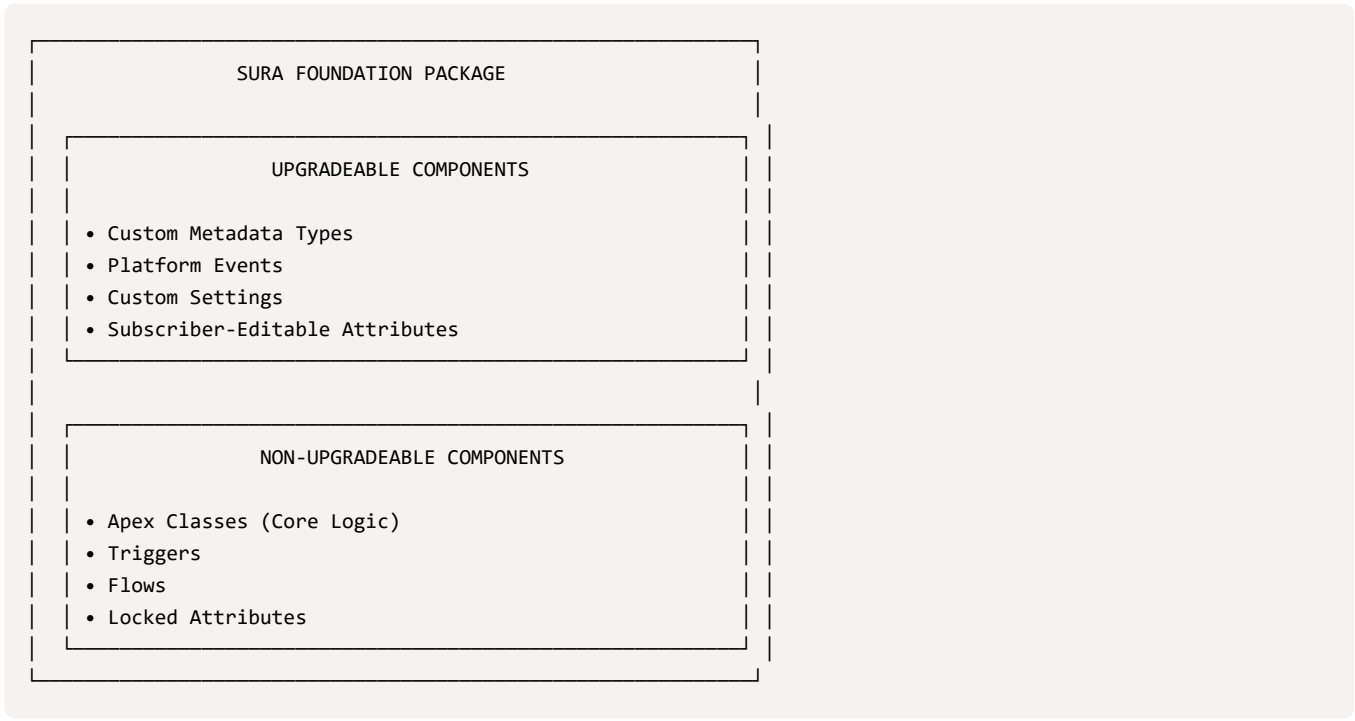




🚀 Empaquetamiento y Distribución

Managed Package Strategy

Estructura del Paquete:



Ventajas del Managed Package:

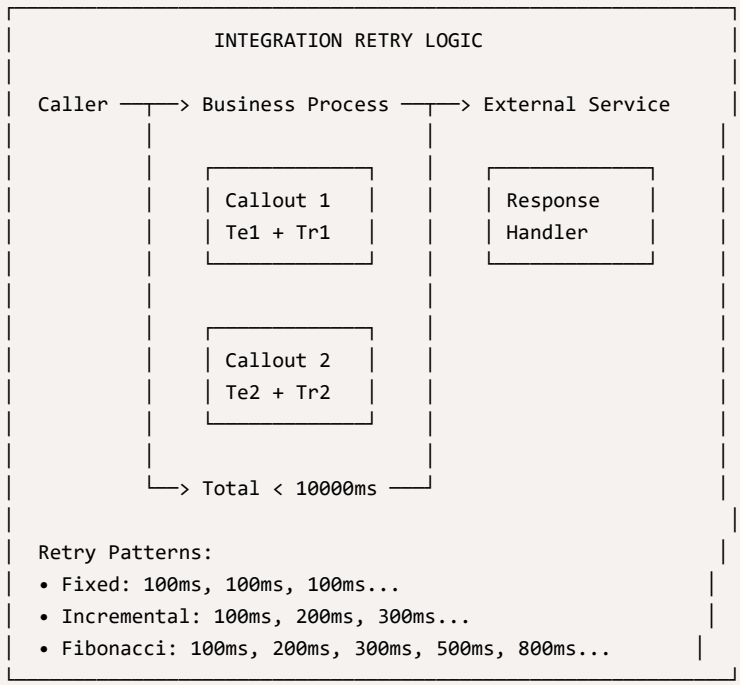
1. **Código Protegido**: Lógica core no visible ni modificable
2. **Versionado Controlado**: Gestión de versiones y actualizaciones
3. **Distribución Centralizada**: Instalación en múltiples orgs
4. **Compatibilidad**: Asegura funcionamiento correcto
5. **Soporte**: Mantenimiento centralizado del framework

Estrategia de Versionado:

- **Versiones Inestables**: 0.1.0-NEXT, 0.1.0-2, etc.
- **Versiones Estables**: 1.0-1, 1.0-3, etc.
- **Actualizaciones**: Automáticas para componentes upgradeables

🔧 Arquitectura de Integración Externa

Sistema de Reintentos

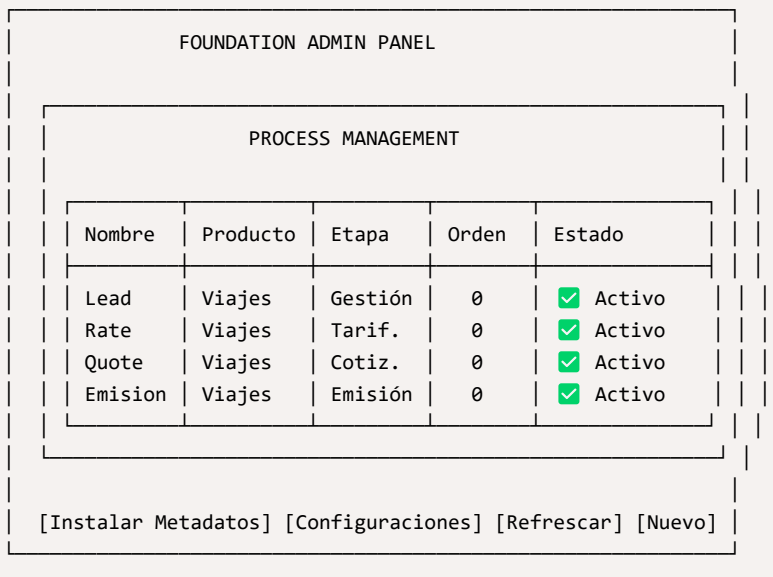


Configuración de Integraciones:

- **Por Integración**: Configuración independiente
- **Códigos de Respuesta**: Definir qué códigos activan reintentos (400, 404, 500, etc.)
- **Cantidad de Reintentos**: Mínimo 1, máximo 5
- **Intervalos de Espera**: Configurables por patrón
- **Límites de Gobierno**: Respeto a los 10 segundos de Salesforce

📋 Panel de Administración

Características del Panel:



Funcionalidades:

- Activar/Desactivar procesos dinámicamente
- Configurar orden de ejecución
- Monitorear estado de integraciones
- Gestionar feature flags por ambiente

🎯 Beneficios Arquitectónicos

Técnicos

- **Modularidad**: Componentes independientes y reutilizables
- **Escalabilidad**: Fácil adición de nuevos productos y procesos
- **Mantenibilidad**: Código organizado con responsabilidades claras
- **Testabilidad**: Componentes aislados facilitan pruebas unitarias
- **Performance**: Medición individual de cada proceso

Operacionales

- **Disponibilidad**: Framework disponible desde Salesforce Core y OmniStudio
- **Configurabilidad**: Feature flags y metadata configurable
- **Monitoreo**: Eventos y logs centralizados
- **Distribución**: Managed package para múltiples organizaciones

Estratégicos

- **Tiempo al Mercado**: Reducción en tiempo de implementación
- **Consistencia**: Estándares unificados para todos los productos
- **Evolución**: Base sólida para futuras expansiones
- **Riesgo**: Reducción de errores por código acoplado

🔮 Roadmap Arquitectónico

Versión Actual (1.0)

- ✅ Framework core implementado
- ✅ Patrones de diseño establecidos
- ✅ Integración con Vlocity
- ✅ Productos Motos y Autos

Versión 2.0 (Próxima)

- 🔄 Expansión a Viajes y Arrendamiento
- 🔄 Dashboard de monitoreo avanzado

- 🔄 APIs REST para integraciones externas
- 🔄 Optimizaciones de performance

Versión 3.0 (Futuro)

- ⏳ Soporte multi-tenant avanzado
- ⏳ Machine Learning para optimización
- ⏳ Integración con herramientas DevOps
- ⏳ Expansión a otras líneas de negocio

Fecha de Actualización: Mayo 2025**Versión del Documento:** 1.0**Arquitectos:** Soulberto Lorenzo, Jean Carlos Melendez**Estado:** Documento Vivo - Actualización Continua

Patrones de Diseño - Framework Foundation

Descripción General

Este documento describe los patrones de diseño implementados en el Framework Foundation, analizando su propósito, implementación y beneficios dentro de la arquitectura del sistema. El framework utiliza múltiples patrones que trabajan en conjunto para crear una solución robusta, extensible y mantenible.

Patrones Implementados

1. Observer Pattern

Implementación

Clase: `BaseObserver` **Interfaz:** `IObserver`

Propósito

Permite que múltiples objetos observen y reaccionen a eventos del sistema sin acoplamiento directo entre el emisor y los receptores.

Cómo se Utiliza

```
public virtual class BaseObserver implements IObserver {
    public virtual void updateObserver(String flag, Object data) {
        Core.debug('flag="' + flag);
    }

    public virtual void publishPlatFormEvent(String flag, String productName) {
        Core.debug('Publishing platform event for product= ' + productName);
    }
}
```

Beneficios en el Framework

- **Desacoplamiento:** Los componentes pueden comunicarse sin conocerse directamente
- **Extensibilidad:** Nuevos observadores pueden agregarse sin modificar el código existente
- **Event-Driven Architecture:** Facilita una arquitectura basada en eventos
- **Reactividad:** El sistema puede responder automáticamente a cambios de estado

Casos de Uso

- Notificaciones de cambios de estado en objetos de negocio
- Logging automático de eventos del sistema
- Integración con Platform Events de Salesforce
- Auditoría de operaciones críticas

2. Provider Pattern

Implementación

Clase: `FeatureFlagProvider` **Interfaz:** `IFeatureFlagProvider`

Propósito

Abstrae la obtención de datos y configuraciones, permitiendo diferentes fuentes de información sin cambiar el código cliente.

Cómo se Utiliza

```
public with sharing class FeatureFlagProvider implements IFeatureFlagProvider {
    public Set<String> getCustomPermissionNames(){
        // Obtiene permisos personalizados desde Salesforce
        List<CustomPermission> perms = [SELECT Id, DeveloperName FROM CustomPermission];
        // ... procesamiento
        return customPermissionNames;
    }

    public Map<String,FeatureFlag__mdt> getFeatureFlags(){
        return FeatureFlag__mdt.getAll();
    }
}
```

Beneficios en el Framework

- **Abstracción de Fuentes:** Oculta la complejidad de obtener datos de diferentes fuentes
- **Testabilidad:** Permite inyección de dependencias para testing
- **Flexibilidad:** Diferentes proveedores para diferentes entornos
- **Cacheo:** Centraliza estrategias de cache y performance

Casos de Uso

- Obtención de feature flags desde Custom Metadata Types
- Acceso a permisos personalizados
- Configuraciones dinámicas sin deployments
- Testing con datos mock

3. Strategy Pattern

Implementación

Clases: `FeatureFlags`, `AccountConfigurator` **Interfaces:** `IFeatureFlags`, `AccountConfigurator`

Propósito

Permite intercambiar algoritmos o comportamientos en tiempo de ejecución sin modificar el código cliente.

Cómo se Utiliza

En FeatureFlags

```
public FeatureEvaluationResult evaluate(String featureName) {
    // Estrategia 1: Mock Values (para testing)
    if (Test.isRunningTest() && mockValues.containsKey(featureName)) {
        return new FeatureEvaluationResult(mockValues.get(featureName), featureName, FeatureReason.MOCK_VALUE);
    }

    // Estrategia 2: Custom Permissions
    if (customPermissionNames.contains(featureName)) {
        if (FeatureManagement.checkPermission(featureName)) {
            return new FeatureEvaluationResult(true, featureName, FeatureReason.HAS_CUSTOM_PERMISSION);
        }
    }

    // Estrategia 3: Custom Metadata Types
    if (features.containsKey(featureName)) {
```

```
        if (features.get(featureName).Is_Active__c) {
            return new FeatureEvaluationResult(true, featureName, FeatureReason.CUSTOM_METADATA_TYPE_ENABLED);
        }
    }
}
```

En AccountConfigurator

```
global interface AccountConfigurator {
    void configure(Account account);
}

// Diferentes estrategias de configuración
global class IndustryAccountConfigurator implements AccountConfigurator {
    global void configure(Account account) {
        account.Industry = this.industry;
        // Lógica específica por industria
    }
}
```

Beneficios en el Framework

- **Intercambiabilidad:** Diferentes algoritmos para el mismo propósito
- **Extensibilidad:** Nuevas estrategias sin modificar código existente
- **Mantenibilidad:** Separación clara de responsabilidades
- **Configurabilidad:** Comportamiento dinámico basado en configuración

Casos de Uso

- Evaluación de feature flags con múltiples fuentes
- Configuración modular de objetos Account
- Algoritmos de procesamiento intercambiables
- Validaciones condicionales

4. Adapter Pattern

Implementación

Clase: `BaseProcessFoundationVlocityAdapter`

Propósito

Permite que interfaces incompatibles trabajen juntas, actuando como puente entre diferentes sistemas o APIs.

Cómo se Utiliza

```
global abstract class BaseProcessFoundationVlocityAdapter {
    global static Boolean applyInvoke(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap,
        Map<String, Object> optionMap
    ) {
        String stage = (String) inputMap?.get('Stage');
        // Adapta la interfaz de Vlocity a BaseProcessManager
        return BaseProcessManager.invoke(stage, optionMap, inputMap, outputMap);
    }
```

```
    public Boolean invokeMethod(String methodName, Map<String, Object> inputMap,
                                Map<String, Object> outputMap, Map<String, Object> optionMap) {
        // Invocación dinámica por nombre de método
        switch on methodName.toUpperCase() {
            when 'APPLYINVOKE' {
                return applyInvoke(inputMap, outputMap, optionMap);
            }
        }
    }
}
```

Beneficios en el Framework

- **Integración:** Conecta Vlocity framework con lógica de negocio personalizada
- **Abstracción:** Oculta complejidad del BaseProcessManager
- **Estandarización:** Interfaz consistente para diferentes invocaciones
- **Desacoplamiento:** Vlocity no depende directamente de clases internas

Casos de Uso

- Integración entre Vlocity y BaseProcessManager
- Adaptación de APIs externas
- Puente entre diferentes versiones de interfaces
- Normalización de parámetros entre sistemas

5. Chain of Responsibility Pattern

Implementación

Clase: `BaseProcessManager` **Componentes:** `ProductHandler`, `ProcessFactory`, `ReverseIterator`

Propósito

Permite pasar solicitudes a lo largo de una cadena de manejadores hasta que uno de ellos procese la solicitud.

Cómo se Utiliza

```
global static Boolean invoke(String stage, Map<String, Object> options,
                             Map<String, Object> input, Map<String, Object> output) {
    // 1. Obtener handlers para el stage
    ReverseIterator reverseIterator = new ReverseIterator(
        ((List<ProductHandler>) ProcessFactory.newInstances(
            ProcessFactory.getAllDefinitionsByStage(stage, ignoreCache)
        ))
    );

    // 2. Construir la cadena de responsabilidad
    if (!reverseIterator.isEmpty()) {
        ProductHandler previous = (ProductHandler) reverseIterator.next();

        while (reverseIterator.hasNext()) {
            ProductHandler current = (ProductHandler) reverseIterator.next();
            current.setNext(previous);  // Enlazar handlers
            previous = current;
        }
```

```
        // 3. Ejecutar la cadena
        ((ProductHandler) reverseIterator.first()).apply(options, input, output);
    }

    return true;
}
```

Beneficios en el Framework

- **Flexibilidad:** Configuración dinámica de la cadena de procesamiento
- **Desacoplamiento:** Handlers no conocen la estructura completa de la cadena
- **Extensibilidad:** Nuevos handlers sin modificar código existente
- **Configurabilidad:** Orden y composición basada en metadata

Casos de Uso

- Procesamiento de órdenes con múltiples validaciones
- Pipeline de transformación de datos
- Workflows configurables por stage
- Procesamiento secuencial con puntos de control

6. Builder Pattern

Implementación

Clase: `AccountBuilder` **Interfaz:** `AccountConfigurator`

Propósito

Construye objetos complejos paso a paso, permitiendo diferentes representaciones del mismo tipo de objeto.

Cómo se Utiliza

```
global class AccountBuilder {
    global Account account;
    global List<AccountConfigurator> configurators = new List<AccountConfigurator>();

    // Métodos fluidos para construcción
    global AccountBuilder setName(String name) {
        account.Name = name;
        return this;
    }

    global AccountBuilder setPhone(String phone) {
        account.Phone = phone;
        return this;
    }

    global AccountBuilder addConfigurators(List<AccountConfigurator> configuratorsList) {
        this.configurators.addAll(configuratorsList);
        return this;
    }

    // Construcción final
    global Account build() {
        for (AccountConfigurator configurator : configurators) {
            configurator.configure(account);
```

```
        }
        return account;
    }
}
```

Beneficios en el Framework

- **Fluent Interface:** API intuitiva y legible
- **Flexibilidad:** Construcción opcional e incremental
- **Extensibilidad:** Configuradores personalizados
- **Reutilización:** Mismo builder para diferentes variaciones

Casos de Uso

- Construcción de objetos Account complejos
- Configuración modular con Strategy pattern
- APIs de creación user-friendly
- Testing con builders de datos mock

7. Factory Pattern

Implementación

Clase: `ProcessFactory` (referenciada en BaseProcessManager)

Propósito

Crea objetos sin especificar la clase exacta del objeto que será creado.

Cómo se Utiliza

```
// Uso implícito en BaseProcessManager
List<ProductHandler> handlers = ProcessFactory.newInstances(
    ProcessFactory.getAllDefinitionsByStage(stage, ignoreCache)
);
```

Beneficios en el Framework

- **Abstracción:** Oculta lógica de instanciación
- **Flexibilidad:** Diferentes tipos de handlers basados en configuración
- **Centralización:** Lógica de creación en un solo lugar
- **Cache:** Optimización de instanciación

Casos de Uso

- Creación dinámica de ProductHandlers
- Instanciación basada en metadata
- Cache de instancias para performance
- Inyección de dependencias

Interacción Entre Patrones

Feature Flags Ecosystem

```
Provider Pattern → Strategy Pattern → Observer Pattern
(FeatureFlagProvider) → (FeatureFlags) → (BaseObserver)
```

Process Management Chain

```
Adapter Pattern → Chain of Responsibility → Factory Pattern
(VlocityAdapter) → (BaseProcessManager) → (ProcessFactory)
```

Account Building System

```
Builder Pattern ← Strategy Pattern
(AccountBuilder) ← (AccountConfigurator)
```

Beneficios Arquitectónicos

Mantenibilidad

- **Separación de responsabilidades:** Cada patrón maneja una preocupación específica
- **Código modular:** Cambios aislados sin efectos secundarios
- **Testing independiente:** Cada componente puede probarse por separado

Extensibilidad

- **Open/Closed Principle:** Abierto para extensión, cerrado para modificación
- **Configuración sin código:** Metadata-driven configuration
- **Plugin architecture:** Nuevas funcionalidades como plugins

Performance

- **Cache strategies:** Implementadas en Provider y Factory patterns
- **Lazy loading:** Instanciación bajo demanda
- **Bulk operations:** Optimización para operaciones masivas

Reusabilidad

- **Componentes intercambiables:** Interfaces bien definidas
- **Configuraciones reutilizables:** Estrategias aplicables en múltiples contextos
- **Abstracción de complejidad:** APIs simples para funcionalidad compleja

Consideraciones de Implementación

Performance

- Cache habilitado por feature flags
- Factory pattern para optimización de instanciación
- Configuradores eficientes para operaciones bulk

Security

- Feature flags basados en permisos de usuario
- Validación en configuradores personalizados
- Respeto a field-level security

Testing

- Mock strategies en Strategy pattern
- Builder pattern para datos de test
- Provider pattern para inyección de dependencias

Evolución del Framework

Patrones Futuros Recomendados

- **Command Pattern:** Para operaciones undoable
- **Memento Pattern:** Para state management
- **Visitor Pattern:** Para operaciones sobre estructuras complejas
- **Template Method:** Para algoritmos con pasos variables

Mejoras Arquitectónicas

- Event sourcing con Observer pattern
- CQRS con Strategy pattern
- Microservices communication con Adapter pattern
- Domain-driven design con Builder pattern

Conclusión

El Framework Foundation implementa una arquitectura sólida basada en patrones de diseño probados. La combinación de estos patrones crea un sistema que es:

- **Flexible:** Adaptable a diferentes requerimientos
- **Extensible:** Fácil de expandir con nueva funcionalidad
- **Mantenible:** Código limpio y bien estructurado
- **Testeable:** Componentes independientes y mockeables
- **Performante:** Optimizaciones integradas en el diseño

Esta arquitectura proporciona una base sólida para el desarrollo de aplicaciones empresariales complejas en Salesforce, manteniendo la simplicidad de uso mientras se ofrece la flexibilidad necesaria para casos de uso avanzados.

Modelo de Datos - SURA Foundation Framework

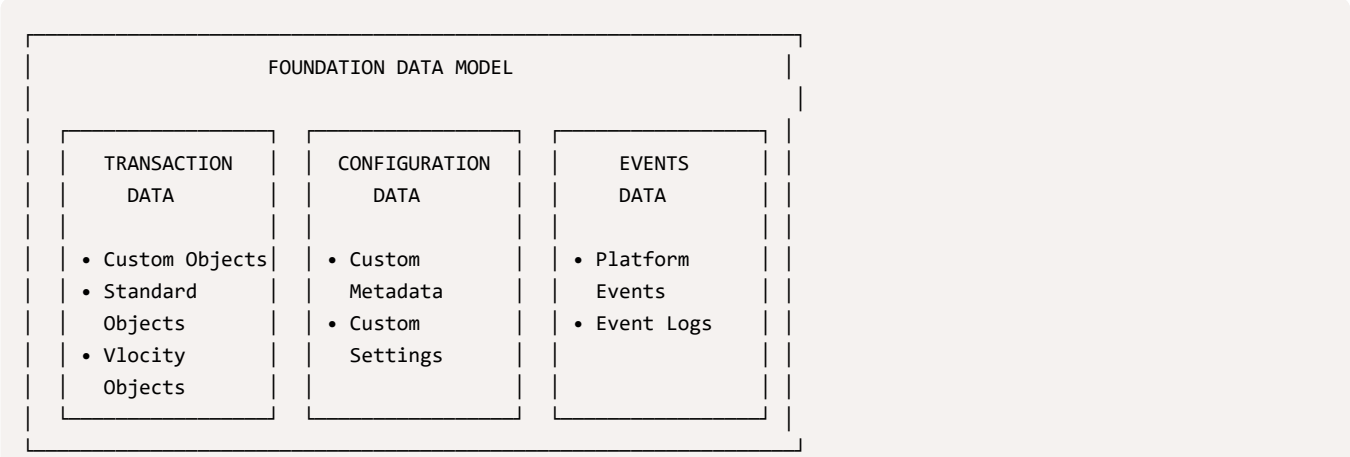
Resumen Ejecutivo

El modelo de datos de **SURA Foundation Framework** está diseñado para soportar la arquitectura modular y escalable del sistema. Se basa en una combinación de **Custom Objects** para datos transaccionales, **Custom Metadata Types** para configuración, y **Platform Events** para comunicación asíncrona entre componentes.

🏗️ Arquitectura de Datos

Principios de Diseño

1. **Separación de Responsabilidades**: Datos transaccionales vs configuración
2. **Flexibilidad**: Configuración sin deployments mediante metadata
3. **Trazabilidad**: Log completo de eventos y transacciones
4. **Escalabilidad**: Modelo preparado para múltiples productos
5. **Integración**: Compatibilidad con objetos estándar de Salesforce/Vlocity



📊 Custom Objects Principales

1. SFFoundation_EventLog__c

Propósito: Registro centralizado de eventos del framework

Campo	Tipo	Descripción
Name	Auto Number	Identificador único del evento (EL-{0000})
EventType__c	Picklist	Tipo de evento (Process, Integration, Error, etc.)
ProcessName__c	Text(255)	Nombre del proceso que generó el evento
ProductFamily__c	Picklist	Familia de producto (Motos, Autos, Viajes, etc.)
Stage__c	Picklist	Etapa del proceso (Conocimiento, Tarificación, etc.)
Status__c	Picklist	Estado del evento (Success, Error, Warning, Info)
Message__c	Long Text	Mensaje detallado del evento
StackTrace__c	Long Text	Stack trace en caso de errores
RecordId__c	Text(18)	ID del registro relacionado
UserId__c	Lookup(User)	Usuario que ejecutó la acción
SessionId__c	Text(255)	ID de sesión para trazabilidad
ExecutionTime__c	Number(10,2)	Tiempo de ejecución en milisegundos
Payload__c	Long Text	Datos de entrada/salida (JSON)
CreatedDate	DateTime	Fecha y hora de creación

Índices:

- EventType__c, CreatedDate
- ProcessName__c, Status__c
- ProductFamily__c, Stage__c

2. SFFoundation_ProcessExecution__c

Propósito: Seguimiento de ejecuciones de procesos

Campo	Tipo	Descripción
Name	Auto Number	Número de ejecución (PE-{0000})
ProcessName__c	Text(255)	Nombre del proceso ejecutado
ProductFamily__c	Picklist	Familia de producto
HandlerClass__c	Text(255)	Clase handler ejecutada
Stage__c	Picklist	Etapa del proceso
Phase__c	Picklist	Fase (Pre, In, Pos)
Status__c	Picklist	Estado (Running, Completed, Failed)
StartTime__c	DateTime	Hora de inicio
EndTime__c	DateTime	Hora de finalización
Duration__c	Number(10,2)	Duración en milisegundos
InputParameters__c	Long Text	Parámetros de entrada (JSON)
OutputParameters__c	Long Text	Parámetros de salida (JSON)
ErrorMessage__c	Long Text	Mensaje de error si aplica
ParentExecution__c	Lookup(SFFoundation_ProcessExecution__c)	Ejecución padre
LeadId__c	Lookup(Lead)	Lead relacionado
OpportunityId__c	Lookup(Opportunity)	Oportunidad relacionada
QuoteId__c	Lookup(Quote)	Cotización relacionada
PolicyId__c	Text(255)	ID de póliza externa

3. SFFoundation_IntegrationLog__c

Propósito: Log de integraciones con servicios externos

Campo	Tipo	Descripción
Name	Auto Number	Número de integración (IL-{0000})
ServiceName__c	Text(255)	Nombre del servicio integrado
Endpoint__c	Text(255)	URL del endpoint
HttpMethod__c	Picklist	Método HTTP (GET, POST, PUT, DELETE)
RequestHeaders__c	Long Text	Headers de la request (JSON)
RequestBody__c	Long Text	Cuerpo de la request
ResponseStatus__c	Number(3,0)	Código de estado HTTP
ResponseHeaders__c	Long Text	Headers de la response (JSON)
ResponseBody__c	Long Text	Cuerpo de la response
RequestTime__c	DateTime	Hora de envío
ResponseTime__c	DateTime	Hora de respuesta
Duration__c	Number(10,2)	Duración en milisegundos
RetryAttempt__c	Number(2,0)	Número de intento
MaxRetries__c	Number(2,0)	Máximo número de reintentos
IsSuccess__c	Checkbox	Indica si fue exitosa
ErrorCode__c	Text(50)	Código de error interno
ErrorMessage__c	Text(255)	Mensaje de error
ProcessExecutionId__c	Lookup(SFFoundation_ProcessExecution__c)	Ejecución relacionada

⚙ Custom Metadata Types

1. SFFoundation_ProcessConfiguration__mdt

Propósito: Configuración de procesos por producto y etapa

Campo	Tipo	Descripción
DeveloperName	Text(40)	Nombre único del proceso
MasterLabel	Text(40)	Etiqueta del proceso
ProductFamily__c	Text(255)	Familia de producto
Stage__c	Text(255)	Etapa del proceso
HandlerClass__c	Text(255)	Clase Apex del handler
Order__c	Number(3,0)	Orden de ejecución
IsActive__c	Checkbox	Indica si está activo
Description__c	Text(255)	Descripción del proceso
ApiName__c	Text(255)	Nombre de API para referencias
ConfigurationJSON__c	Long Text	Configuración adicional (JSON)

Ejemplo de registros:

```
Lead_Motos_Conocimiento:
  ProductFamily__c: "Seguros de Motos"
  Stage__c: "Conocimiento"
  HandlerClass__c: "MT_Kn_123"
  Order__c: 0
  IsActive__c: true

Rate_Autos_Tarificacion:
  ProductFamily__c: "Seguros de Autos"
  Stage__c: "Tarificación"
  HandlerClass__c: "AU_Rt_456"
  Order__c: 0
  IsActive__c: true
```

2. SFFoundation_FeatureFlag__mdt

Propósito: Control de funcionalidades por ambiente

Campo	Tipo	Descripción
DeveloperName	Text(40)	Nombre único del feature
MasterLabel	Text(40)	Etiqueta del feature
IsEnabled__c	Checkbox	Indica si está habilitado
Environment__c	Text(50)	Ambiente (Sandbox, Production, etc.)
Scope__c	Picklist	Alcance (Global, Product, Process)
ProductFamily__c	Text(255)	Familia de producto específica
Description__c	Text(255)	Descripción de la funcionalidad
EffectiveDate__c	Date	Fecha de activación
ExpirationDate__c	Date	Fecha de expiración
ConfigurationJSON__c	Long Text	Configuración específica (JSON)

3. SFFoundation_IntegrationConfiguration__mdt

Propósito: Configuración de integraciones externas

Campo	Tipo	Descripción
DeveloperName	Text(40)	Nombre único de la integración
MasterLabel	Text(40)	Etiqueta de la integración
ServiceName__c	Text(255)	Nombre del servicio
BaseURL__c	Text(255)	URL base del servicio
AuthMethod__c	Picklist	Método de autenticación
TimeoutMs__c	Number(5,0)	Timeout en milisegundos
MaxRetries__c	Number(2,0)	Máximo número de reintentos
RetryPattern__c	Picklist	Patrón de reintento (Fixed, Incremental, Fibonacci)
RetryIntervalMs__c	Number(4,0)	Intervalo base de reintento
RetryOnHttpCodes__c	Text(255)	Códigos HTTP que activan reintento
IsActive__c	Checkbox	Indica si está activa
Description__c	Text(255)	Descripción de la integración

4. SFFoundation_EventLogConfiguration__mdt

Propósito: Configuración del sistema de logging

Campo	Tipo	Descripción
DeveloperName	Text(40)	Nombre único de configuración
MasterLabel	Text(40)	Etiqueta de configuración
LogLevel__c	Picklist	Nivel de log (ERROR, WARN, INFO, DEBUG)
Category__c	Text(255)	Categoría de eventos
IsEnabled__c	Checkbox	Indica si está habilitado
RetentionDays__c	Number(3,0)	Días de retención de logs
MaxRecordsPerBatch__c	Number(4,0)	Máximo registros por lote
CleanupSchedule__c	Text(255)	Programación de limpieza (Cron)
NotificationEmail__c	Email	Email para notificaciones críticas

Platform Events

1. SFFoundation_ProcessEvent__e

Propósito: Comunicación de eventos entre procesos

Campo	Tipo	Descripción
EventType__c	Text(255)	Tipo de evento
ProcessName__c	Text(255)	Nombre del proceso
ProductFamily__c	Text(255)	Familia de producto
Stage__c	Text(255)	Etapa del proceso
Phase__c	Text(50)	Fase (Pre, In, Pos)
Status__c	Text(50)	Estado del evento
RecordId__c	Text(18)	ID del registro relacionado
UserId__c	Text(18)	ID del usuario
SessionId__c	Text(255)	ID de sesión
Payload__c	Long Text	Datos del evento (JSON)
Timestamp__c	DateTime	Timestamp del evento

2. SFFoundation_IntegrationEvent__e

Propósito: Eventos de integraciones con servicios externos

Campo	Tipo	Descripción
ServiceName__c	Text(255)	Nombre del servicio
EventType__c	Text(100)	Tipo de evento (Request, Response, Error)
Status__c	Text(50)	Estado de la integración
HttpStatus__c	Number(3,0)	Código de estado HTTP
Duration__c	Number(10,2)	Duración en milisegundos
RecordId__c	Text(18)	ID del registro relacionado
RequestId__c	Text(255)	ID único de la request
ErrorCode__c	Text(50)	Código de error
ErrorMessage__c	Text(255)	Mensaje de error
Payload__c	Long Text	Datos de la integración (JSON)

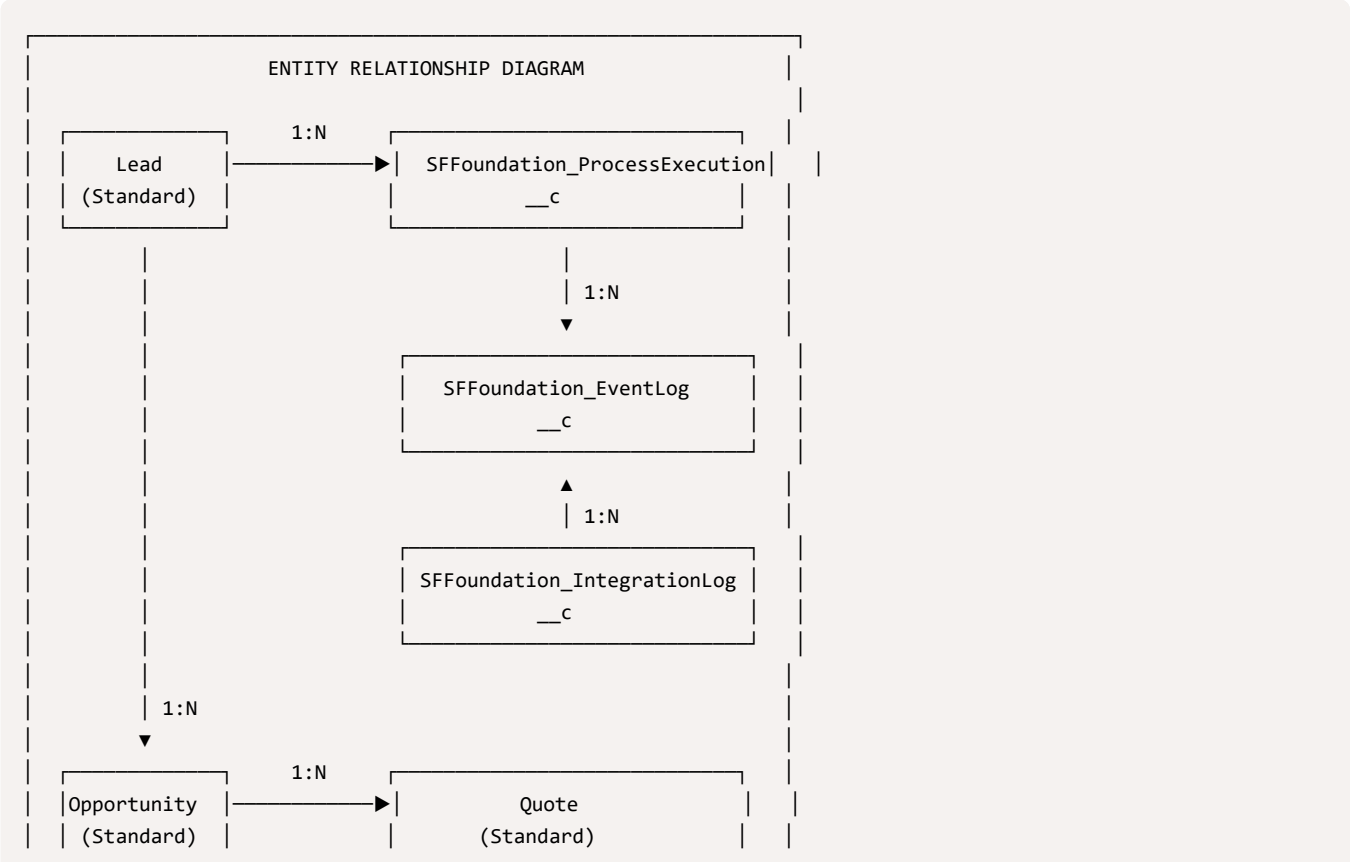
3. SFFoundation_NotificationEvent__e

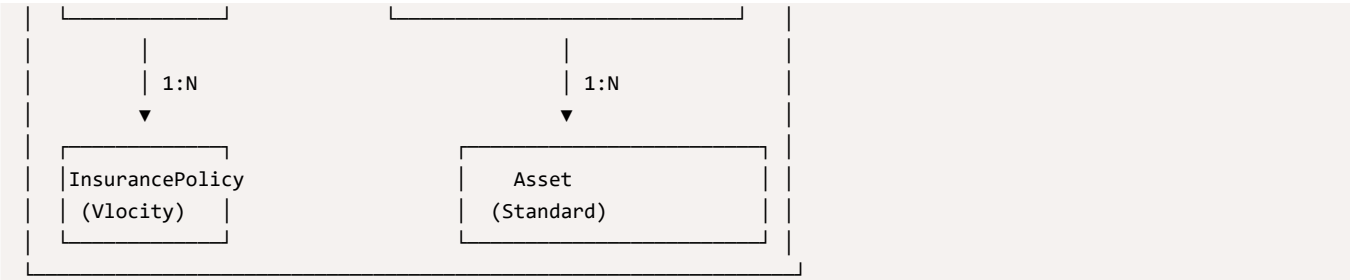
Propósito: Notificaciones del sistema

Campo	Tipo	Descripción
NotificationType__c	Text(100)	Tipo de notificación
Severity__c	Text(50)	Severidad (Critical, High, Medium, Low)
Title__c	Text(255)	Título de la notificación
Message__c	Long Text	Mensaje de la notificación
RecipientId__c	Text(18)	ID del destinatario
RecipientType__c	Text(50)	Tipo de destinatario (User, Queue, Group)
RecordId__c	Text(18)	ID del registro relacionado
Category__c	Text(100)	Categoría de la notificación
ExpirationDate__c	DateTime	Fecha de expiración
IsRead__c	Checkbox	Indica si fue leída

🔗 Relaciones y Dependencias

Diagrama de Entidades Principales





Flujo de Datos por Proceso

1. Conocimiento (Lead Management)

```
Lead Creation ──┬──> ProcessExecution__c
                │
                ├──> EventLog__c (Lead Created)
                │
                ├──> IntegrationLog__c (RUNT/FASECOLDA)
                │
                └──> Platform Event (Lead_Process_Complete)
```

2. Tarificación (Rating)

```
Rate Request ──┬──> ProcessExecution__c
               │
               ├──> EventLog__c (Rating Started)
               │
               ├──> IntegrationLog__c (External Rating Service)
               │
               ├──> EventLog__c (Rating Completed)
               │
               └──> Platform Event (Rating_Complete)
```

3. Cotización (Quoting)

```
Quote Creation ──┬──> ProcessExecution__c
                 │
                 ├──> EventLog__c (Quote Generated)
                 │
                 ├──> IntegrationLog__c (Payment Gateway)
                 │
                 └──> Platform Event (Quote_Ready)
```

4. Emisión (Issuance)

```
Policy Issuance ──┬──> ProcessExecution__c
                  │
                  ├──> EventLog__c (Policy Issued)
                  │
                  ├──> IntegrationLog__c (Core Insurance System)
                  │
                  └──> Platform Event (Policy_Issued)
```

📋 Custom Settings

1. SFFoundation_GlobalSettings__c

Propósito: Configuraciones globales del framework

Campo	Tipo	Descripción
`Name`	Text(38)	Nombre del setting
`IsEnabled__c`	Checkbox	Indica si está habilitado
`LogLevel__c`	Text(10)	Nivel de log global
`DefaultTimeout__c`	Number(5,0)	Timeout por defecto
`MaxRetries__c`	Number(2,0)	Reintentos por defecto
`CleanupBatchSize__c`	Number(4,0)	Tamaño de lote para limpieza
`NotificationEmail__c`	Email	Email para notificaciones

2. SFFoundation_ProductSettings__c

Propósito: Configuraciones específicas por producto

Campo	Tipo	Descripción
`Name`	Text(38)	Nombre del producto
`IsActive__c`	Checkbox	Indica si está activo
`DefaultHandler__c`	Text(255)	Handler por defecto
`ConfigurationJSON__c`	Long Text	Configuración específica (JSON)
`ValidationRules__c`	Long Text	Reglas de validación (JSON)

🔍 Índices y Performance

Estrategia de Indexación

Objetos de Alto Volumen

SFFoundation_EventLog__c:

- Índice compuesto: `(EventType__c, CreatedDate)`
- Índice compuesto: `(ProcessName__c, Status__c)`
- Índice simple: `CreatedDate` (para cleanup)

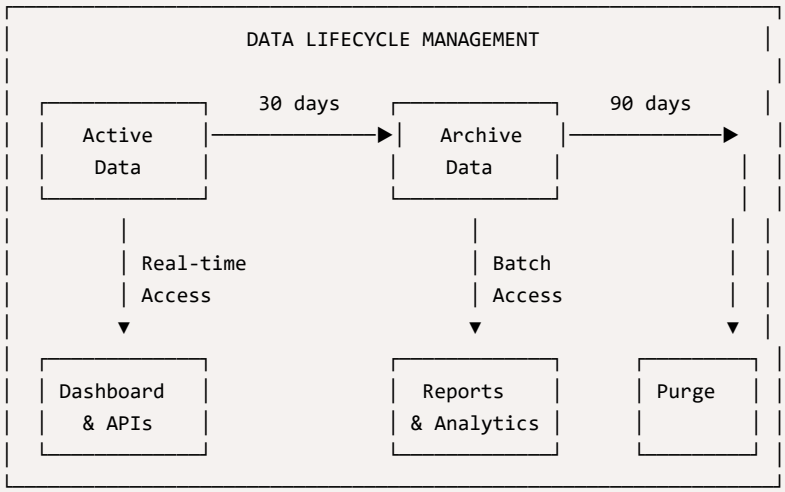
SFFoundation_ProcessExecution__c:

- Índice compuesto: `(ProcessName__c, Status__c, StartTime__c)`
- Índice simple: `LeadId__c`
- Índice simple: `OpportunityId__c`

SFFoundation_IntegrationLog__c:

- Índice compuesto: `(ServiceName__c, RequestTime__c)`
- Índice simple: `IsSuccess__c`
- Índice simple: `ProcessExecutionId__c`

Estrategias de Archivado



📊 Ejemplo de Datos de Configuración

Configuración de Procesos

```
{
  "processes": [
    {
      "developerName": "Lead_Motos_Conocimiento",
      "productFamily": "Seguros de Motos",
      "stage": "Conocimiento",
      "handlerClass": "MT_Kn_123",
      "order": 0,
      "isActive": true,
      "configuration": {
        "validateRUNT": true,
        "requireFasecolda": true,
        "mandatoryFields": ["VehicleType__c", "Model__c", "Year__c"]
      }
    },
    {
      "developerName": "Rate_Motos_Tarificacion",
      "productFamily": "Seguros de Motos",
      "stage": "Tarificación",
      "handlerClass": "MT_Rt_456",
      "order": 0,
      "isActive": true,
      "configuration": {
        "externalRatingService": "SURA_Rating_API",
        "fallbackMethod": "internal",
        "cacheResults": true
```

```
      }
    }
  ]
}
```

Configuración de Feature Flags

```
{
  "featureFlags": [
    {
      "developerName": "EnableNewRatingEngine",
      "isEnabled": false,
      "environment": "Production",
      "scope": "Global",
      "effectiveDate": "2025-06-01",
      "description": "Habilita el nuevo motor de tarificación"
    },
    {
      "developerName": "EnableAdvancedLogging",
      "isEnabled": true,
      "environment": "Sandbox",
      "scope": "Product",
      "productFamily": "Seguros de Motos",
      "description": "Logging avanzado para debugging"
    }
  ]
}
```

🚀 Consideraciones de Implementación

Mejores Prácticas

1. **Naming Conventions**:
   - Custom Objects: `SFFoundation_[ObjectName]__c`
   - Custom Fields: `[FieldName]__c`
   - Metadata Types: `SFFoundation_[TypeName]__mdt`
   - Platform Events: `SFFoundation_[EventName]__e`
2. **Data Governance**:
   - Configuración centralizada via Custom Metadata
   - Logs con retención automática
   - Validaciones a nivel de campo y objeto
3. **Security**:
   - Field-level security apropiada
   - Sharing rules para data isolation
   - Encryption para campos sensibles
4. **Monitoring**:
   - Dashboards para métricas key

- Alertas automáticas para errores
- Reports de performance

Límites y Consideraciones

- **Governor Limits**: Considerar límites de DML, SOQL y heap
- **Storage**: Monitorear uso de data storage
- **API Limits**: Gestionar callouts hacia servicios externos
- **Platform Events**: Respetar límites de publicación

Fecha de Actualización: Mayo 2025**Versión del Documento:** 1.0**Arquitectos de Datos:** Soulberto Lorenzo, Jean Carlos Melendez**Estado:** Documento Vivo - Actualización Continua

`ProductHandler` Apex Class Documentation

Author: Jean Carlos Melendez (jean@cloudblue.us (mailto:jean@cloudblue.us)) **Last Modified By:** Jean Carlos Melendez**Last Modified On:** 2025-04-23

📌 Overview

`ProductHandler` es una clase abstracta que define el comportamiento base para una cadena de responsabilidad (Chain of Responsibility) en el procesamiento de productos dentro de una arquitectura basada en eventos, observadores y banderas de características (`FeatureFlags`).

Su objetivo principal es permitir a los desarrolladores extender la lógica de procesamiento de productos de manera flexible y desacoplada, implementando manejadores específicos que heredan de esta clase.

🔧 Class Definition

```
global abstract class ProductHandler extends Subject
```

Esta clase:

- **Es global**: puede ser accedida desde cualquier contexto.
- **Es abstracta**: no puede ser instanciada directamente.
- **Extiende `Subject`**: lo que implica que implementa el patrón de observador (Observer pattern).

🧩 Properties

Propiedad	Tipo	Acceso	Descripción
`productName`	`String`	global	Nombre del producto que el handler maneja.
`processName`	`String`	global	Nombre del proceso que se está ejecutando.
`nextHandler`	`ProductHandler`	global	Referencia al siguiente handler en la cadena.
`responseProcess`	`Object`	public	Respuesta generada por el proceso del handler.
`requestInput`	`Object`	global	Entrada original que recibe el handler.
`state`	`String`	public	Estado del procesamiento (`success` o `error`).
`FLAG_PLATFORM_NAME`	`String` (constante)	static	Bandera para habilitar eventos de plataforma.

🏗️ Constructor

ProductHandler(String productName)

Inicializa una nueva instancia del handler con el nombre del producto que procesará.

🔁 Chain of Responsibility

setNext(ProductHandler nextHandler)

Asigna el siguiente handler en la cadena.

Retorna: `ProductHandler` (el handler siguiente).

⚙️ Métodos de Configuración

setProcessName(String processName)

Define el nombre del proceso actual.

setResponseProcess(Object responseProcess)

Guarda la respuesta generada por el handler.

setRequestInput(Object requestInput)

Guarda la entrada del request en la propiedad del handler.

👀 Observadores

setObservers()

Agrega observadores al handler. Por defecto, se agrega un `BaseObserver`.

🎯 Lógica de Ejecución

shouldSkipHandler(Map options, Map input, Map output)

Evalúa si el handler debe ser omitido según:

- Si `productName` está presente en el `input`.
- Si el `productName` coincide con el que maneja este handler.

Retorna: `Boolean`

- `true` si debe ser omitido.
- `false` si debe ejecutar su lógica.

🔄 Proceso Principal

apply(Map options, Map input, Map output)

Este es el método principal del handler. Ejecuta las siguientes fases:

1. Verifica si debe omitir el handler (`shouldSkipHandler`).
2. Configura los observadores.
3. Evalúa la `FeatureFlag` para eventos de plataforma.
4. Ejecuta el preprocesamiento (`onPre`).
5. Ejecuta el procesamiento principal (`process`).
6. Ejecuta el posprocesamiento (`onPos`).
7. Notifica observadores.
8. Publica log del servicio (`publishServiceEventLog`).
9. Captura y reporta errores personalizados (`ProductHandlerImplementationException`).

🛠️ Métodos de Fase de Proceso

Método	Descripción	Entrada	Retorno
`process`	Ejecuta la lógica principal del handler	`options`, `input`, `output`	`Map<String, Object>`
`onPre`	Lógica previa al procesamiento principal	`options`, `input`, `output`	`Map<String, Object>`
`onPos`	Lógica posterior al procesamiento principal	`options`, `input`, `output`	`Map<String, Object>`

⚠️ Estos métodos son virtuales y deben ser sobreescritos en las clases hijas.

🧻 Publicación de Logs

publishServiceEventLog(String executionResult)

Publica un log asincrónico si la bandera `sura_foundation_event_logs` está habilitada. Incluye detalles del producto, respuesta serializada, proceso y plataforma.

⚠️ Excepciones

Manejo específico para `ProductHandlerImplementationException`, cuyos datos se añaden al `output`:

- `errorMsg`
- `errorCode`
- `error`

También se registra el estado como `"error"` y se serializa el mensaje de error.

🔄 Ejemplo de Cadena de Handlers

```
ProductHandler handlerA = new SomeProductHandler('ProductoA');
ProductHandler handlerB = new SomeProductHandler('ProductoB');

handlerA.setNext(handlerB);

Map<String, Object> input = new Map<String, Object>{ 'productName' => 'ProductoA' };
Map<String, Object> output = new Map<String, Object>();

handlerA.apply(new Map<String, Object>(), input, output);
```

📚 Recomendaciones

- Extiende esta clase e implementa `process`, `onPre` y `onPos` en tus propias clases de producto.
- Utiliza `FeatureFlags` para activar funcionalidades específicas sin modificar código productivo.
- Encadena handlers para distribuir la lógica por tipo de producto.
-

LeadProductHandler Apex Class Documentation

Author: Soulberto Lorenzo (soulberto@cloudblue.us (mailto:soulberto@cloudblue.us)) **Last Modified By**: Jean Carlos Melendez**Last Modified On**: 2025-04-25

📌 Overview

`LeadProductHandler` es una clase que extiende `ProductHandler` y representa un manejador específico para el producto tipo Lead. Esta clase implementa la lógica de creación de registros `Lead` en Salesforce usando un patrón de configuración flexible a través del constructor `LeadBuilderFoundation`.

También define validaciones específicas y observadores dedicados (`LeadObserver`) para su propio proceso de negocio.

🔧 Class Definition

```
global virtual class LeadProductHandler extends ProductHandler
```

Esta clase:

- **Es global y virtual**: puede ser accedida globalmente y sobreescrita si es necesario.
- **Extiende `ProductHandler`**: hereda todo el comportamiento de la cadena de responsabilidad.

🧩 Properties

Propiedad	Tipo	Acceso	Descripción
`PROCESS_NAME`	`String`	static final	Constante que define el nombre del proceso: `"LEAD"`

📐 Constructor

LeadProductHandler(String productName)

Inicializa una instancia del handler para un producto tipo Lead.

👀 Observadores

setObservers()

Sobrescribe el método base para asignar el `LeadObserver` como observador del handler.

⚙️ Métodos de Configuración

Estos métodos sobrescriben los definidos en la clase base para garantizar coherencia:

Método	Descripción
`setProcessName`	Define el nombre del proceso (`LEAD`).
`setResponseProcess`	Asigna el objeto de respuesta del handler.
`setRequestInput`	Almacena el objeto de entrada que será procesado.

🔄 Proceso Principal

process(Map options, Map input, Map output)

Este método ejecuta la lógica principal del handler para crear un `Lead` usando `LeadBuilderFoundation`.

Flujo de ejecución:

1. Verifica si se debe omitir el comportamiento por defecto con la opción `skipDefaultBehavior`.
2. Configura el nombre del proceso como `"LEAD"`.
3. Extrae los configuradores (`LeadConfigurator`) del input.
4. Construye el objeto `Lead` mediante el patrón Builder.
5. Agrega el resultado al `output` y lo retorna.

Manejo de errores:

- Captura y transforma excepciones `DmlException` y `Exception` en `ProductHandlerImplementationException` con código HTTP 422.

Retorna: Un `Map<String, Object>` con el resultado del proceso (`resultProcess`).

✅ Validación de Reglas de Negocio

isValidForLeadRequirements(Map<String, Object> data)

Método virtual para definir las validaciones de negocio específicas que deben cumplirse antes de crear un `Lead`. En esta implementación retorna siempre `true`, pero está diseñado para ser extendido.

Retorna: `Boolean`

🎭 Ejemplo de Uso

```
LeadProductHandler leadHandler = new LeadProductHandler('ProductoLead');

Map<String, Object> input = new Map<String, Object>{
  'status' => 'Nuevo',
  'firstName' => 'Ana',
  'lastName' => 'García',
  'email' => 'ana.garcia@example.com',
  'company' => 'Empresa XYZ'
};

Map<String, Object> output = new Map<String, Object>();

leadHandler.apply(new Map<String, Object>(), input, output);

System.debug(output.get('resultProcess')); // Muestra el Lead creado
```

📝 Consideraciones

- `LeadBuilderFoundation` encapsula la lógica de creación del `Lead`.
- `LeadConfigurator` permite extender dinámicamente la configuración del Lead.
- Utiliza excepciones personalizadas para controlar errores del proceso.

LeadConfigurator Apex Interface Documentation

Author: Jean Carlos Melendez**Last Modified By**: Jean Carlos Melendez**Last Modified On**: 2025-01-30

📌 Overview

`LeadConfigurator` es una interfaz global en Apex que define un contrato para aplicar configuraciones personalizadas a registros de tipo `Lead`. Su propósito es permitir la extensión modular del proceso de creación de Leads, brindando flexibilidad y encapsulamiento para adaptar configuraciones específicas según el contexto o negocio.

🔧 Interface Definition

```
global interface LeadConfigurator {
  void configure(Lead lead);
}
```

🧩 Method Summary

Método	Tipo de Retorno	Descripción
`configure`	`void`	Método obligatorio que implementa la lógica de configuración sobre un objeto `Lead`.

💉 Method Details

configure(Lead lead)

Descripción: Este método debe ser implementado por cualquier clase que desee proporcionar lógica personalizada de configuración sobre un objeto `Lead`. Puede ser utilizado, por ejemplo, para asignar valores a campos adicionales, ejecutar validaciones o asociar datos relacionados al Lead durante su construcción.

Parámetros:

Nombre	Tipo	Descripción
`lead`	`Lead`	Instancia de `Lead` sobre la cual se aplicarán las configuraciones.

Retorno: No retorna ningún valor (`void`), pero modifica directamente el objeto `Lead` recibido por referencia.

📚 Ejemplo de Implementación

```
global class CountryConfigurator implements LeadConfigurator {
  public void configure(Lead lead) {
    lead.Country = 'Colombia';
  }
}
```

📝 Consideraciones

- Esta interfaz permite desacoplar las configuraciones específicas del proceso de creación de Leads.
- Es útil cuando se construyen múltiples leads con diferentes reglas de negocio.
- Puede ser utilizada en conjunto con patrones como Builder o Chain of Responsibility.

🧠 Recomendación

Implementa múltiples clases `LeadConfigurator` para cubrir distintos aspectos del negocio y agrúpalas como una lista en el `LeadBuilderFoundation` o clases similares, mejorando así la mantenibilidad y escalabilidad del sistema.

RateProductHandler Apex Class Documentation

Author: Soulberto Lorenzo (soulberto@cloudblue.us (mailto:soulberto@cloudblue.us)) **Last Modified By**: Jean Carlos Melendez**Last Modified On**: 2025-04-23

📌 Overview

`RateProductHandler` es una clase que extiende `ProductHandler` y está diseñada para manejar la tarificación de productos en Salesforce. Utiliza un Integration Procedure (IP) para calcular tarifas basadas en parámetros de entrada, gestionando tanto la validación como la ejecución del procedimiento y el formateo de resultados.

🔧 Class Definition

```
global virtual class RateProductHandler extends ProductHandler
```

🧩 Properties

Propiedad	Tipo	Acceso	Descripción
`PROCESS_NAME`	`String`	`static final`	Nombre del proceso: `"TARIFICACION"`
`OPTION_SKIP_BEHAVIOR`	`String`	`static final`	Opción para omitir la ejecución normal del handler.
`RATING_IPROCEDURE`	`String`	`static final`	Nombre del Integration Procedure utilizado para la tarificación.
`EXECUTOR`	`IntegrationProcedureExecutor`	`global`	Ejecuta el procedimiento de integración.
`REQUIRED_PARAMETERS`	`Set<String>`	`static final`	Parámetros obligatorios para ejecutar el IP.

📐 Constructores

RateProductHandler(String productName, IntegrationProcedureExecutor executor)

Crea una instancia del handler con un ejecutor personalizado.

RateProductHandler(String productName)

Crea una instancia con un ejecutor simulado (`MockIntegrationProcedureExecutor`), útil para pruebas.

⚙️ Métodos de Configuración

Método	Descripción
`setProcessName(String)`	Asigna el nombre del proceso.
`setResponseProcess(Object)`	Guarda la respuesta generada.
`setRequestInput(Object)`	Guarda los datos de entrada procesados.

✅ Validación de Parámetros

isValidForRatingParameters(Map<String, Object> data)

Verifica que todos los parámetros definidos en `REQUIRED_PARAMETERS` estén presentes en el `input`.

Retorna: `true` si todos están presentes, `false` si falta alguno.

🔁 Método Principal

process(Map options, Map input, Map output)

Este método es el núcleo de la lógica del handler. Valida, construye y ejecuta el IP.

Retorno: `Map<String, Object>` — Resultado del proceso de tarificación.

Excepción: Lanza `ProductHandlerImplementationException` si la validación falla.

🔧 Utilidades

buildInputMap(Map input, Map options)

Combina el `input` original con las `options` para generar el mapa que se enviará al IP.

⚙️ Ejecución del Integration Procedure

executeIntegrationProcedure(Map ipInput)

Ejecuta el Integration Procedure definido en `RATING_IPROCEDURE` usando el `EXECUTOR`.

📚 Ejemplo de Uso

```
RateProductHandler handler = new RateProductHandler('ProductoX');

Map<String, Object> input = new Map<String, Object>{
  'includeInputKeys' => true,
  'instanceKey' => 'INST001',
  'filters' => new Map<String, Object>{ 'edad' => 35 }
};

Map<String, Object> output = new Map<String, Object>();

handler.apply(new Map<String, Object>(), input, output);

System.debug(output.get('resultProcess'));
```

📝 Consideraciones

- Este handler se adapta perfectamente a arquitecturas basadas en IP.
- Recomendado sobrescribir `setObservers()` para agregar `RateObserver`.
- Bien estructurado para mantenibilidad y pruebas.

IntegrationProcedureExecutor Apex Interface Documentation

Author: Jean Carlos Melendez**Last Modified By**: Jean Carlos Melendez**Last Modified On**: 2025-01-28

📌 Overview

`IntegrationProcedureExecutor` es una interfaz global que define el contrato para ejecutar procedimientos de integración en Salesforce.

🔧 Interface Definition

```
global interface IntegrationProcedureExecutor {
  Map<String, Object> execute(String procedureName, Map<String, Object> input);
}
```

🧩 Method Summary

Método	Tipo de Retorno	Descripción
execute	Map<String, Object>	Ejecuta un Integration Procedure y retorna los resultados.

📥 Method Details

execute(String procedureName, Map<String, Object> input)

Ejecuta un IP con el nombre y mapa de entrada especificado.

Parámetros:

Nombre	Tipo	Descripción
procedureName	String	Nombre del IP a ejecutar.
input	Map<String, Object>	Datos de entrada.

Retorna: Map<String, Object> — Respuesta del IP.

📚 Ejemplo de Implementación

```
goblal class VlocityIntegrationProcedureExecutor implements IntegrationProcedureExecutor {
  public Map<String, Object> execute(
    String procedureName,
    Map<String, Object> input
  ) {

    return (Map<String, Object>) vlocity_ins.IntegrationProcedureService.runIntegrationService(
      procedureName,
      input,
      new Map<String, Object>()
    );
  }
}
```

📝 Consideraciones

- Ideal para pruebas y desacoplamiento de lógica de negocio.
- Facilita el cumplimiento del principio de inversión de dependencias (SOLID).

QuoteProductHandler Apex Class Documentation

Author: Soulberto Lorenzo (soulberto@cloudblue.us (mailto:soulberto@cloudblue.us)) **Last Modified By**: Jean Carlos Melendez**Last Modified On**: 2025-04-02

📌 Overview

`QuoteProductHandler` es una implementación concreta de `ProductHandler` que permite procesar cotizaciones de productos mediante un `IntegrationProcedure` especializado. Esta clase incluye validaciones estrictas de los datos de entrada, ejecuta un procedimiento de cotización y retorna los resultados, integrando observadores y manejando excepciones adecuadamente.

🔧 Class Definition

```
global virtual class QuoteProductHandler extends ProductHandler
```

Esta clase:

- Es global y virtual, permitiendo su extensión.
- Hereda de `ProductHandler`.
- Está diseñada para manejar lógica de cotización de seguros u otros productos complejos.

🧩 Properties

Propiedad	Tipo	Acceso	Descripción
`PROCESS_NAME`	`String`	`static final`	Nombre del proceso: `"COTIZACION"`
`MAIN_NODE`	`String`	`static final`	Nodo principal esperado en el input: `"quotepolicyJson"`
`OPTION_SKIP_BEHAVIOR`	`String`	`static final`	Clave para omitir el procesamiento por defecto.
`RATING_IPROCEDURE`	`String`	`static final`	Nombre del Integration Procedure para cotización.
`EXECUTOR`	`IntegrationProcedureExecutor`	`global`	Ejecuta el procedimiento de cotización.
`REQUIRED_PARAMETERS`	`Set<String>`	`static final`	Nodos requeridos dentro de `MAIN_NODE`.

🧬 Constructores

QuoteProductHandler(String productName, IntegrationProcedureExecutor executor)

Crea una instancia del handler utilizando un executor personalizado para ambientes productivos.

QuoteProductHandler(String productName)

Crea una instancia usando un `MockIntegrationProcedureExecutor`, ideal para pruebas.

👀 Observadores

setObservers()

Registra el observador `QuoteObserver` para recibir notificaciones sobre el proceso.

⚙️ Métodos de Configuración

Método	Descripción
`setProcessName(String)`	Asigna el nombre del proceso al handler.
`setResponseProcess(Object)`	Guarda la respuesta procesada.
`setRequestInput(Object)`	Guarda los datos de entrada para el proceso.

🔄 Proceso Principal

process(Map<String, Object> options, Map<String, Object> input, Map<String, Object> output)

Este método orquesta el flujo completo del proceso de cotización.

Pasos:

1. Omite el procesamiento si se incluye la opción `skipDefaultBehavior`.
2. Define el nombre del proceso como `"COTIZACION"`.
3. Valida que todos los nodos requeridos estén presentes.
4. Ejecuta el `IntegrationProcedure` para generar la cotización.
5. Asigna el resultado al mapa `output` y lo serializa como respuesta.

Parámetros:

- `options`: Opciones adicionales del proceso.
- `input`: Datos de entrada (incluye `quotepolicyJson`).
- `output`: Mapa de salida que será completado con el resultado.

Retorna: `Map<String, Object>` — Resultado del proceso de cotización.

Excepciones: Lanza `ProductHandlerImplementationException` si la validación falla.

🧪 Validación de Entrada

validateInputMapForQuoting(Map inputMap)

Verifica que el nodo principal y los nodos internos requeridos estén presentes en `quotepolicyJson`.

Nodos requeridos:

- `term`
- `productConfigurationDetail`
- `insuredItems`
- `additionalFields`
- `OpportunityDetails`

Retorna: `Map<Boolean, Object>` — Resultado estándar de validación con mensaje descriptivo.

🧱 Utilidades

createValidationResult(Boolean isValid, String message)

Crea un resultado de validación estructurado en forma de mapa.

Ejemplo de retorno:

```
{ true => 'Todos los parámetros requeridos están presentes.' }
```

⚙️ Ejecución de Integration Procedure

executeIntegrationProcedure(Map ipInput)

Llama al executor con el nombre definido (`sfcore_quotingprocedure`) y los datos de entrada.

Retorna:

```
{ 'resultProcess' => <resultado del executor> }
```

📘 Ejemplo de Uso

```
QuoteProductHandler quoteHandler = new QuoteProductHandler('ProductoCotizacion');

Map<String, Object> input = new Map<String, Object>{
  'quotepolicyJson' => new Map<String, Object>{
    'term' => '12M',
    'productConfigurationDetail' => 'PCD001',
    'insuredItems' => new List<Object>{},
    'additionalFields' => new Map<String, Object>(),
    'OpportunityDetails' => new Map<String, Object>()
  }
};

Map<String, Object> output = new Map<String, Object>();

quoteHandler.apply(new Map<String, Object>(), input, output);

System.debug(output.get('resultProcess'));
```

📝 Consideraciones

- Ideal para procesos de cotización que requieren integración con sistemas externos mediante IPs.
- Es completamente extensible y puede integrarse con distintos observers para monitoreo.
- Utiliza un patrón de validación clara para evitar errores por falta de estructura.

IssueProductHandler Apex Class Documentation

Author: Jean Carlos Melendez (jean@cloudblue.us (mailto:jean@cloudblue.us)) **Last Modified By:** Jean Carlos Melendez**Last Modified On:** 2025-04-02

📌 Overview

`IssueProductHandler` es una implementación de `ProductHandler` responsable de manejar el proceso de emisión de productos. Utiliza un Integration Procedure para realizar la lógica de negocio asociada a la emisión, validando los datos de entrada, realizando preprocesamiento, ejecutando el IP y actualizando el estado de la oportunidad relacionada.

🔧 Class Definition

```
global virtual class IssueProductHandler extends ProductHandler
```

🧩 Properties

Propiedad	Tipo	Acceso	Descripción
`FLAG_PLATFORM_NAME`	`String`	`static final`	Nombre de la Feature Flag para eventos de plataforma.
`IP_PREPROCESS`	`String`	`static final`	Nombre del IP de preprocesamiento.
`PROCESS_NAME`	`String`	`static final`	Nombre del proceso manejado: `"TARIFICACION"`
`OPTION_SKIP_BEHAVIOR`	`String`	`static final`	Clave para omitir comportamiento por defecto.
`ISSUING_IPROCEDURE`	`String`	`static final`	Nombre del IP utilizado para la emisión.
`ERROR_REQUIRED_FIELD`	`String`	`static final`	Mensaje de error usado en validación de campos.
`EXECUTOR`	`IntegrationProcedureExecutor`	`global`	Objeto encargado de ejecutar el Integration Procedure.
`REQUIRED_FIELD_FOR_ISSUE_CORE`	`List<String>`	`static final`	Lista de campos obligatorios para ejecutar la emisión.

📐 Constructores

IssueProductHandler(String productName, IntegrationProcedureExecutor executor)

Inicializa el handler con un ejecutor personalizado.

IssueProductHandler(String productName)

Inicializa con un `MockIntegrationProcedureExecutor`.

⚙️ Métodos de Configuración

Método	Descripción
`setProcessName(String)`	Define el nombre del proceso.
`setResponseProcess(Object)`	Guarda la respuesta del proceso.
`setRequestInput(Object)`	Guarda los datos de entrada originales.

🔄 Métodos del Proceso

onPre(Map options, Map input, Map output)

Realiza validaciones previas y asigna las cuentas aseguradas a la cotización.

process(Map options, Map input, Map output)

Valida los datos requeridos, ejecuta la validación previa (`preIssuingValidation`) y llama al IP de emisión.

onPos(Map options, Map input, Map output)

Actualiza la etapa de la oportunidad relacionada a `'Poliza emitida'`.

executeIntegrationProcedure(Map ipInput)

Ejecuta el IP configurado (`SURAFoundation_IssuingProcedure`) con los datos de entrada validados.

✅ Validaciones

validateInput(Map input, List<String> requiredFields)

Valida que los campos obligatorios estén presentes en el mapa `input`. Lanza excepción si alguno está ausente.

preIssuingValidation(Map input)

Valida los campos clave para emisión (`quoteId`, `effectiveDate`, `producerId`) y garantiza que la cotización y la oportunidad asociada existan y estén completas.

🔧 Métodos Auxiliares

assignAccountsToQuote(List<Map<String, Object>> parties, Id quoteId)

Asocia las cuentas aseguradas al objeto Quote y su oportunidad.

listInsuredPartyAccounts(Map<String, Object> input)

Convierte la información del lead a cuentas aseguradas mediante `LeadUtils`.

📦 Ejemplo de Ejecución

```
IssueProductHandler handler = new IssueProductHandler('ProductoX');

Map<String, Object> input = new Map<String, Object>{
  'leadId' => '00Qxx0000001abc',
  'quoteId' => '0Q0xx0000002def',
  'effectiveDate' => '2025-04-15',
  'producerId' => '001xx000003xyzA'
};

Map<String, Object> output = new Map<String, Object>();

handler.apply(new Map<String, Object>(), input, output);
```

```
System.debug(output.get('resultProcess'));
```

📝 Consideraciones

- El método `onPre` asegura que las cuentas estén correctamente asociadas antes de emitir.
- `onPos` actualiza automáticamente el estado de la oportunidad una vez finalizada la emisión.
- Utiliza validaciones sólidas para garantizar que los datos estén completos antes de ejecutar el IP.
- El handler puede ser fácilmente extendido o simulado en pruebas gracias al uso de `IntegrationProcedureExecutor`.

ResponsabilityHandler

Descripción General

`ResponsabilityHandler` es una clase abstracta que implementa el patrón de diseño Chain of Responsibility (Cadena de Responsabilidad). Esta clase extiende `Subject`, lo que sugiere que también implementa el patrón Observer. La clase está diseñada para procesar objetos `Lead` según planes seleccionados, permitiendo encadenar múltiples handlers para formar un flujo de procesamiento.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

2 de septiembre de 2024

Características de Accesibilidad

La clase y todos sus miembros están anotados con `@namespaceAccessible`, lo que permite que sean accesibles desde otros namespaces en organizaciones con paquetes gestionados.

Atributos

handlerName

- **Tipo**: `String`
- **Accesibilidad**: `public`
- **Descripción**: Nombre identificativo del handler.

nextHandler

- **Tipo**: `ResponsabilityHandler`
- **Accesibilidad**: `public`
- **Descripción**: Referencia al siguiente handler en la cadena de responsabilidad.

Constructor

ResponsabilityHandler(String handlerName)

Descripción

Constructor que inicializa un nuevo handler con un nombre específico.

Parámetros

- `handlerName` (String): Nombre identificativo para el handler.

Código

```
@namespaceAccessible
public ResponsabilityHandler(String handlerName) {
  this.handlerName = handlerName;
}
```

Métodos

setNext(ResponsabilityHandler nextHandler)

Descripción

Establece el siguiente handler en la cadena de responsabilidad y lo devuelve, permitiendo un encadenamiento fluido de métodos.

Parámetros

- `nextHandler` (ResponsabilityHandler): El siguiente handler en la cadena.

Retorno

- `ResponsabilityHandler`: El handler proporcionado, permitiendo encadenamiento de métodos.

Código

```
@namespaceAccessible
public ResponsabilityHandler setNext(ResponsabilityHandler nextHandler) {
  this.nextHandler = nextHandler;
  return nextHandler;
}
```

apply(Lead lead, String selectedPlan)

Descripción

Método virtual que debe ser implementado por las clases concretas derivadas. Este método aplicará la lógica de procesamiento específica del handler a un objeto Lead y un plan seleccionado.

Parámetros

- `lead` (Lead): El objeto Lead a procesar.
- `selectedPlan` (String): El plan seleccionado que puede determinar el tipo de procesamiento.

Retorno

- `void`

Comportamiento por Defecto

En la implementación base, este método lanza una excepción `ProductHandlerNotImplementedException`, indicando que las clases derivadas deben sobrescribir este método con su implementación específica.

Código

```
@namespaceAccessible
public virtual void apply(Lead lead, String selectedPlan) {
  throw new ProductHandlerNotImplementedException(
    'Product Handler must be implemented!'
  );
}
```

Excepciones

ProductHandlerNotImplementedException

Esta excepción se lanza cuando una clase derivada no ha implementado correctamente el método `apply()`. Indica un error en el diseño o implementación de la cadena de responsabilidad.

Patrones de Diseño Implementados

Chain of Responsibility (Cadena de Responsabilidad)

Este patrón permite que múltiples objetos procesen una solicitud. La clase `ResponsabilityHandler` establece la estructura básica para crear una cadena de handlers, donde cada uno puede procesar la solicitud o pasarla al siguiente en la cadena.

Observer (Observador)

Al extender la clase `Subject`, `ResponsabilityHandler` hereda la capacidad de notificar a los objetos observadores sobre cambios o eventos durante el procesamiento.

Dependencias

- `Subject`: Clase base que implementa el patrón Observer.
- `ProductHandlerNotImplementedException`: Excepción personalizada lanzada cuando el método `apply()` no está implementado.
- `Lead`: Objeto estándar de Salesforce utilizado en el método `apply()`.

Ejemplo de Uso

```
// Definir handlers concretos
public class LeadQualificationHandler extends ResponsabilityHandler {
    public LeadQualificationHandler() {
        super('LeadQualification');
    }

    public override void apply(Lead lead, String selectedPlan) {
        // Lógica para calificar el lead
        lead.Status = 'Qualified';
        lead.Rating = 'Hot';

        // Notificar a los observadores
        this.notifyObservers(lead);

        // Continuar con el siguiente handler si existe
        if (this.nextHandler != null) {
            this.nextHandler.apply(lead, selectedPlan);
        }
    }
}

public class LeadAssignmentHandler extends ResponsabilityHandler {
    public LeadAssignmentHandler() {
        super('LeadAssignment');
    }

    public override void apply(Lead lead, String selectedPlan) {
        // Lógica para asignar el lead según el plan
        if (selectedPlan == 'Premium') {
            lead.OwnerId = UserInfo.getUserId(); // Asignar al usuario actual como ejemplo
        }

        // Notificar a los observadores
        this.notifyObservers(lead);

        // Continuar con el siguiente handler si existe
        if (this.nextHandler != null) {
            this.nextHandler.apply(lead, selectedPlan);
```

```
        }
    }
}

// Configurar y utilizar la cadena de responsabilidad
ResponsabilityHandler qualificationHandler = new LeadQualificationHandler();
ResponsabilityHandler assignmentHandler = new LeadAssignmentHandler();

// Encadenar los handlers
qualificationHandler.setNext(assignmentHandler);

// Procesar un lead
Lead newLead = new Lead(
    FirstName = 'John',
    LastName = 'Doe',
    Company = 'ACME Inc.'
);
qualificationHandler.apply(newLead, 'Premium');
```

Notas Adicionales

1. Esta clase abstracta está diseñada para ser extendida por clases concretas que implementen la lógica específica de procesamiento.
2. El patrón Chain of Responsibility implementado aquí permite una separación clara de responsabilidades y facilita la adición de nuevos handlers sin modificar el código existente.
3. La combinación con el patrón Observer (a través de la herencia de `Subject`) permite que otros objetos sean notificados durante el procesamiento, facilitando acciones adicionales o registro de eventos.
4. Aunque el método `apply()` lanza una excepción por defecto, se espera que las clases derivadas sobrescriban este método con su propia implementación.
5. La anotación `@namespaceAccessible` en todos los elementos de la clase indica que está diseñada para ser utilizada en un contexto de paquetes gestionados, permitiendo que clases en otros namespaces puedan extenderla y utilizarla.

FoundationEventLogController

Descripción General

`FoundationEventLogController` es una clase que actúa como controlador para componentes Lightning Web Components (LWC) o Aura Components, proporcionando funcionalidad de gestión de registros de eventos y trabajos programados. Esta clase expone métodos anotados con `@AuraEnabled` para permitir la interacción desde el frontend con las funcionalidades de análisis de eventos y programación de tareas de limpieza.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

7 de marzo de 2025 por "ChangeMeIn@UserSettingsUnder.SFDoc"

Estructura de la Clase

```
public class FoundationEventLogController {
  @AuraEnabled public static Boolean refreshAnalytics(Datetime startAt, Datetime endAt);
  @AuraEnabled public static Boolean cleanAnalytics();
  @AuraEnabled public static eventInfoWrapper loadScheduleInfo();
  @AuraEnabled public static void refreshScheduleInfo(String cronExp, List<String> timeParts);
  @AuraEnabled public static boolean enabledScheduleInfo(Boolean enabled);

  public class eventInfoWrapper {
    @AuraEnabled public Boolean isActive;
    @AuraEnabled public Datetime nextFireTime;
    @AuraEnabled public Datetime previousFireTime;
    @AuraEnabled public String cronExp;
    @AuraEnabled public Integer numberOfExecutions;
  }
}
```

Métodos

refreshAnalytics(Datetime startAt, Datetime endAt)

Descripción

Método que inicia el proceso de actualización de análisis de eventos de forma asíncrona para un rango de fechas específico.

Parámetros

- `startAt` (Datetime): Fecha y hora de inicio del rango para el análisis.
- `endAt` (Datetime): Fecha y hora de fin del rango para el análisis.

Retorno

- `Boolean`: `true` si el proceso se inició exitosamente, `false` en caso de error.

Proceso

1. Valida que las fechas de inicio y fin no sean nulas.
2. Si alguna fecha es nula, registra un mensaje de depuración y usa la fecha/hora actual como valor por defecto.

3. Llama al método `refreshAnalyticsAsync` de `EventLogManager`.

4. Maneja excepciones y retorna el resultado.

Código

```
@AuraEnabled
public static Boolean refreshAnalytics(Datetime startAt, Datetime endAt) {
  try {
    if (startAt == null || endAt == null)
      Core.debug('start Date or End Date is coming null');
    EventLogManager.refreshAnalyticsAsync(
      startAt ?? Datetime.now(),
      endAt ?? Datetime.now(),
      false
    );
  } catch (Exception e) {
    Core.debug('Something has failed refreshing Analytics' + e.getMessage());
    return false;
  }
  return true;
}
```

cleanAnalytics()

Descripción

Método que ejecuta la limpieza de datos analíticos eliminando registros del objeto `EventLog_Subset__c`.

Parámetros

Ninguno.

Retorno

- `Boolean`: `true` si la limpieza fue exitosa, `false` en caso contrario.

Proceso

1. Llama al método `clearAnalytics` de `EventLogManager`.

2. Retorna el resultado de la operación directamente.

Código

```
@AuraEnabled
public static Boolean cleanAnalytics() {
  Boolean result = false;
  result = EventLogManager.clearAnalytics();
  return result;
}
```

loadScheduleInfo()

Descripción

Método que carga y retorna información sobre la configuración del trabajo programado de limpieza de eventos, incluyendo detalles del cronograma y estado de ejecución.

Parámetros

Ninguno.

Retorno

- `eventInfoWrapper`: Objeto wrapper que contiene información completa sobre el trabajo programado.

Proceso

1. Crea una instancia del wrapper de información de eventos.
2. Si se ejecuta en contexto de prueba, crea configuración mock.
3. Si no es prueba, consulta la configuración desde metadatos personalizados.
4. Extrae información básica (estado activo, expresión cron).
5. Si existe un ID de trabajo programado, consulta detalles del CronTrigger.
6. Retorna toda la información encapsulada en el wrapper.

Código

```
@AuraEnabled
public static eventInfoWrapper loadScheduleInfo() {
  eventInfoWrapper info = new eventInfoWrapper();

  Suratech_Foundation_Schedule_Event__mdt config;
  if (Test.isRunningTest()) {
    Id testJobId = System.schedule(
      'Test Event Log Cleanup',
      '0 00 02 ? * WED,THU,FRI *',
      new EventLog_Subset_CleanupScheduleClass()
    );
    config = new Suratech_Foundation_Schedule_Event__mdt(
      MasterLabel = 'DefaultConfig',
      DeveloperName = 'DefaultConfig',
      Frequency__c = '0 00 02 ? * WED,THU,FRI *',
      Is_Active__c = true,
      Schedule_Job_Id__c = testJobId
    );
  } else {
    config = [
      SELECT
        Id,
        Frequency__c,
        Schedule_Job_Id__c,
        MasterLabel,
        DeveloperName,
        Is_Active__c
      FROM Suratech_Foundation_Schedule_Event__mdt
      WHERE DeveloperName = 'DefaultConfig'
      LIMIT 1
    ];
  }

  info.isActive = config.Is_Active__c;
  info.cronExp = config.Frequency__c;

  if (!Test.isRunningTest() && config.Id == null) {
    throw new MetadataNotFoundException(
      'No Suratech_Foundation_Schedule_Event__mdt Custom Metadata found for "config"'
    );
  }
```

```
  if (config.Schedule_Job_Id__c != null) {
    CronTrigger ct = [
      SELECT
        Id,
        CronExpression,
        TimesTriggered,
        NextFireTime,
        PreviousFireTime
      FROM CronTrigger
      WHERE Id = :config.Schedule_Job_Id__c
    ];
    info.previousFireTime = ct.PreviousFireTime;
    info.nextFireTime = ct.NextFireTime;
    info.numberOfExecutions = ct.TimesTriggered;
  }
  return info;
}
```

refreshScheduleInfo(String cronExp, List<String> timeParts)

Descripción

Método que actualiza la configuración del trabajo programado, incluyendo la reprogramación con una nueva expresión cron.

Parámetros

- `cronExp` (String): Nueva expresión cron para la programación del trabajo.
- `timeParts` (List

  ): Partes de tiempo (no utilizado en la implementación actual).

Retorno

- `void`: Este método no devuelve ningún valor.

Proceso

1. Obtiene la configuración actual de metadatos personalizados.
2. Actualiza la frecuencia con la nueva expresión cron.
3. Si existe un trabajo programado anterior, lo cancela.
4. Programa un nuevo trabajo con la nueva expresión cron.
5. Actualiza la configuración con el nuevo ID de trabajo.
6. Guarda los cambios en metadatos (excepto en pruebas).

Código

```
@AuraEnabled
public static void refreshScheduleInfo(
  String cronExp,
  List<String> timeParts
) {
  System.debug('Eventos');

  Suratech_Foundation_Schedule_Event__mdt config;
  if (Test.isRunningTest()) {
```

```
    Id testJobId = System.schedule(
      'Test Event Log Cleanup',
      cronExp,
      new EventLog_Subset_CleanupScheduleClass()
    );
    config = new Suratech_Foundation_Schedule_Event__mdt(
      MasterLabel = 'DefaultConfig',
      DeveloperName = 'DefaultConfig',
      Frequency__c = '',
      Is_Active__c = true,
      Schedule_Job_Id__c = ''
    );
  } else {
    config = [
      SELECT
        Id,
        Frequency__c,
        Schedule_Job_Id__c,
        MasterLabel,
        DeveloperName,
        Is_Active__c
      FROM Suratech_Foundation_Schedule_Event__mdt
      WHERE DeveloperName = 'DefaultConfig'
      LIMIT 1
    ];
  }

  config.Frequency__c = cronExp;

  if (config.Id == null) {
    config.MasterLabel = 'DefaultConfig';
    config.DeveloperName = 'DefaultConfig';
  }

  if (config.Schedule_Job_Id__c != null || config.Schedule_Job_Id__c == '') {
    try {
      System.abortJob(config.Schedule_Job_Id__c);
    } catch (Exception e) {
      System.debug('ERROR with Abbort Job: ' + e.getMessage());
    }
  }

  String jobName = Test.isRunningTest()
    ? 'Old Event Log Cleanup'
    : 'Test Event Log Cleanup';
  String jobId = System.schedule(
    jobName,
    cronExp,
    new EventLog_Subset_CleanupScheduleClass()
  );
  config.Is_Active__c = true;
  config.Schedule_Job_Id__c = jobId;

  if (!Test.isRunningTest())
    MetadataUtils.upsertMetadata(config);
```

```
  System.debug('Job rescheduled with ID: ' + jobId);
}
```

enabledScheduleInfo(Boolean enabled)

Descripción

Método que habilita o deshabilita la configuración del trabajo programado (método incompleto en la implementación actual).

Parámetros

- `enabled` (Boolean): Indica si el trabajo debe estar habilitado o deshabilitado.

Retorno

- `Boolean`: `true` si la operación fue exitosa, `false` en caso de error.

Proceso

1. Obtiene la configuración de metadatos personalizados.
2. Intenta actualizar la configuración (implementación incompleta).
3. Maneja excepciones y retorna el resultado.

Nota: Este método parece estar incompleto, ya que no actualiza realmente el estado habilitado/deshabilitado.

Clase Wrapper

eventInfoWrapper

Descripción

Clase interna que encapsula información sobre el trabajo programado de eventos para ser utilizada por componentes del frontend.

Propiedades

- `isActive` (Boolean): Indica si el trabajo programado está activo.
- `nextFireTime` (Datetime): Próxima fecha y hora de ejecución del trabajo.
- `previousFireTime` (Datetime): Última fecha y hora de ejecución del trabajo.
- `cronExp` (String): Expresión cron que define la programación.
- `numberOfExecutions` (Integer): Número de veces que el trabajo ha sido ejecutado.

Dependencias

- **`EventLogManager`**: Clase que proporciona la funcionalidad principal de gestión de eventos.
- **`EventLog_Subset_CleanupScheduleClass`**: Clase que implementa la lógica del trabajo programado.
- **`Suratech_Foundation_Schedule_Event__mdt`**: Tipo de metadatos personalizados para configuración.
- **`MetadataUtils`**: Clase utilitaria para operaciones con metadatos.
- **`Core`**: Clase utilitaria para registro de depuración.
- **`MetadataNotFoundException`**: Excepción personalizada para casos donde no se encuentra la configuración.

Ejemplos de Uso

Ejemplo 1: Uso desde Lightning Web Component

```
// foundationEventLogManager.js
import { LightningElement, track, wire } from 'lwc';
import refreshAnalytics from '@salesforce/apex/FoundationEventLogController.refreshAnalytics';
import cleanAnalytics from '@salesforce/apex/FoundationEventLogController.cleanAnalytics';
import loadScheduleInfo from '@salesforce/apex/FoundationEventLogController.loadScheduleInfo';
import refreshScheduleInfo from '@salesforce/apex/FoundationEventLogController.refreshScheduleInfo';

export default class FoundationEventLogManager extends LightningElement {
    @track scheduleInfo = {};
    @track isLoading = false;
    @track startDate;
    @track endDate;
    @track cronExpression;

    connectedCallback() {
        this.loadScheduleInformation();
    }

    async loadScheduleInformation() {
        try {
            this.isLoading = true;
            this.scheduleInfo = await loadScheduleInfo();
            this.cronExpression = this.scheduleInfo.cronExp;
        } catch (error) {
            console.error('Error loading schedule info:', error);
        } finally {
            this.isLoading = false;
        }
    }

    async handleRefreshAnalytics() {
        if (!this.startDate || !this.endDate) {
            // Mostrar error de validación
            return;
        }

        try {
            this.isLoading = true;
            const result = await refreshAnalytics({
                startAt: new Date(this.startDate),
                endAt: new Date(this.endDate)
            });

            if (result) {
                // Mostrar mensaje de éxito
                this.showToast('Success', 'Analytics refresh started successfully', 'success');
            } else {
                // Mostrar mensaje de error
                this.showToast('Error', 'Failed to start analytics refresh', 'error');
            }
        } catch (error) {
            console.error('Error refreshing analytics:', error);
            this.showToast('Error', error.body.message, 'error');
        } finally {
```

```
            this.isLoading = false;
        }
    }

    async handleCleanAnalytics() {
        try {
            this.isLoading = true;
            const result = await cleanAnalytics();

            if (result) {
                this.showToast('Success', 'Analytics cleaned successfully', 'success');
            } else {
                this.showToast('Error', 'Failed to clean analytics', 'error');
            }
        } catch (error) {
            console.error('Error cleaning analytics:', error);
            this.showToast('Error', error.body.message, 'error');
        } finally {
            this.isLoading = false;
        }
    }

    async handleUpdateSchedule() {
        if (!this.cronExpression) {
            this.showToast('Error', 'Please provide a valid cron expression', 'error');
            return;
        }

        try {
            this.isLoading = true;
            await refreshScheduleInfo({
                cronExp: this.cronExpression,
                timeParts: []
            });

            this.showToast('Success', 'Schedule updated successfully', 'success');
            await this.loadScheduleInformation(); // Reload info
        } catch (error) {
            console.error('Error updating schedule:', error);
            this.showToast('Error', error.body.message, 'error');
        } finally {
            this.isLoading = false;
        }
    }

    handleStartDateChange(event) {
        this.startDate = event.target.value;
    }

    handleEndDateChange(event) {
        this.endDate = event.target.value;
    }

    handleCronExpressionChange(event) {
        this.cronExpression = event.target.value;
    }
```

```
    showToast(title, message, variant) {
        const event = new ShowToastEvent({
            title: title,
            message: message,
            variant: variant
        });
        this.dispatchEvent(event);
    }
}
```

```
<!-- foundationEventLogManager.html -->
<template>
    <lightning-card title="Foundation Event Log Manager" icon-name="standard:logging">
        <div class="slds-p-horizontal_small">
            <!-- Analytics Section -->
            <div class="slds-section slds-is-open">
                <h3 class="slds-section__title">
                    <span class="slds-truncate slds-p-horizontal_small" title="Analytics Management">
                        Analytics Management
                    </span>
                </h3>
                <div class="slds-section__content slds-p-around_medium">
                    <div class="slds-grid slds-gutters">
                        <div class="slds-col slds-size_1-of-2">
                            <lightning-input
                                type="datetime-local"
                                label="Start Date"
                                value={startDate}
                                onchange={handleStartDateChange}>
                            </lightning-input>
                        </div>
                        <div class="slds-col slds-size_1-of-2">
                            <lightning-input
                                type="datetime-local"
                                label="End Date"
                                value={endDate}
                                onchange={handleEndDateChange}>
                            </lightning-input>
                        </div>
                    </div>

                    <div class="slds-m-top_medium">
                        <lightning-button
                            variant="brand"
                            label="Refresh Analytics"
                            onclick={handleRefreshAnalytics}
                            disabled={isLoading}>
                        </lightning-button>

                        <lightning-button
                            variant="destructive"
                            label="Clean Analytics"
                            onclick={handleCleanAnalytics}
                            disabled={isLoading}
                            class="slds-m-left_small">
                        </lightning-button>
```

```
            </div>
        </div>
    </div>

    <!-- Schedule Section -->
    <div class="slds-section slds-is-open slds-m-top_large">
        <h3 class="slds-section__title">
            <span class="slds-truncate slds-p-horizontal_small" title="Schedule Management">
                Schedule Management
            </span>
        </h3>
        <div class="slds-section__content slds-p-around_medium">
            <div class="slds-grid slds-gutters">
                <div class="slds-col slds-size_1-of-2">
                    <lightning-input
                        label="Cron Expression"
                        value={cronExpression}
                        onchange={handleCronExpressionChange}
                        help="Example: 0 0 2 * * ? (Daily at 2 AM)">
                    </lightning-input>
                </div>
                <div class="slds-col slds-size_1-of-2">
                    <div class="slds-form-element">
                        <label class="slds-form-element__label">Schedule Status</label>
                        <div class="slds-form-element__control">
                            <lightning-badge
                                label={scheduleInfo.isActive ? 'Active' : 'Inactive'}
                                variant={scheduleInfo.isActive ? 'success' : 'error'}>
                            </lightning-badge>
                        </div>
                    </div>
                </div>
            </div>

            <template if:true={scheduleInfo.nextFireTime}>
                <div class="slds-grid slds-gutters slds-m-top_medium">
                    <div class="slds-col slds-size_1-of-3">
                        <div class="slds-form-element">
                            <label class="slds-form-element__label">Next Execution</label>
                            <div class="slds-form-element__control">
                                <lightning-formatted-date-time
                                    value={scheduleInfo.nextFireTime}>
                                </lightning-formatted-date-time>
                            </div>
                        </div>
                    </div>
                    <div class="slds-col slds-size_1-of-3">
                        <div class="slds-form-element">
                            <label class="slds-form-element__label">Last Execution</label>
                            <div class="slds-form-element__control">
                                <lightning-formatted-date-time
                                    value={scheduleInfo.previousFireTime}>
                                </lightning-formatted-date-time>
                            </div>
                        </div>
                    </div>
                    <div class="slds-col slds-size_1-of-3">
```

```
                            <div class="slds-form-element">
                                <label class="slds-form-element__label">Executions Count</label>
                                <div class="slds-form-element__control">
                                    <span class="slds-text-heading_small">
                                        {scheduleInfo.numberOfExecutions}
                                    </span>
                                </div>
                            </div>
                        </div>
                    </div>
                </template>

                <div class="slds-m-top_medium">
                    <lightning-button
                        variant="brand"
                        label="Update Schedule"
                        onclick={handleUpdateSchedule}
                        disabled={isLoading}>
                    </lightning-button>
                </div>
            </div>
        </div>

        <!-- Loading Spinner -->
        <template if:true={isLoading}>
            <lightning-spinner
                alternative-text="Processing..."
                size="medium">
            </lightning-spinner>
        </template>
    </div>
</lightning-card>
</template>
```

Ejemplo 2: Uso desde Apex para Administración Programática

```
public class EventLogAdministration {

    // Método para refrescar análisis para el último mes
    public static void refreshLastMonthAnalytics() {
        Datetime lastMonth = Datetime.now().addDays(-30);
        Datetime now = Datetime.now();

        Boolean success = FoundationEventLogController.refreshAnalytics(lastMonth, now);

        if (success) {
            System.debug('Analytics refresh for last month initiated successfully');
        } else {
            System.debug('Failed to initiate analytics refresh');
        }
    }

    // Método para programar limpieza diaria
    public static void scheduleDailyCleanup() {
        String dailyCronExp = '0 0 2 * * ?'; // Todos los días a las 2 AM

        try {
```

```
            FoundationEventLogController.refreshScheduleInfo(dailyCronExp, new List<String>());
            System.debug('Daily cleanup scheduled successfully');
        } catch (Exception e) {
            System.debug('Error scheduling daily cleanup: ' + e.getMessage());
        }
    }

    // Método para obtener información del cronograma
    public static void displayScheduleInfo() {
        try {
            FoundationEventLogController.eventInfoWrapper info =
                FoundationEventLogController.loadScheduleInfo();

            System.debug('Schedule Status: ' + (info.isActive ? 'Active' : 'Inactive'));
            System.debug('Cron Expression: ' + info.cronExp);
            System.debug('Next Fire Time: ' + info.nextFireTime);
            System.debug('Previous Fire Time: ' + info.previousFireTime);
            System.debug('Number of Executions: ' + info.numberOfExecutions);
        } catch (Exception e) {
            System.debug('Error loading schedule info: ' + e.getMessage());
        }
    }

    // Método para limpiar análisis
    public static void performAnalyticsCleanup() {
        Boolean success = FoundationEventLogController.cleanAnalytics();

        if (success) {
            System.debug('Analytics cleaned successfully');
        } else {
            System.debug('Failed to clean analytics');
        }
    }
}

// Uso de los métodos
EventLogAdministration.refreshLastMonthAnalytics();
EventLogAdministration.scheduleDailyCleanup();
EventLogAdministration.displayScheduleInfo();
EventLogAdministration.performAnalyticsCleanup();
```

Ejemplo 3: Configuración de Expresiones Cron Comunes

```
public class CronExpressionHelper {

    // Expresiones cron comunes para la limpieza de eventos
    public static final Map<String, String> COMMON_SCHEDULES = new Map<String, String>{
        'DAILY_2AM' => '0 0 2 * * ?',           // Todos los días a las 2 AM
        'WEEKLY_SUNDAY' => '0 0 2 ? * SUN',     // Domingos a las 2 AM
        'MONTHLY_FIRST' => '0 0 2 1 * ?',       // Primer día del mes a las 2 AM
        'WEEKDAYS_ONLY' => '0 0 2 ? * MON-FRI', // Días laborables a las 2 AM
        'TWICE_DAILY' => '0 0 2,14 * * ?'       // Dos veces al día (2 AM y 2 PM)
    };

    public static void setupWeeklyCleanup() {
        FoundationEventLogController.refreshScheduleInfo(
            COMMON_SCHEDULES.get('WEEKLY_SUNDAY'),
```

```
            new List<String>()
        );
    }

    public static void setupDailyCleanup() {
        FoundationEventLogController.refreshScheduleInfo(
            COMMON_SCHEDULES.get('DAILY_2AM'),
            new List<String>()
        );
    }

    public static void setupMonthlyCleanup() {
        FoundationEventLogController.refreshScheduleInfo(
            COMMON_SCHEDULES.get('MONTHLY_FIRST'),
            new List<String>()
        );
    }
}
```

Mejores Prácticas Implementadas

1. **Separación de Responsabilidades**: El controlador actúa como una capa de interfaz entre el frontend y la lógica de negocio.
2. **Manejo de Contexto de Pruebas**: Incluye lógica específica para contextos de prueba.
3. **Validación de Datos**: Valida parámetros de entrada y maneja casos nulos.
4. **Manejo de Excepciones**: Incluye bloques try-catch para manejar errores.
5. **Uso de Wrapper Classes**: Utiliza clases wrapper para estructurar datos complejos para el frontend.

Consideraciones y Mejoras Potenciales

Observaciones sobre el Código

1. **Método Incompleto**: El método `enabledScheduleInfo` parece estar incompleto y no actualiza realmente el estado del trabajo programado.
2. **Parámetro No Utilizado**: El parámetro `timeParts` en `refreshScheduleInfo` no se utiliza en la implementación.
3. **Inconsistencia en Nombres de Trabajos**: Los nombres de trabajos en las pruebas están intercambiados.
4. **Lógica de Cancelación**: La lógica para cancelar trabajos programados podría mejorarse.

Mejoras Sugeridas

```
// Versión mejorada del método enabledScheduleInfo
@AuraEnabled
public static boolean enabledScheduleInfo(Boolean enabled) {
    try {
        Suratech_Foundation_Schedule_Event__mdt config;
        if (Test.isRunningTest()) {
            config = new Suratech_Foundation_Schedule_Event__mdt(
                MasterLabel = 'DefaultConfig',
                DeveloperName = 'DefaultConfig',
                Frequency__c = '0 0 2 * * ?',
                Is_Active__c = enabled,
                Schedule_Job_Id__c = ''
```

```
            );
        } else {
            config = [
                SELECT Id, Frequency__c, Schedule_Job_Id__c, MasterLabel, DeveloperName, Is_Active__c
                FROM Suratech_Foundation_Schedule_Event__mdt
                WHERE DeveloperName = 'DefaultConfig'
                LIMIT 1
            ];
        }

        // Actualizar el estado
        config.Is_Active__c = enabled;

        // Si se está deshabilitando y hay un trabajo programado, cancelarlo
        if (!enabled && String.isNotBlank(config.Schedule_Job_Id__c)) {
            try {
                System.abortJob(config.Schedule_Job_Id__c);
                config.Schedule_Job_Id__c = null;
            } catch (Exception e) {
                System.debug('Error aborting job: ' + e.getMessage());
            }
        }

        if (!Test.isRunningTest()) {
            MetadataUtils.upsertMetadata(config);
        }

        return true;
    } catch (Exception e) {
        System.debug('Error in enabledScheduleInfo: ' + e.getMessage());
        return false;
    }
}
```

Notas Adicionales

1. **Interfaz de Usuario**: Esta clase está diseñada específicamente para proporcionar una interfaz entre componentes Lightning y la funcionalidad de gestión de eventos.
2. **Configuración Centralizada**: Utiliza metadatos personalizados para mantener la configuración centralizada y modificable sin cambios de código.
3. **Soporte para Testing**: Incluye lógica robusta para manejar contextos de prueba.
4. **Flexibilidad de Programación**: Permite programación flexible de trabajos usando expresiones cron.
5. **Monitoreo de Trabajos**: Proporciona información detallada sobre el estado y historial de ejecución de trabajos programados.

Core

Descripción General

`Core` es una clase global utilitaria fundamental que proporciona funcionalidades básicas de debugging y monitoreo de límites del sistema Salesforce. Esta clase actúa como una biblioteca central de utilidades que es ampliamente utilizada en todo el framework Foundation de Suratech para registro de depuración y seguimiento del rendimiento del sistema.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

4 de febrero de 2025 por Jean Carlos Melendez

Estructura de la Clase

```
global without sharing class Core {
  global static void debug(String message);
  global static String getCalloutStatistics();
  global static String getCPUTimeStatistics();
  global static String getSOQLStatistics();
  global static String getSOSLStatistics();
  global static void getSymmaryLimits();
}
```

Características Clave

- **Modificador `without sharing`**: La clase no hereda reglas de compartición, permitiendo acceso completo a datos del sistema para funciones de monitoreo.
- **Métodos estáticos globales**: Todos los métodos son estáticos y globales, permitiendo su uso desde cualquier contexto sin necesidad de instanciación.

Métodos

debug(String message)

Descripción

Método estático global que proporciona funcionalidad de debugging estandarizada para todo el framework. Registra mensajes con un nivel de logging específico y un prefijo identificador.

Parámetros

- `message` (String): Mensaje a registrar en los logs de depuración.

Retorno

- `void`: Este método no devuelve ningún valor.

Características

- **Nivel de Logging**: Utiliza `LoggingLevel.FINEST` para registros de depuración detallados.
- **Prefijo Estándar**: Añade el prefijo "Core: " a todos los mensajes para facilitar la identificación.

- **Uso Universal**: Es utilizado por múltiples clases del framework como método central de debugging.

Código

```
global static void debug(String message) {
  System.debug(LoggingLevel.FINEST, 'Core: ' + message);
}
```

getCalloutStatistics()

Descripción

Método que retorna estadísticas de uso de callouts HTTP, mostrando cuántos callouts se han realizado versus el límite permitido.

Parámetros

Ninguno.

Retorno

- `String`: Cadena formateada que muestra "Callouts statistics=X/Y" donde X es el número actual de callouts y Y es el límite máximo.

Uso

Útil para monitorear el consumo de callouts y prevenir que se alcancen los límites del sistema.

Código

```
global static String getCalloutStatistics() {
  return 'Callouts statistics=' +
    System.Limits.getCallouts() +
    '/' +
    System.Limits.getLimitCallouts();
}
```

getCPUTimeStatistics()

Descripción

Método que retorna estadísticas de uso de tiempo de CPU, mostrando el tiempo consumido versus el límite permitido.

Parámetros

Ninguno.

Retorno

- `String`: Cadena formateada que muestra "CPU time usage=X/Y" donde X es el tiempo de CPU utilizado en milisegundos y Y es el límite máximo.

Uso

Crítico para monitorear el rendimiento y prevenir timeouts por exceso de procesamiento.

Código

```
global static String getCPUTimeStatistics() {
  return 'CPU time usage=' +
    System.Limits.getCpuTime() +
    '/' +
    System.Limits.getLimitCpuTime();
}
```

getSOQLStatistics()

Descripción

Método que retorna estadísticas de uso de consultas SOQL, mostrando cuántas consultas se han ejecutado versus el límite permitido.

Parámetros

Ninguno.

Retorno

- `String`: Cadena formateada que muestra "SOQL queries usage=X/Y" donde X es el número de consultas SOQL ejecutadas y Y es el límite máximo.

Uso

Esencial para prevenir excepciones por exceso de consultas SOQL en una transacción.

Código

```
global static String getSOQLStatistics() {
  return 'SOQL queries usage=' +
    System.Limits.getQueries() +
    '/' +
    System.Limits.getLimitQueries();
}
```

getSOSLStatistics()

Descripción

Método que retorna estadísticas de uso de consultas SOSL, mostrando cuántas búsquedas se han ejecutado versus el límite permitido.

Parámetros

Ninguno.

Retorno

- `String`: Cadena formateada que muestra "SOSL queries usage=X/Y" donde X es el número de consultas SOSL ejecutadas y Y es el límite máximo.

Uso

Útil para monitorear el uso de búsquedas SOSL y evitar alcanzar los límites del sistema.

Código

```
global static String getSOSLStatistics() {
  return 'SOSL queries usage=' +
    System.Limits.getSoslQueries() +
    '/' +
    System.Limits.getLimitSoslQueries();
}
```

getSymmaryLimits()

Descripción

Método que registra un resumen completo de todas las estadísticas de límites del sistema de forma formateada.

Nota: Existe un error tipográfico en el nombre del método (debería ser `getSummaryLimits`).

Parámetros

Ninguno.

Retorno

- `void`: Este método no devuelve ningún valor, sino que registra la información.

Proceso

1. Obtiene estadísticas de CPU, SOQL, SOSL y Callouts.
2. Formatea toda la información en una sola cadena.
3. Registra el resumen usando el método `debug`.

Código

```
global static void getSymmaryLimits() {
  Core.debug(
    '===== Statistics(' +
      Core.getCPUTimeStatistics() +
      ',' +
      Core.getSOQLStatistics() +
      ',' +
      Core.getSOSLStatistics() +
      ',' +
      Core.getCalloutStatistics() +
      ') ====='
  );
}
```

Límites del Sistema Salesforce Monitoreados

Tipo de Límite	Método de Acceso	Límite Típico	Descripción
Callouts HTTP	`getCallouts()`	100 por transacción	Llamadas a servicios externos
Tiempo de CPU	`getCpuTime()`	10,000ms (síncr.) / 60,000ms (asíncr.)	Tiempo de procesamiento
Consultas SOQL	`getQueries()`	100 por transacción	Consultas a la base de datos
Consultas SOSL	`getSoslQueries()`	20 por transacción	Búsquedas de texto

Ejemplos de Uso

Ejemplo 1: Debugging Básico

```
public class ExampleService {

    public static void processData(List<Account> accounts) {
        Core.debug('Starting data processing for ' + accounts.size() + ' accounts');

        try {
            for (Account acc : accounts) {
                // Procesar cada cuenta
                processAccount(acc);
                Core.debug('Processed account: ' + acc.Name);
            }

            Core.debug('Data processing completed successfully');
        } catch (Exception e) {
            Core.debug('Error in data processing: ' + e.getMessage());
```

```
            throw e;
        }
    }

    private static void processAccount(Account acc) {
        // Lógica de procesamiento
        Core.debug('Processing account with ID: ' + acc.Id);
    }
}

// Uso
List<Account> accounts = [SELECT Id, Name FROM Account LIMIT 10];
ExampleService.processData(accounts);
```

Ejemplo 2: Monitoreo de Límites en Procesamiento por Lotes

```
public class BatchProcessorWithMonitoring implements Database.Batchable<SObject> {

    public Database.QueryLocator start(Database.BatchableContext bc) {
        Core.debug('Starting batch processing');
        Core.getSymmaryLimits(); // Registrar límites iniciales

        return Database.getQueryLocator('SELECT Id, Name FROM Account WHERE IsActive__c = true');
    }

    public void execute(Database.BatchableContext bc, List<Account> scope) {
        Core.debug('Processing batch of ' + scope.size() + ' records');

        // Registrar límites antes del procesamiento
        Core.debug('Limits before processing:');
        Core.debug(Core.getCPUTimeStatistics());
        Core.debug(Core.getSOQLStatistics());

        try {
            // Procesar registros
            for (Account acc : scope) {
                processAccount(acc);

                // Verificar límites periódicamente
                if (System.Limits.getCpuTime() > 8000) { // 80% del límite
                    Core.debug('CPU time approaching limit: ' + Core.getCPUTimeStatistics());
                }
            }

            // Actualizar registros
            update scope;

        } catch (Exception e) {
            Core.debug('Error in batch execution: ' + e.getMessage());

            // Registrar límites en caso de error
            Core.getSymmaryLimits();
            throw e;
        }

        // Registrar límites después del procesamiento
        Core.debug('Limits after processing:');
```

```
        Core.getSymmaryLimits();
    }

    public void finish(Database.BatchableContext bc) {
        Core.debug('Batch processing completed');
        Core.getSymmaryLimits(); // Registrar límites finales
    }

    private void processAccount(Account acc) {
        // Simular procesamiento
        acc.Description = 'Processed at ' + System.now();
    }
}

// Ejecutar el batch
Database.executeBatch(new BatchProcessorWithMonitoring(), 200);
```

Ejemplo 3: Monitoreo en Integraciones

```
public class IntegrationServiceWithMonitoring {

    public static Map<String, Object> callExternalService(String endpoint, Map<String, Object> payload) {
        Core.debug('Starting external service call to: ' + endpoint);

        // Verificar límites antes de la llamada
        Core.debug('Pre-callout limits: ' + Core.getCalloutStatistics());

        try {
            // Configurar la solicitud HTTP
            HttpRequest request = new HttpRequest();
            request.setEndpoint(endpoint);
            request.setMethod('POST');
            request.setBody(JSON.serialize(payload));
            request.setHeader('Content-Type', 'application/json');

            // Realizar la llamada
            Http http = new Http();
            HttpResponse response = http.send(request);

            Core.debug('External service response received. Status: ' + response.getStatusCode());

            // Verificar límites después de la llamada
            Core.debug('Post-callout limits: ' + Core.getCalloutStatistics());

            // Procesar respuesta
            Map<String, Object> result = new Map<String, Object>{
                'statusCode' => response.getStatusCode(),
                'body' => response.getBody(),
                'success' => response.getStatusCode() == 200
            };

            return result;

        } catch (Exception e) {
            Core.debug('Error in external service call: ' + e.getMessage());

            // Registrar todos los límites en caso de error
```

```
            Core.getSymmaryLimits();

            throw new IntegrationException('Failed to call external service: ' + e.getMessage(), e);
        }
    }

    public class IntegrationException extends Exception {}
}

// Uso
Map<String, Object> payload = new Map<String, Object>{
    'action' => 'create',
    'data' => new Map<String, Object>{'name' => 'Test', 'value' => 123}
};

Map<String, Object> result = IntegrationServiceWithMonitoring.callExternalService(
    'https://api.example.com/data',
    payload
);
```

Ejemplo 4: Utility Class para Monitoreo Avanzado

```
public class PerformanceMonitor {

    private static Map<String, Long> startTimes = new Map<String, Long>();

    // Iniciar monitoreo de una operación
    public static void startOperation(String operationName) {
        startTimes.put(operationName, System.currentTimeMillis());
        Core.debug('Started operation: ' + operationName);
        Core.getSymmaryLimits();
    }

    // Finalizar monitoreo de una operación
    public static void endOperation(String operationName) {
        Long startTime = startTimes.get(operationName);
        if (startTime != null) {
            Long duration = System.currentTimeMillis() - startTime;
            Core.debug('Completed operation: ' + operationName + ' in ' + duration + 'ms');
            startTimes.remove(operationName);
        }
        Core.getSymmaryLimits();
    }

    // Verificar si estamos cerca de los límites
    public static Boolean isApproachingLimits() {
        Boolean approaching = false;

        // Verificar CPU (80% del límite)
        if (System.Limits.getCpuTime() > (System.Limits.getLimitCpuTime() * 0.8)) {
            Core.debug('WARNING: Approaching CPU time limit');
            approaching = true;
        }

        // Verificar SOQL (80% del límite)
        if (System.Limits.getQueries() > (System.Limits.getLimitQueries() * 0.8)) {
            Core.debug('WARNING: Approaching SOQL queries limit');
```

```
            approaching = true;
        }

        // Verificar Callouts (80% del límite)
        if (System.Limits.getCallouts() > (System.Limits.getLimitCallouts() * 0.8)) {
            Core.debug('WARNING: Approaching callouts limit');
            approaching = true;
        }

        return approaching;
    }

    // Generar reporte detallado de uso
    public static String generateUsageReport() {
        List<String> report = new List<String>();

        report.add('=== PERFORMANCE USAGE REPORT ===');
        report.add('CPU Time: ' + Core.getCPUTimeStatistics());
        report.add('SOQL Queries: ' + Core.getSOQLStatistics());
        report.add('SOSL Queries: ' + Core.getSOSLStatistics());
        report.add('Callouts: ' + Core.getCalloutStatistics());

        // Calcular porcentajes de uso
        Decimal cpuUsagePercent = (Decimal.valueOf(System.Limits.getCpuTime()) /
                                   System.Limits.getLimitCpuTime() * 100).setScale(2);
        Decimal soqlUsagePercent = (Decimal.valueOf(System.Limits.getQueries()) /
                                    System.Limits.getLimitQueries() * 100).setScale(2);

        report.add('CPU Usage: ' + cpuUsagePercent + '%');
        report.add('SOQL Usage: ' + soqlUsagePercent + '%');
        report.add('================================');

        String fullReport = String.join(report, '\n');
        Core.debug(fullReport);

        return fullReport;
    }
}

// Uso del monitor de rendimiento
PerformanceMonitor.startOperation('DataMigration');

try {
    // Realizar operaciones
    List<Account> accounts = [SELECT Id, Name FROM Account LIMIT 1000];

    for (Account acc : accounts) {
        acc.Description = 'Updated at ' + System.now();

        // Verificar límites periódicamente
        if (PerformanceMonitor.isApproachingLimits()) {
            Core.debug('Pausing operation due to approaching limits');
            break;
        }
    }

    update accounts;
```

```
} finally {
    PerformanceMonitor.endOperation('DataMigration');
    PerformanceMonitor.generateUsageReport();
}
```

Ejemplo 5: Debugging en Triggers

```
public class AccountTriggerHandler {

    public static void handleAfterInsert(List<Account> newAccounts) {
        Core.debug('AccountTriggerHandler.handleAfterInsert started for ' + newAccounts.size() + ' accounts');

        try {
            // Verificar límites antes del procesamiento
            if (System.Limits.getQueries() > 80) { // Cerca del límite de 100
                Core.debug('WARNING: High SOQL usage detected: ' + Core.getSOQLStatistics());
            }

            // Procesar cuentas
            for (Account acc : newAccounts) {
                processNewAccount(acc);
            }

            Core.debug('AccountTriggerHandler.handleAfterInsert completed successfully');

        } catch (Exception e) {
            Core.debug('Error in AccountTriggerHandler.handleAfterInsert: ' + e.getMessage());

            // Registrar límites en caso de error para debugging
            Core.getSymmaryLimits();

            throw e;
        }
    }

    private static void processNewAccount(Account acc) {
        Core.debug('Processing new account: ' + acc.Name + ' (ID: ' + acc.Id + ')');

        // Lógica de procesamiento específica
        // ...
    }
}
```

Uso en el Framework Foundation

La clase `Core` es utilizada extensivamente en todo el framework Foundation de Suratech:

- **EventLogManager**: Para debugging de operaciones de logs
- **GenerateURLServiceHandler**: Para debugging de procesos de integración
- **CCMServiceHandler**: Para debugging de comunicaciones CCM
- **FoundationEventLogController**: Para debugging de operaciones del controlador
- **QuoteUtils**: Para debugging de operaciones de cotización

Mejores Prácticas para su Uso

1. **Debugging Consistente**: Utilizar `Core.debug()` en lugar de `System.debug()` directo para mantener consistencia en el formato de logs.
2. **Monitoreo Proactivo**: Usar los métodos de estadísticas para monitorear límites antes de que se alcancen.
3. **Logging en Puntos Críticos**: Registrar estadísticas al inicio y fin de operaciones complejas.
4. **Manejo de Errores**: Incluir `Core.getSymmaryLimits()` en bloques catch para facilitar el debugging.
5. **Verificación Periódica**: En bucles largos, verificar límites periódicamente para evitar excepciones.

Consideraciones de Rendimiento

1. **Overhead Mínimo**: Los métodos de la clase `Core` tienen overhead mínimo y pueden usarse liberalmente.
2. **Frecuencia de Llamadas**: Los métodos de estadísticas pueden llamarse frecuentemente sin impacto significativo.
3. **Nivel de Logging**: `LoggingLevel.FINEST` puede ser filtrado en producción según la configuración de logging.

Posibles Mejoras

Corrección del Error Tipográfico

```
// Método corregido
global static void getSummaryLimits() {
  Core.debug(
    '===== Statistics(' +
      Core.getCPUTimeStatistics() +
      ',' +
      Core.getSOQLStatistics() +
      ',' +
      Core.getSOSLStatistics() +
      ',' +
      Core.getCalloutStatistics() +
      ') ====='
  );
}
```

Extensión con Métodos Adicionales

```
// Métodos adicionales que podrían añadirse
global static String getDMLStatistics() {
  return 'DML statements usage=' +
    System.Limits.getDMLStatements() +
    '/' +
    System.Limits.getLimitDMLStatements();
}

global static String getHeapSizeStatistics() {
  return 'Heap size usage=' +
    System.Limits.getHeapSize() +
    '/' +
    System.Limits.getLimitHeapSize();
}
```

```
global static Boolean isApproachingAnyLimit(Integer thresholdPercent) {
  // Lógica para verificar si algún límite está cerca del threshold
  return false; // Implementación simplificada
}
```

Notas Adicionales

1. **Clase Fundamental**: Esta es una clase fundamental del framework que es utilizada por prácticamente todas las demás clases del sistema.
2. **Logging Centralizado**: Proporciona un punto central para el logging, facilitando cambios futuros en la estrategia de logging.
3. **Monitoreo de Rendimiento**: Esencial para el monitoreo de rendimiento y la prevención de errores por límites del sistema.
4. **Simplicidad Intencional**: La simplicidad de la clase es intencional, proporcionando funcionalidad básica pero esencial de forma confiable.
5. **Sin Dependencias**: No depende de otras clases del framework, asegurando que siempre esté disponible para uso.

BaseProcessManager

Descripción General

Clase global que actúa como el gestor principal de procesos en el sistema Foundation. Implementa el patrón Chain of Responsibility para ejecutar secuencias de `ProductHandler` organizados por etapas (stages), con soporte opcional para cache y manejo dinámico de configuraciones.

Información de la Clase

- **Autor:** Jean Carlos Melendez
- **Última modificación:** 04-01-2025
- **Modificado por:** fabian.lopez@nespon.com
- **Tipo:** Clase global con sharing
- **Propósito:** Gestor centralizado de procesos y cadenas de responsabilidad

Método Principal

invoke (Método Estático Global)

```
global static Boolean invoke(
    String stage,
    Map<String, Object> options,
    Map<String, Object> input,
    Map<String, Object> output
)
```

Descripción: Método principal que ejecuta una cadena de procesos para una etapa específica, construyendo dinámicamente la cadena de responsabilidad y ejecutándola.

Parámetros:

- `stage` (String): Etapa del proceso a ejecutar, debe existir en `SF_MainProcess__mdt.Stage__c`
- `options` (Map<String, Object>): Opciones de configuración para el proceso
- `input` (Map<String, Object>): Datos de entrada para el proceso
- `output` (Map<String, Object>): Mapa donde se almacenarán los resultados del proceso

Retorna:

- `Boolean`: Siempre retorna `true` si la ejecución es exitosa

Excepciones:

- `ProcessManagerNotFoundException`: Se lanza cuando el stage no existe en los metadatos

Flujo de Ejecución Detallado

1. Validación de Stage

```
if (!CorePicklistUtils.containsVal(SF_MainProcess__mdt.Stage__c, stage))
```

- Valida que el stage exista en el Custom Metadata Type `SF_MainProcess__mdt`
- Utiliza `CorePicklistUtils` para verificar valores válidos
- Lanza excepción si el stage no es válido

2. Evaluación de Cache

```
FeatureFlag__mdt flag = [
    SELECT Is_Active__c
    FROM FeatureFlag__mdt
    WHERE DeveloperName = 'sura_foundation_cache'
    LIMIT 1
];
cacheEnabled = flag.Is_Active__c;
```

Comportamiento:

- Consulta el feature flag `sura_foundation_cache`
- Si existe y está activo, habilita el cache
- En caso de excepción (flag no existe), deshabilita el cache
- Establece `ignoreCache = !cacheEnabled`

3. Construcción de la Cadena de Responsabilidad

```
ReverseIterator reverseIterator = new ReverseIterator(
    ((List<ProductHandler>) ProcessFactory.newInstances(
        ProcessFactory.getAllDefinitionsByStage(stage, ignoreCache)
    ))
);
```

Proceso:

1. `ProcessFactory.getAllDefinitionsByStage(stage, ignoreCache)` obtiene definiciones
2. `ProcessFactory.newInstances()` crea instancias de `ProductHandler`
3. `ReverseIterator` invierte el orden para construcción correcta de la cadena

4. Enlazado de Handlers (Chain Building)

```
ProductHandler previous = (ProductHandler) reverseIterator.next();

while (reverseIterator.hasNext()) {
    ProductHandler current = (ProductHandler) reverseIterator.next();
    current.setNext(previous);
    previous = current;
}
```

Algoritmo:

- Itera en reversa para construir la cadena correctamente
- Cada handler actual apunta al anterior como `next`
- El resultado es una cadena donde el primer handler procesará toda la secuencia

5. Ejecución de la Cadena

```
((ProductHandler) reverseIterator.first()).apply(options, input, output);
```

- Ejecuta el primer handler de la cadena
- Cada handler decide si procesar y/o pasar al siguiente
- Los datos fluyen a través de `options`, `input` y `output`

6. Finalización y Logging

```
System.debug(output);
System.debug('finalizar proceso baseprocess');
Core.getSymmaryLimits();
```

- Registra el output final en debug
- Ejecuta `Core.getSymmaryLimits()` para monitoreo de límites
- Retorna `true` indicando ejecución exitosa

Patrones de Diseño Implementados

Chain of Responsibility

- **Propósito:** Permite que múltiples objetos manejen una solicitud sin acoplar el emisor con los receptores
- **Implementación:** Cada `ProductHandler` puede procesar la solicitud y/o pasarla al siguiente
- **Ventajas:** Flexibilidad para agregar/remover handlers dinámicamente

Factory Pattern

- **Uso:** `ProcessFactory` para crear instancias de handlers
- **Beneficio:** Centraliza la lógica de creación e instanciación

Iterator Pattern

- **Implementación:** `ReverseIterator` para navegación controlada
- **Propósito:** Facilita la construcción correcta de la cadena

Componentes y Dependencias

Dependencias Principales

- **SF_MainProcess__mdt:** Custom Metadata Type para definición de stages
- **FeatureFlag__mdt:** Para control de cache (`sura_foundation_cache`)
- **ProcessFactory:** Factory para creación de handlers
- **ProductHandler:** Clase base para handlers de la cadena
- **ReverseIterator:** Iterador personalizado para construcción de cadena
- **CorePicklistUtils:** Utilidades para validación de picklist values

- **Core:** Utilidades del sistema (límites, debugging)

Excepciones Personalizadas

- **ProcessManagerNotFoundException:** Excepción específica para stages no encontrados

Configuración del Sistema

Feature Flags Utilizados

- `sura_foundation_cache`: Controla si el cache está habilitado para el sistema

Custom Metadata Types

- `SF_MainProcess__mdt`: Define los stages disponibles y sus configuraciones
- `FeatureFlag__mdt`: Configuración de feature flags del sistema

Casos de Uso

Procesamiento de Órdenes

```
Map<String, Object> input = new Map<String, Object>{'orderId' => '123'};
Map<String, Object> output = new Map<String, Object>();
Map<String, Object> options = new Map<String, Object>{'validateOnly' => false};

BaseProcessManager.invoke('ORDER_PROCESSING', options, input, output);
```

Validación de Datos

```
BaseProcessManager.invoke('DATA_VALIDATION', options, inputData, results);
```

Integración con Sistemas Externos

```
BaseProcessManager.invoke('EXTERNAL_INTEGRATION', config, requestData, responseData);
```

Ventajas del Diseño

Flexibilidad

- Configuración dinámica de handlers por stage
- Soporte para cache configurable
- Fácil adición/remoción de pasos del proceso

Mantenibilidad

- Separación clara de responsabilidades
- Configuración basada en metadata
- Logging integrado para debugging

Performance

- Cache opcional para mejorar rendimiento

- Factory pattern para optimización de instanciación
- Monitoreo de límites integrado

Escalabilidad

- Soporte para múltiples stages
- Handlers reutilizables entre diferentes procesos
- Configuración sin código a través de metadata

Consideraciones de Performance

Cache Management

- Feature flag `sura_foundation_cache` controla el uso de cache
- Cache puede mejorar significativamente la performance en procesos frecuentes
- Considerar el impacto de memory vs. processing time

Límites de Salesforce

- `Core.getSymmaryLimits()` monitorea el uso de límites
- Importante para procesos que consumen muchos recursos
- Permite identificar cuellos de botella

Mejoras Potenciales

- Implementar retry logic para procesos fallidos
- Agregar métricas de performance por handler
- Soporte para procesamiento asíncrono (Queueable/Batch)
- Implementar circuit breaker pattern para handlers problemáticos
- Agregar validación de tipos para input/output maps
- Soporte para handlers condicionales basados en datos de entrada

BaseProcessFoundationVlocityAdapter

Descripción General

Clase abstracta global que actúa como adaptador entre el framework de Vlocity y el sistema de gestión de procesos base (`BaseProcessManager`). Proporciona una interfaz estandarizada para invocar procesos a través de diferentes etapas del flujo de trabajo.

Información de la Clase

- **Autor:** Jean Carlos Melendez
- **Última modificación:** 03-31-2025
- **Modificado por:** Jean Carlos Melendez
- **Tipo:** Clase abstracta global
- **Propósito:** Adaptador para integración Vlocity-BaseProcessManager

Métodos

applyInvoke (Método Estático)

```
global static Boolean applyInvoke(
    Map<String, Object> inputMap,
    Map<String, Object> outputMap,
    Map<String, Object> optionMap
)
```

Descripción: Método estático global que actúa como punto de entrada principal para invocar procesos a través del BaseProcessManager.

Parámetros:

- `inputMap` (Map<String, Object>): Mapa de datos de entrada para el proceso
- `outputMap` (Map<String, Object>): Mapa donde se almacenarán los resultados del proceso
- `optionMap` (Map<String, Object>): Mapa de opciones de configuración para el proceso

Retorna:

- `Boolean`: `true` si el proceso se ejecutó exitosamente, `false` en caso contrario

Comportamiento:

1. Extrae el valor de 'Stage' del inputMap para determinar la etapa del proceso
2. Registra la etapa en el debug log
3. Delega la ejecución al `BaseProcessManager.invoke()` con los parámetros reorganizados
4. Retorna el resultado de la invocación

Flujo de Datos:

```
inputMap['Stage'] → stage → BaseProcessManager.invoke(stage, optionMap, inputMap, outputMap)
```

invokeMethod

```
public Boolean invokeMethod(
    String methodName,
    Map<String, Object> inputMap,
    Map<String, Object> outputMap,
    Map<String, Object> optionMap
)
```

Descripción: Método público que proporciona un mecanismo de invocación dinámica basado en el nombre del método.

Parámetros:

- `methodName` (String): Nombre del método a ejecutar (case-insensitive)
- `inputMap` (Map<String, Object>): Mapa de datos de entrada
- `outputMap` (Map<String, Object>): Mapa de datos de salida
- `optionMap` (Map<String, Object>): Mapa de opciones de configuración

Retorna:

- `Boolean`: Resultado de la ejecución del método solicitado

Comportamiento:

- Utiliza una estructura `switch on` para determinar qué método ejecutar
- Convierte el nombre del método a mayúsculas para comparación case-insensitive
- Actualmente soporta únicamente 'APPLYINVOKE'
- Registra métodos no soportados en el debug log y retorna `false`

Métodos Soportados:

- `'APPLYINVOKE'`: Invoca el método `applyInvoke()`

Patrón de Diseño

Adapter Pattern

Esta clase implementa el **patrón Adapter**, actuando como:

- **Adaptador:** Entre Vlocity framework y BaseProcessManager
- **Interfaz unificada:** Para diferentes tipos de invocaciones de procesos
- **Punto de entrada:** Centralizado para operaciones de proceso

Command Pattern

El método `invokeMethod` implementa aspectos del **patrón Command**:

- Encapsula la invocación de métodos como objetos
- Permite invocar métodos dinámicamente por nombre
- Facilita la extensión con nuevos comandos

Integración con Vlocity

Propósito en el Ecosistema

- **Bridge:** Conecta componentes Vlocity con lógica de negocio personalizada
- **Abstracción:** Oculta la complejidad del BaseProcessManager de Vlocity
- **Estandarización:** Proporciona interfaz consistente para invocaciones

Flujo de Integración

```
Vlocity Component → BaseProcessFoundationVlocityAdapter → BaseProcessManager → Business Logic
```

Consideraciones Técnicas

Visibilidad Global

- Declarada como `global` para permitir acceso desde paquetes Vlocity
- Métodos estáticos globales para invocación directa sin instanciación

Manejo de Parámetros

- Utiliza casting explícito para asegurar tipos correctos
- Usa operador de navegación segura (`?.`) para prevenir null pointer exceptions

Extensibilidad

- Estructura switch permite agregar nuevos métodos fácilmente
- Clase abstracta permite herencia para especializaciones

Dependencias

- **BaseProcessManager:** Clase principal que maneja la lógica de procesos
- **Vlocity Framework:** Para integración con componentes Vlocity
- **System.debug:** Para logging y diagnóstico

Casos de Uso

1. **Procesos Vlocity:** Ejecutar lógica de negocio desde flows o components Vlocity
2. **Integración de Etapas:** Manejar diferentes etapas de procesos complejos
3. **Invocación Dinámica:** Ejecutar métodos específicos basado en configuración
4. **Punto de Entrada Unificado:** Centralizar invocaciones desde múltiples fuentes

Mejoras Potenciales

- Implementar manejo de excepciones más robusto

- Agregar validación de parámetros de entrada
- Incluir logging más detallado para auditoría
- Expandir métodos soportados en `invokeMethod`
- Implementar caché para mejorar performance en invocaciones frecuentes

BaseObserver

Descripción General

Clase base virtual que implementa la interfaz `IObserver` para el patrón Observer. Proporciona implementaciones base para la observación de eventos y publicación de eventos de plataforma.

Información de la Clase

- **Autor:** Soulberto Lorenzo soulberto@cloudblue.us (mailto:soulberto@cloudblue.us)
- **Última modificación:** 10-11-2024
- **Modificado por:** Jean Carlos Melendez
- **Tipo:** Clase virtual pública
- **Implementa:** IObserver

Métodos

updateObserver

```
public virtual void updateObserver(String flag, Object data)
```

Descripción: Método virtual para actualizar el observador cuando se produce un evento.

Parámetros:

- `flag` (String): Bandera que identifica el tipo de evento o acción
- `data` (Object): Datos asociados al evento

Comportamiento:

- Registra un mensaje de debug con el valor de la bandera
- Al ser virtual, puede ser sobrescrito en clases herederas

Autor: Soulberto Lorenzo soulberto@cloudblue.us (mailto:soulberto@cloudblue.us) | 08-16-2024

publishPlatFormEvent

```
public virtual void publishPlatFormEvent(String flag, String productName)
```

Descripción: Método virtual para publicar eventos de plataforma relacionados con productos.

Parámetros:

- `flag` (String): Bandera que identifica el tipo de evento
- `productName` (String): Nombre del producto asociado al evento

Comportamiento:

- Registra un mensaje de debug indicando la publicación del evento para el producto especificado
- Al ser virtual, puede ser sobrescrito en clases herederas para implementar lógica específica

Autor: Soulberto Lorenzo soulberto@cloudblue.us (mailto:soulberto@cloudblue.us) | 08-22-2024

Patrón de Diseño

Esta clase implementa el **patrón Observer**, donde:

- Actúa como observador base que puede recibir notificaciones
- Proporciona métodos virtuales que permiten a las clases herederas personalizar el comportamiento
- Facilita la publicación de eventos de plataforma

Uso Recomendado

1. Extender esta clase para crear observadores específicos
2. Sobrescribir los métodos virtuales según las necesidades del negocio
3. Utilizar para implementar comunicación desacoplada entre componentes

Consideraciones

- Los métodos son virtuales, por lo que deben ser sobrescritos para funcionalidad completa
- La implementación actual solo incluye logging para propósitos de debug
- Requiere la implementación completa de la interfaz `IObserver`

Adapter

Descripción General

`Adapter` es una interfaz pública que define el contrato fundamental para el patrón de diseño Adaptador en el framework Foundation de Suratech. Esta interfaz establece un método estándar que permite adaptar diferentes tipos de servicios, APIs o sistemas externos a una estructura común de datos, facilitando la integración y el intercambio de información entre componentes heterogéneos.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

2 de septiembre de 2024

Definición de la Interfaz

```
@namespaceAccessible
public interface Adapter {
  Map<String, Object> adaptee();
}
```

Decoradores y Accesibilidad

@namespaceAccessible

La interfaz está marcada con `@namespaceAccessible`, lo que indica que está diseñada para ser accesible desde diferentes espacios de nombres (namespaces). Esto es especialmente relevante en contextos de paquetes gestionados donde se necesita que la interfaz sea accesible desde fuera del namespace del paquete.

Métodos

adaptee()

Descripción

Método que debe ser implementado por todas las clases que implementen esta interfaz. Su propósito es adaptar la funcionalidad específica de un servicio, API o sistema a un formato estándar de mapa de datos.

Parámetros

Ninguno.

Retorno

- `Map<String, Object>`: Mapa que contiene los datos adaptados en un formato estándar que puede ser procesado de manera uniforme por otros componentes del sistema.

Propósito y Aplicaciones

El propósito principal de esta interfaz es implementar el patrón de diseño Adaptador, que permite:

1. **Integración de Sistemas Heterogéneos**: Adaptar diferentes APIs, servicios o sistemas a una interfaz común.
2. **Desacoplamiento**: Separar la lógica específica de integración de la lógica de negocio principal.
3. **Flexibilidad**: Permitir el intercambio de diferentes implementaciones sin afectar el código cliente.
4. **Estandarización**: Proporcionar un formato común de datos para sistemas diversos.

5. **Extensibilidad**: Facilitar la adición de nuevos tipos de adaptadores sin modificar código existente.

Esta interfaz es especialmente útil en escenarios como:

- **Integraciones con APIs Externas**: Adaptar respuestas de diferentes APIs a un formato común.
- **Migración de Sistemas**: Facilitar la transición entre diferentes sistemas o versiones.
- **Compatibilidad con Múltiples Proveedores**: Permitir trabajar con diferentes proveedores de servicios usando la misma interfaz.
- **Transformación de Datos**: Convertir datos de un formato a otro de manera consistente.

Patrón de Diseño: Adapter

Estructura del Patrón

```
Client -> Target Interface (Adapter) -> Adaptee (Specific Implementation)
```

En este caso:

- **Client**: Código que utiliza el adaptador
- **Target Interface**: La interfaz `Adapter`
- **Adaptee**: Las implementaciones específicas que adaptan diferentes servicios

Beneficios del Patrón

1. **Reutilización de Código**: Permite reutilizar clases existentes con interfaces incompatibles.
2. **Separación de Responsabilidades**: Separa la lógica de adaptación de la lógica de negocio.
3. **Flexibilidad**: Facilita el cambio de implementaciones sin afectar el código cliente.

Implementaciones Existentes

InsServiceAdapter

Una implementación conocida de esta interfaz que se utiliza para adaptar servicios de Vlocity Insurance:

```
@namespaceAccessible
public class InsServiceAdapter implements Adapter {
  @namespaceAccessible
  public Map<String, Object> adaptee() {
    // return new URLGenerationNotificationHandler();
    return new Map<String, Object>();
  }
}
```

Ejemplos de Implementación

Ejemplo 1: Adaptador para Servicios de Pago

```
public class PaymentServiceAdapter implements Adapter {

    private String paymentProvider;
    private Map<String, Object> configuration;

    public PaymentServiceAdapter(String paymentProvider, Map<String, Object> configuration) {
```

```
        this.paymentProvider = paymentProvider;
        this.configuration = configuration;
    }

    public Map<String, Object> adaptee() {
        try {
            switch on paymentProvider.toLowerCase() {
                when 'stripe' {
                    return adaptStripeService();
                }
                when 'paypal' {
                    return adaptPayPalService();
                }
                when 'square' {
                    return adaptSquareService();
                }
                when else {
                    throw new AdapterException('Unsupported payment provider: ' + paymentProvider);
                }
            }
        } catch (Exception e) {
            return new Map<String, Object>{
                'success' => false,
                'error' => e.getMessage(),
                'provider' => paymentProvider
            };
        }
    }

    private Map<String, Object> adaptStripeService() {
        // Lógica específica para adaptar Stripe
        return new Map<String, Object>{
            'provider' => 'stripe',
            'apiVersion' => '2023-10-16',
            'supportedMethods' => new List<String>{'card', 'bank_transfer', 'apple_pay'},
            'configuration' => new Map<String, Object>{
                'publicKey' => configuration.get('stripe_public_key'),
                'webhookEndpoint' => configuration.get('stripe_webhook_url')
            },
            'capabilities' => new Map<String, Object>{
                'subscriptions' => true,
                'refunds' => true,
                'disputes' => true,
                'payouts' => true
            }
        };
    }

    private Map<String, Object> adaptPayPalService() {
        // Lógica específica para adaptar PayPal
        return new Map<String, Object>{
            'provider' => 'paypal',
            'apiVersion' => 'v2',
            'supportedMethods' => new List<String>{'paypal', 'card', 'venmo'},
            'configuration' => new Map<String, Object>{
                'clientId' => configuration.get('paypal_client_id'),
                'webhookId' => configuration.get('paypal_webhook_id')
```

```
            },
            'capabilities' => new Map<String, Object>{
                'subscriptions' => true,
                'refunds' => true,
                'disputes' => true,
                'payouts' => false
            }
        };
    }

    private Map<String, Object> adaptSquareService() {
        // Lógica específica para adaptar Square
        return new Map<String, Object>{
            'provider' => 'square',
            'apiVersion' => '2023-12-13',
            'supportedMethods' => new List<String>{'card', 'cash_app', 'afterpay'},
            'configuration' => new Map<String, Object>{
                'applicationId' => configuration.get('square_application_id'),
                'locationId' => configuration.get('square_location_id')
            },
            'capabilities' => new Map<String, Object>{
                'subscriptions' => false,
                'refunds' => true,
                'disputes' => true,
                'payouts' => true
            }
        };
    }

    public class AdapterException extends Exception {}
}
```

Ejemplo 2: Adaptador para APIs de Notificación

```
public class NotificationServiceAdapter implements Adapter {

    private String notificationService;
    private Map<String, Object> serviceConfig;

    public NotificationServiceAdapter(String notificationService, Map<String, Object> serviceConfig) {
        this.notificationService = notificationService;
        this.serviceConfig = serviceConfig;
    }

    public Map<String, Object> adaptee() {
        Map<String, Object> adapterInfo = new Map<String, Object>{
            'service' => notificationService,
            'timestamp' => System.now(),
            'isAvailable' => checkServiceAvailability()
        };

        switch on notificationService.toLowerCase() {
            when 'sendgrid' {
                adapterInfo.putAll(adaptSendGridService());
            }
            when 'twilio' {
                adapterInfo.putAll(adaptTwilioService());
```

```
            }
            when 'slack' {
                adapterInfo.putAll(adaptSlackService());
            }
            when 'teams' {
                adapterInfo.putAll(adaptTeamsService());
            }
            when else {
                adapterInfo.put('error', 'Unsupported notification service: ' + notificationService);
                adapterInfo.put('isAvailable', false);
            }
        }

        return adapterInfo;
    }

    private Map<String, Object> adaptSendGridService() {
        return new Map<String, Object>{
            'type' => 'email',
            'endpoint' => 'https://api.sendgrid.com/v3/mail/send',
            'authentication' => 'api_key',
            'maxBatchSize' => 1000,
            'supportedFeatures' => new List<String>{
                'html_content', 'attachments', 'templates', 'scheduling'
            },
            'rateLimits' => new Map<String, Object>{
                'requests_per_second' => 600,
                'emails_per_month' => getEmailQuota()
            }
        };
    }

    private Map<String, Object> adaptTwilioService() {
        return new Map<String, Object>{
            'type' => 'sms',
            'endpoint' => 'https://api.twilio.com/2010-04-01/Accounts/' + serviceConfig.get('account_sid') +
'/Messages.json',
            'authentication' => 'basic_auth',
            'maxBatchSize' => 1,
            'supportedFeatures' => new List<String>{
                'sms', 'mms', 'whatsapp', 'voice_calls'
            },
            'rateLimits' => new Map<String, Object>{
                'requests_per_second' => 10,
                'messages_per_day' => getSMSQuota()
            }
        };
    }

    private Map<String, Object> adaptSlackService() {
        return new Map<String, Object>{
            'type' => 'chat',
            'endpoint' => 'https://slack.com/api/chat.postMessage',
            'authentication' => 'bearer_token',
            'maxBatchSize' => 1,
            'supportedFeatures' => new List<String>{
                'rich_text', 'attachments', 'blocks', 'threads'
```

```
            },
            'rateLimits' => new Map<String, Object>{
                'requests_per_minute' => 50,
                'messages_per_channel_per_second' => 1
            }
        };
    }

    private Map<String, Object> adaptTeamsService() {
        return new Map<String, Object>{
            'type' => 'chat',
            'endpoint' => serviceConfig.get('webhook_url'),
            'authentication' => 'webhook',
            'maxBatchSize' => 1,
            'supportedFeatures' => new List<String>{
                'adaptive_cards', 'mentions', 'attachments'
            },
            'rateLimits' => new Map<String, Object>{
                'requests_per_second' => 4,
                'requests_per_app_per_second' => 4
            }
        };
    }

    private Boolean checkServiceAvailability() {
        // Lógica para verificar disponibilidad del servicio
        // Podría hacer una llamada de ping o verificar configuración
        return serviceConfig != null && !serviceConfig.isEmpty();
    }

    private Integer getEmailQuota() {
        // Obtener cuota de emails basada en el plan
        return (Integer) serviceConfig.get('email_quota') ?? 40000;
    }

    private Integer getSMSQuota() {
        // Obtener cuota de SMS basada en el plan
        return (Integer) serviceConfig.get('sms_quota') ?? 1000;
    }
}
```

Ejemplo 3: Adaptador para Servicios de Almacenamiento

```
public class StorageServiceAdapter implements Adapter {

    private String storageProvider;
    private Map<String, Object> credentials;

    public StorageServiceAdapter(String storageProvider, Map<String, Object> credentials) {
        this.storageProvider = storageProvider;
        this.credentials = credentials;
    }

    public Map<String, Object> adaptee() {
        Map<String, Object> storageInfo = new Map<String, Object>{
            'provider' => storageProvider,
            'adaptedAt' => System.now()
```

```
        };

        try {
            switch on storageProvider.toLowerCase() {
                when 'aws_s3' {
                    storageInfo.putAll(adaptAWSS3());
                }
                when 'azure_blob' {
                    storageInfo.putAll(adaptAzureBlob());
                }
                when 'google_cloud' {
                    storageInfo.putAll(adaptGoogleCloud());
                }
                when 'salesforce_files' {
                    storageInfo.putAll(adaptSalesforceFiles());
                }
                when else {
                    throw new UnsupportedStorageException('Provider not supported: ' + storageProvider);
                }
            }

            storageInfo.put('isConfigured', true);

        } catch (Exception e) {
            storageInfo.put('isConfigured', false);
            storageInfo.put('error', e.getMessage());
        }

        return storageInfo;
    }

    private Map<String, Object> adaptAWSS3() {
        return new Map<String, Object>{
            'baseUrl' => 'https://s3.amazonaws.com',
            'region' => credentials.get('aws_region') ?? 'us-east-1',
            'bucketName' => credentials.get('bucket_name'),
            'accessKeyId' => credentials.get('access_key_id'),
            'supportedOperations' => new List<String>{
                'upload', 'download', 'delete', 'list', 'copy'
            },
            'maxFileSize' => 5368709120, // 5GB
            'supportedFormats' => new List<String>{
                'pdf', 'jpg', 'png', 'docx', 'xlsx', 'zip'
            },
            'encryption' => new Map<String, Object>{
                'supported' => true,
                'types' => new List<String>{'AES256', 'aws:kms'}
            }
        };
    }

    private Map<String, Object> adaptAzureBlob() {
        return new Map<String, Object>{
            'baseUrl' => 'https://' + credentials.get('account_name') + '.blob.core.windows.net',
            'containerName' => credentials.get('container_name'),
            'accountName' => credentials.get('account_name'),
            'supportedOperations' => new List<String>{
```

```
                'upload', 'download', 'delete', 'list', 'snapshot'
            },
            'maxFileSize' => 4398046511104, // 4TB
            'supportedFormats' => new List<String>{
                'pdf', 'jpg', 'png', 'docx', 'xlsx', 'zip', 'mp4'
            },
            'encryption' => new Map<String, Object>{
                'supported' => true,
                'types' => new List<String>{'Microsoft Managed', 'Customer Managed'}
            }
        };
    }

    private Map<String, Object> adaptGoogleCloud() {
        return new Map<String, Object>{
            'baseUrl' => 'https://storage.googleapis.com',
            'bucketName' => credentials.get('bucket_name'),
            'projectId' => credentials.get('project_id'),
            'supportedOperations' => new List<String>{
                'upload', 'download', 'delete', 'list', 'versioning'
            },
            'maxFileSize' => 5497558138880, // 5TB
            'supportedFormats' => new List<String>{
                'pdf', 'jpg', 'png', 'docx', 'xlsx', 'zip', 'mp4', 'csv'
            },
            'encryption' => new Map<String, Object>{
                'supported' => true,
                'types' => new List<String>{'Google-managed', 'Customer-managed', 'Customer-supplied'}
            }
        };
    }

    private Map<String, Object> adaptSalesforceFiles() {
        return new Map<String, Object>{
            'baseUrl' => URL.getSalesforceBaseUrl().toExternalForm(),
            'supportedOperations' => new List<String>{
                'upload', 'download', 'delete', 'share', 'version'
            },
            'maxFileSize' => 2147483648, // 2GB
            'supportedFormats' => new List<String>{
                'pdf', 'jpg', 'png', 'docx', 'xlsx', 'pptx', 'txt'
            },
            'encryption' => new Map<String, Object>{
                'supported' => true,
                'types' => new List<String>{'Platform Encryption'}
            },
            'sharingModel' => 'Salesforce Security Model'
        };
    }

    public class UnsupportedStorageException extends Exception {}
}
```

Uso con Factory Pattern

```
public class AdapterFactory {

    public static Adapter createAdapter(String adapterType, Map<String, Object> configuration) {
        switch on adapterType.toLowerCase() {
            when 'payment' {
                String provider = (String) configuration.get('provider');
                return new PaymentServiceAdapter(provider, configuration);
            }
            when 'notification' {
                String service = (String) configuration.get('service');
                return new NotificationServiceAdapter(service, configuration);
            }
            when 'storage' {
                String provider = (String) configuration.get('provider');
                return new StorageServiceAdapter(provider, configuration);
            }
            when 'insurance' {
                return new InsServiceAdapter();
            }
            when else {
                throw new UnsupportedAdapterException('Unsupported adapter type: ' + adapterType);
            }
        }
    }

    public class UnsupportedAdapterException extends Exception {}
}

// Uso del factory
Map<String, Object> paymentConfig = new Map<String, Object>{
    'provider' => 'stripe',
    'stripe_public_key' => 'pk_test_...',
    'stripe_webhook_url' => 'https://myapp.com/webhooks/stripe'
};

Adapter paymentAdapter = AdapterFactory.createAdapter('payment', paymentConfig);
Map<String, Object> paymentInfo = paymentAdapter.adaptee();
```

Ejemplo de Uso en Servicios de Integración

```
public class IntegrationOrchestrator {

    private List<Adapter> adapters;

    public IntegrationOrchestrator() {
        this.adapters = new List<Adapter>();
    }

    public void addAdapter(Adapter adapter) {
        adapters.add(adapter);
    }

    public Map<String, Object> executeIntegration() {
        Map<String, Object> integrationResults = new Map<String, Object>{
            'totalAdapters' => adapters.size(),
```

```
            'results' => new List<Map<String, Object>>(),
            'executedAt' => System.now()
        };

        List<Map<String, Object>> results = new List<Map<String, Object>>();

        for (Integer i = 0; i < adapters.size(); i++) {
            try {
                Adapter adapter = adapters[i];
                Map<String, Object> adapterResult = adapter.adaptee();

                results.add(new Map<String, Object>{
                    'adapterIndex' => i,
                    'success' => true,
                    'data' => adapterResult
                });

            } catch (Exception e) {
                results.add(new Map<String, Object>{
                    'adapterIndex' => i,
                    'success' => false,
                    'error' => e.getMessage()
                });
            }
        }

        integrationResults.put('results', results);
        return integrationResults;
    }
}

// Uso del orquestador
IntegrationOrchestrator orchestrator = new IntegrationOrchestrator();

// Añadir múltiples adaptadores
orchestrator.addAdapter(AdapterFactory.createAdapter('payment', paymentConfig));
orchestrator.addAdapter(AdapterFactory.createAdapter('notification', notificationConfig));
orchestrator.addAdapter(AdapterFactory.createAdapter('storage', storageConfig));

// Ejecutar todas las integraciones
Map<String, Object> results = orchestrator.executeIntegration();
```

Mejores Prácticas para Implementación

1. **Consistencia en el Formato de Retorno**: Mantener una estructura consistente en el mapa de retorno para facilitar el procesamiento.

2. **Manejo de Errores**: Incluir información de errores en el mapa de retorno en lugar de lanzar excepciones cuando sea posible.

3. **Información de Metadatos**: Incluir metadatos útiles como timestamps, versiones de API, y información de configuración.

4. **Validación de Configuración**: Validar la configuración requerida antes de intentar adaptar el servicio.

5. **Documentación Clara**: Documentar claramente qué estructura de datos devuelve cada implementación.

Consideraciones de Diseño

Ventajas

1. **Flexibilidad**: Permite trabajar con múltiples servicios usando la misma interfaz.
2. **Mantenibilidad**: Facilita el mantenimiento al encapsular la lógica específica de cada servicio.
3. **Testabilidad**: Permite crear implementaciones mock fácilmente para pruebas.
4. **Extensibilidad**: Nuevos adaptadores pueden añadirse sin modificar código existente.

Consideraciones

1. **Complejidad**: Puede añadir complejidad al sistema si se abusa del patrón.
2. **Overhead**: Puede introducir overhead si la adaptación es muy simple.
3. **Consistencia**: Requiere disciplina para mantener consistencia entre implementaciones.

Notas Adicionales

1. La anotación `@namespaceAccessible` indica que esta interfaz está diseñada para ser utilizada en contextos de paquetes gestionados.
2. El comentario en la documentación del método muestra un parámetro vacío, lo que sugiere que originalmente se planeó que el método tuviera parámetros.
3. Esta interfaz forma parte de un sistema más amplio de adaptadores en el framework Foundation de Suratech.
4. La simplicidad de la interfaz es intencional, proporcionando máxima flexibilidad para las implementaciones.

RateProductHandler

Descripción General

`RateProductHandler` es una clase virtual global que extiende `ProductHandler` y se encarga de gestionar el proceso de calificación (rating) de un producto utilizando Integration Procedures.

Autor y Proyecto

- **Autor**: Soulberto Lorenzo soulberto@cloudblue.us (mailto:soulberto@cloudblue.us)
- **Última modificación**: 21 de febrero de 2025
- **Último modificador**: Jean Carlos Melendez

Constantes y Variables

```
public static final String PROCESS_NAME = 'Rating';
public static final String OPTION_SKIP_BEHAVIOR = 'skipDefaultBehavior';
public static final String RATING_IPROCEDURE = 'sfcore_ratingprocedure';
global IntegrationProcedureExecutor EXECUTOR;
public static final Set<String> REQUIRED_PARAMETERS = new Set<String>{
  'includeInputKeys',
  'instanceKey',
  'filters'
};
```

Constructores

```
global RateProductHandler(String productName, IntegrationProcedureExecutor executor)
```

Constructor que recibe el nombre del producto y un ejecutor de Integration Procedure.

```
global RateProductHandler(String productName)
```

Constructor que recibe solo el nombre del producto e inicializa un ejecutor mock.

Métodos Principales

setObservers

```
public override void setObservers()
```

Sobrescribe el método de la clase padre para añadir un observador de tipo `RateObserver`.

isValidForRatingParameters

```
global Boolean isValidForRatingParameters(Map<String, Object> data)
```

Verifica que todos los parámetros requeridos estén presentes en el mapa de datos:

- Recorre los parámetros requeridos definidos en `REQUIRED_PARAMETERS`
- Devuelve `false` si falta algún parámetro

- Devuelve `true` si todos los parámetros están presentes

process

```
public override Map<String, Object> process(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
)
```

Método principal que procesa el producto:

- Verifica si debe omitir el comportamiento predeterminado
- Establece el nombre del proceso como 'Rating'
- Valida los parámetros de entrada
- Construye el mapa de entrada para el Integration Procedure
- Ejecuta el Integration Procedure
- Actualiza el mapa de salida con los resultados
- Devuelve el mapa de salida actualizado

buildInputMap

```
public Map<String, Object> buildInputMap(
  Map<String, Object> input,
  Map<String, Object> options
)
```

Construye el mapa de entrada para el Integration Procedure:

- Crea una copia del mapa de entrada
- Agrega todas las opciones si no están vacías
- Devuelve el mapa combinado

executeIntegrationProcedure

```
public Map<String, Object> executeIntegrationProcedure(
  Map<String, Object> ipInput
)
```

Ejecuta el Integration Procedure definido en `RATING_IPROCEDURE`:

- Utiliza el ejecutor para invocar el procedimiento
- Devuelve un mapa con la clave 'resultProcess' conteniendo el resultado

Dependencias

- `ProductHandler`: Clase padre que extiende
- `IntegrationProcedureExecutor`: Interfaz o clase para ejecutar Integration Procedures
- `MockIntegrationProcedureExecutor`: Implementación mock del ejecutor

- `RateObserver`: Clase observadora para el proceso de rating
- `ProductHandlerImplementationException`: Excepción personalizada para errores de implementación

Notas Técnicas

- La clase utiliza el patrón Observer con `addObserver`
- Implementa validación de parámetros requeridos
- Utiliza el patrón Strategy para la ejecución de procedimientos de integración
- Gestiona opciones para omitir comportamientos predeterminados
- Proporciona manejo de errores con excepciones personalizadas

QuoteProductHandler

Descripción General

`QuoteProductHandler` es una clase virtual global que extiende `ProductHandler`. Esta clase está diseñada para gestionar el proceso de cotización de productos mediante la ejecución de procedimientos de integración, validando la estructura de los datos de entrada y gestionando los resultados.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

21 de febrero de 2025 por Jean Carlos Melendez

Constantes

Nombre	Tipo	Descripción
`PROCESS_NAME`	`String`	Nombre del proceso: 'QUOTING'
`MAIN_NODE`	`String`	Clave del nodo principal en los datos de entrada: 'quotepolicyJson'
`OPTION_SKIP_BEHAVIOR`	`String`	Opción para omitir el comportamiento predeterminado: 'skipDefaultBehavior'
`RATING_IPROCEDURE`	`String`	Nombre del procedimiento de integración para cotización: 'sfcore_quotingprocedure'
`REQUIRED_PARAMETERS`	`Set<String>`	Conjunto de parámetros requeridos para la cotización: 'term', 'productConfigurationDetail', 'insuredItems', 'additionalFields', 'OpportunityDetails'

Variables Globales

Nombre	Tipo	Descripción
`EXECUTOR`	`IntegrationProcedureExecutor`	Ejecutor de procedimientos de integración

Constructores

QuoteProductHandler(String productName, IntegrationProcedureExecutor executor)

Descripción

Constructor que inicializa el handler con un nombre de producto específico y un ejecutor de procedimientos de integración personalizado.

Parámetros

- `productName` (String): Nombre del producto a gestionar.
- `executor` (IntegrationProcedureExecutor): Ejecutor personalizado para los procedimientos de integración.

Código

```
global QuoteProductHandler(
  String productName,
  IntegrationProcedureExecutor executor
) {
  super(productName);
  this.EXECUTOR = executor;
}
```

QuoteProductHandler(String productName)

Descripción

Constructor alternativo que inicializa el handler con un nombre de producto específico y un ejecutor mock por defecto.

Parámetros

- `productName` (String): Nombre del producto a gestionar.

Código

```
global QuoteProductHandler(String productName) {
  super(productName);
  this.EXECUTOR = new MockIntegrationProcedureExecutor();
}
```

Métodos

setObservers()

Descripción

Método sobrescrito que configura los observadores para el handler, específicamente añadiendo un `QuoteObserver`.

Retorno

- `void`

Código

```
public override void setObservers() {
  this.addObserver(new QuoteObserver());
}
```

process(Map<String, Object> options, Map<String, Object> input, Map<String, Object> output)

Descripción

Método principal sobrescrito que procesa la cotización del producto utilizando un procedimiento de integración y gestiona los resultados.

Parámetros

- `options` (Map<String, Object>): Opciones adicionales para el proceso.
- `input` (Map<String, Object>): Datos de entrada para el proceso.
- `output` (Map<String, Object>): Mapa de salida donde se guardarán los resultados.

Retorno

- `Map<String, Object>`: Mapa con los datos de la cotización creada.

Proceso

1. Verifica si se debe omitir el comportamiento predeterminado según las opciones.
2. Valida la estructura de los datos de entrada utilizando `validateInputMapForQuoting`.
3. Si la validación falla, lanza una excepción con el mensaje de error.
4. Establece el nombre del proceso como 'Quoting'.
5. Ejecuta el procedimiento de integración.
6. Actualiza el mapa de salida con los resultados.
7. Establece el proceso de respuesta.
8. Añade los datos de entrada originales al mapa de salida.
9. Devuelve el mapa de salida completo.

Código

```
public override Map<String, Object> process(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
) {
  if (options?.containsKey(OPTION_SKIP_BEHAVIOR)) {
    return input;
  }

  Map<Boolean, Object> validation = QuoteProductHandler.validateInputMapForQuoting(
    input
  );
  if (validation.containsKey(false)) {
    throw new ProductHandlerImplementationException(
      validation.get(false).toString(),
      'UNPROCESSABLE_ENTITY',
      422
    );
  }

  this.setProcessName('Quoting');

  Map<String, Object> resultProcess = executeIntegrationProcedure(input);
  Core.getSymmaryLimits();
  output.clear();
  output.putAll(resultProcess);
  this.setResponseProcess(output);
  output.putAll(input);

  return output;
}
```

executeIntegrationProcedure(Map<String, Object> ipInput)

Descripción

Ejecuta el procedimiento de integración para la cotización del producto.

Parámetros

- `ipInput` (Map<String, Object>): Datos de entrada para el procedimiento de integración.

Retorno

- `Map<String, Object>`: Mapa con los resultados del procedimiento de integración.

Código

```
public Map<String, Object> executeIntegrationProcedure(
  Map<String, Object> ipInput
) {
  return new Map<String, Object>{
    'resultProcess' => EXECUTOR.execute(RATING_IPROCEDURE, ipInput)
  };
}
```

validateInputMapForQuoting(Map<String, Object> inputMap)

Descripción

Método estático que verifica si todos los nodos principales requeridos están presentes en el mapa de entrada.

Parámetros

- `inputMap` (Map<String, Object>): Mapa de entrada que contiene los datos a validar.

Retorno

- `Map<Boolean, Object>`: Mapa con el resultado de la validación. La clave es un booleano que indica si la validación fue exitosa, y el valor es un mensaje descriptivo.

Proceso

1. Verifica la existencia del nodo principal (`MAIN_NODE`).
2. Obtiene el mapa correspondiente al nodo principal.
3. Verifica la existencia de cada uno de los parámetros requeridos definidos en `REQUIRED_PARAMETERS`.
4. Devuelve un mapa con el resultado de la validación y un mensaje descriptivo.

Código

```
public static Map<Boolean, Object> validateInputMapForQuoting(
  Map<String, Object> inputMap
) {
  if (!inputMap.containsKey(MAIN_NODE) || inputMap?.get(MAIN_NODE) == null) {
    return createValidationResult(
      false,
      'Falta el nodo principal requerido: ' + MAIN_NODE
    );
  }
  Map<String, Object> mainNode = (Map<String, Object>) inputMap.get(
    MAIN_NODE
  );

  for (String node : REQUIRED_PARAMETERS) {
    if (!mainNode.containsKey(node)) {
      return createValidationResult(
        false,
```

```
        'Falta el nodo requerido: ' + node
      );
    }
  }
  System.debug('Todos los nodos requeridos están presentes.');
  return createValidationResult(
    true,
    'Todos los parámetros requeridos están presentes.'
  );
}
```

createValidationResult(Boolean isValid, String message)

Descripción

Método privado estático que crea un mapa estándar para representar el resultado de una validación.

Parámetros

- `isValid` (Boolean): Indicador de si la validación fue exitosa.
- `message` (String): Mensaje descriptivo del resultado.

Retorno

- `Map<Boolean, Object>`: Mapa con el estado de validación y el mensaje.

Código

```
private static Map<Boolean, Object> createValidationResult(
  Boolean isValid,
  String message
) {
  return new Map<Boolean, Object>{ isValid => message };
}
```

Excepciones

ProductHandlerImplementationException

Esta excepción se lanza cuando la validación de los datos de entrada falla, indicando qué nodo requerido está faltando. Se lanza con el código HTTP 422 (UNPROCESSABLE_ENTITY).

Dependencias

- `ProductHandler`: Clase base que `QuoteProductHandler` extiende.
- `IntegrationProcedureExecutor`: Interfaz o clase que define el método `execute` para ejecutar procedimientos de integración.
- `MockIntegrationProcedureExecutor`: Implementación mock de `IntegrationProcedureExecutor` utilizada en el constructor alternativo.
- `QuoteObserver`: Clase observadora añadida en el método `setObservers()`.
- `Core`: Clase utilitaria que proporciona métodos como `getSymmaryLimits()`.
- `ProductHandlerImplementationException`: Excepción personalizada lanzada cuando hay errores en el proceso.

Patrones de Diseño Implementados

Observer

A través del método `setObservers()`, la clase implementa el patrón Observer, permitiendo que otros objetos (específicamente, `QuoteObserver`) sean notificados de eventos durante el procesamiento.

Strategy

La clase utiliza el patrón Strategy al recibir un `IntegrationProcedureExecutor` que puede tener diferentes implementaciones, permitiendo cambiar el comportamiento de ejecución de procedimientos de integración sin modificar el código de la clase.

Flujo de Proceso

El proceso de cotización sigue el siguiente flujo:

1. **Validación**: Verifica que todos los nodos y parámetros requeridos estén presentes en los datos de entrada.
2. **Preparación**: Establece el nombre del proceso como 'Quoting'.
3. **Ejecución**: Llama al procedimiento de integración a través del executor.
4. **Gestión de Resultados**: Actualiza el mapa de salida con los resultados y añade los datos de entrada originales.
5. **Notificación**: Notifica a los observadores registrados a través del método heredado `setResponseProcess`.

Estructura de Datos de Entrada

Para que el proceso de cotización funcione correctamente, los datos de entrada deben tener la siguiente estructura:

```
{
  "quotepolicyJson": {
    "term": { /* ... */ },
    "productConfigurationDetail": { /* ... */ },
    "insuredItems": [ /* ... */ ],
    "additionalFields": { /* ... */ },
    "OpportunityDetails": { /* ... */ }
  },
  /* Otros campos opcionales */
}
```

Ejemplo de Uso

```
// Crear el ejecutor de procedimientos de integración
IntegrationProcedureExecutor executor = new SomeIntegrationProcedureExecutor();

// Instanciar el handler
QuoteProductHandler handler = new QuoteProductHandler('AutoInsurance', executor);

// Preparar los datos de entrada
Map<String, Object> term = new Map<String, Object>{
  'startDate' => Date.today(),
  'endDate' => Date.today().addYears(1)
};

Map<String, Object> productConfigDetail = new Map<String, Object>{
  'productCode' => 'AUTO-001',
  'coverageLevel' => 'Premium'
};

List<Map<String, Object>> insuredItems = new List<Map<String, Object>>{
  new Map<String, Object>{
```

```
    'type' => 'Vehicle',
    'make' => 'Toyota',
    'model' => 'Corolla',
    'year' => 2020
  }
};

Map<String, Object> additionalFields = new Map<String, Object>{
  'discountCode' => 'NEWCUST10'
};

Map<String, Object> opportunityDetails = new Map<String, Object>{
  'opportunityId' => '0061x00000AbCdEf',
  'accountId' => '0011x00000GhIjKl'
};

Map<String, Object> quotepolicyJson = new Map<String, Object>{
  'term' => term,
  'productConfigurationDetail' => productConfigDetail,
  'insuredItems' => insuredItems,
  'additionalFields' => additionalFields,
  'OpportunityDetails' => opportunityDetails
};

Map<String, Object> input = new Map<String, Object>{
  'quotepolicyJson' => quotepolicyJson
};

// Mapas para opciones y salida
Map<String, Object> options = new Map<String, Object>();
Map<String, Object> output = new Map<String, Object>();

try {
  // Procesar la cotización
  Map<String, Object> result = handler.process(options, input, output);

  // Obtener los resultados
  Map<String, Object> resultProcess = (Map<String, Object>)result.get('resultProcess');
  System.debug('Cotización completada. Resultado: ' + resultProcess);
} catch (ProductHandlerImplementationException e) {
  System.debug('Error en el proceso de cotización: ' + e.getMessage());
}
```

Extensión de la Clase

```
public class SpecializedQuoteHandler extends QuoteProductHandler {

    public SpecializedQuoteHandler(String productName) {
        super(productName);
    }

    // Sobrescribir el método para añadir validaciones adicionales
    public override Map<String, Object> process(
        Map<String, Object> options,
        Map<String, Object> input,
        Map<String, Object> output
```

```
    ) {
        // Realizar validaciones adicionales específicas
        Map<String, Object> quotepolicyJson = (Map<String, Object>)input.get(MAIN_NODE);
        if (quotepolicyJson != null) {
            Map<String, Object> term = (Map<String, Object>)quotepolicyJson.get('term');
            if (term != null) {
                Date startDate = (Date)term.get('startDate');
                Date endDate = (Date)term.get('endDate');

                if (startDate != null && endDate != null && startDate > endDate) {
                    throw new ProductHandlerImplementationException(
                        'La fecha de inicio no puede ser posterior a la fecha de fin',
                        'UNPROCESSABLE_ENTITY',
                        422
                    );
                }
            }
        }

        // Llamar al método base para continuar con el procesamiento estándar
        return super.process(options, input, output);
    }

    // Sobrescribir el método para personalizar la ejecución del procedimiento
    public override Map<String, Object> executeIntegrationProcedure(
        Map<String, Object> ipInput
    ) {
        // Agregar datos adicionales al input antes de ejecutar
        Map<String, Object> quotepolicyJson = (Map<String, Object>)ipInput.get(MAIN_NODE);
        Map<String, Object> additionalFields = (Map<String, Object>)quotepolicyJson.get('additionalFields');

        // Añadir un campo adicional
        additionalFields.put('handlerType', 'Specialized');

        // Ejecutar el procedimiento y procesar los resultados
        Map<String, Object> results = super.executeIntegrationProcedure(ipInput);

        // Realizar procesamientos adicionales en los resultados si es necesario

        return results;
    }
}
```

Notas Adicionales

1. La clase utiliza el operador de navegación segura (`?.`) para evitar excepciones de referencia nula al acceder a valores en mapas.
2. La clase implementa un mecanismo para omitir su comportamiento predeterminado si la opción `skipDefaultBehavior` está presente.
3. El método `executeIntegrationProcedure` encapsula la ejecución del procedimiento de integración, facilitando la prueba unitaria y la extensibilidad.
4. La validación de la estructura de datos de entrada se realiza a través de un método estático, lo que permite su reutilización en otras partes del código.
5. La salida del proceso incluye tanto los resultados del procedimiento de integración como los datos de entrada originales, proporcionando un contexto completo.

6. La llamada a `Core.getSymmaryLimits()` sugiere que se está realizando algún tipo de monitoreo o registro de los límites de recursos durante el proceso.

ProductHandlerNotImplementedException

Descripción General

`ProductHandlerNotImplementedException` es una clase de excepción personalizada que extiende la clase `Exception` estándar de Apex. Esta clase está diseñada para representar errores específicos que ocurren cuando se intenta utilizar un manejador de productos que no ha sido implementado correctamente o cuando se intenta acceder a funcionalidades no implementadas dentro de un manejador de productos.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

20 de agosto de 2024

Estructura

```
public with sharing class ProductHandlerNotImplementedException extends Exception {
}
```

La clase tiene una implementación minimalista, sin agregar métodos o propiedades adicionales más allá de los heredados de la clase base `Exception`.

Herencia

- **Clase Base**: `Exception`
- **Modificadores**: `public with sharing`

Propósito

El propósito principal de esta clase es proporcionar un tipo de excepción específico para errores relacionados con implementaciones faltantes o incompletas en manejadores de productos, permitiendo:

1. **Especificidad del Error**: Distinguir claramente los errores de "no implementado" de otros tipos de excepciones en el sistema de manejo de productos.
2. **Manejo Específico**: Permitir que el código que captura excepciones pueda identificar y manejar específicamente casos donde una funcionalidad requerida no está implementada.
3. **Claridad en el Código**: Mejorar la legibilidad del código al indicar explícitamente cuando se está tratando con un caso de funcionalidad no implementada.
4. **Documentación Implícita**: Servir como recordatorio para los desarrolladores de que ciertas funcionalidades aún necesitan ser implementadas.

Métodos Heredados

Al extender la clase `Exception` estándar, `ProductHandlerNotImplementedException` hereda los siguientes métodos y propiedades:

Constructores

- **`ProductHandlerNotImplementedException()`**: Constructor por defecto.
- **`ProductHandlerNotImplementedException(String message)`**: Constructor que acepta un mensaje de error.

- `ProductHandlerNotImplementedException(Exception cause)`: Constructor que acepta otra excepción como causa.
- `ProductHandlerNotImplementedException(String message, Exception cause)`: Constructor que acepta un mensaje y otra excepción como causa.

Métodos

- `getMessage()`: Devuelve el mensaje de error asociado con la excepción.
- `getCause()`: Devuelve la excepción que causó esta excepción, si existe.
- `getTypeName()`: Devuelve el nombre del tipo de excepción.
- `setMessage(String message)`: Establece el mensaje de error para la excepción.
- `getLineNumber()`: Devuelve el número de línea donde se generó la excepción.
- `getStackTraceString()`: Devuelve una representación de texto del seguimiento de la pila.

Ejemplos de Uso

Ejemplo 1: Método Abstracto No Implementado

```
public abstract class AbstractProductHandler {

    public abstract void processProduct(Product2 product);

    public void validateAndProcess(Product2 product) {
        // Validar el producto
        if (product == null) {
            throw new IllegalArgumentException('El producto no puede ser nulo');
        }

        // Ejecutar el procesamiento específico del producto
        try {
            processProduct(product);
        } catch (System.MethodNotImplementedException e) {
            // Capturar la excepción estándar y lanzar nuestra excepción personalizada
            throw new ProductHandlerNotImplementedException(
                'El método processProduct no está implementado para el manejador ' +
                this.getClass().getName(), e
            );
        }
    }
}

public class IncompleteProductHandler extends AbstractProductHandler {
    // Nota: No se implementa el método abstracto processProduct
    // Esto generará un error en tiempo de ejecución
}

// Uso
try {
    AbstractProductHandler handler = new IncompleteProductHandler();
    Product2 product = [SELECT Id, Name FROM Product2 LIMIT 1];
    handler.validateAndProcess(product);
} catch (ProductHandlerNotImplementedException e) {
    System.debug('Error: ' + e.getMessage());
```

```
    // Manejar el caso específico de método no implementado
}
```

Ejemplo 2: Funcionalidad No Implementada

```
public class GenericProductHandler {

    public void processProduct(String productType, Map<String, Object> productData) {
        switch on productType.toUpperCase() {
            when 'HARDWARE' {
                processHardwareProduct(productData);
            }
            when 'SOFTWARE' {
                processSoftwareProduct(productData);
            }
            when 'SERVICE' {
                // Esta funcionalidad aún no está implementada
                throw new ProductHandlerNotImplementedException(
                    'El procesamiento de productos de tipo SERVICE no está implementado todavía'
                );
            }
            when else {
                throw new IllegalArgumentException('Tipo de producto no reconocido: ' + productType);
            }
        }
    }

    private void processHardwareProduct(Map<String, Object> productData) {
        // Implementación para productos de hardware
    }

    private void processSoftwareProduct(Map<String, Object> productData) {
        // Implementación para productos de software
    }
}

// Uso
try {
    GenericProductHandler handler = new GenericProductHandler();
    Map<String, Object> productData = new Map<String, Object>{
        'name' => 'Servicio de Mantenimiento',
        'price' => 1200.00
    };
    handler.processProduct('SERVICE', productData);
} catch (ProductHandlerNotImplementedException e) {
    System.debug('Error: ' + e.getMessage());
    // Informar al usuario que la funcionalidad no está disponible
    ApexPages.addMessage(new ApexPages.Message(
        ApexPages.Severity.WARNING,
        'Esta funcionalidad estará disponible próximamente. Error: ' + e.getMessage()
    ));
}
```

Ejemplo 3: Implementación Parcial con Método Placeholder

```
public interface ProductHandler {
    void initialize();
    void process();
    void finalize();
}

public abstract class BaseProductHandler implements ProductHandler {

    public void initialize() {
        // Implementación común de inicialización
    }

    public abstract void process();

    public void finalize() {
        // Implementación común de finalización
    }
}

public class PartialProductHandler extends BaseProductHandler {

    // Implementación del método abstracto requerido, pero no completamente funcional
    public override void process() {
        // Esto es solo un placeholder, no una implementación completa
        throw new ProductHandlerNotImplementedException(
            'La implementación de process() está en desarrollo. Por favor, consulte la documentación para
alternativas.'
        );
    }
}

// Uso
try {
    ProductHandler handler = new PartialProductHandler();
    handler.initialize();
    handler.process(); // Esto lanzará ProductHandlerNotImplementedException
    handler.finalize();
} catch (ProductHandlerNotImplementedException e) {
    System.debug('Error: ' + e.getMessage());
    // Redirigir a una implementación alternativa o documentación
}
```

Relación con Otras Clases en el Sistema

Esta excepción está diseñada para trabajar en conjunto con:

1. **ProductHandler**: Interfaz o clase base que define el contrato para manejadores de productos.
2. **Implementaciones Concretas de ProductHandler**: Clases que implementan `ProductHandler` para tipos específicos de productos.
3. **QuoteProductHandler**: Manejador especializado para productos en cotizaciones.
4. **ProductHandlerImplementationException**: Otra excepción personalizada que puede ser utilizada para errores diferentes a "no implementado".

Mejores Prácticas para su Uso

1. **Mensajes Descriptivos**: Proporcionar mensajes de error claros y descriptivos que indiquen qué funcionalidad específica no está implementada.
2. **Documentación Adicional**: Cuando sea posible, incluir en el mensaje información sobre cuándo estará disponible la funcionalidad o alternativas.
3. **Uso Selectivo**: Utilizar esta excepción específicamente para casos de "no implementado", no para errores generales de implementación.
4. **Captura Específica**: Al capturar excepciones, manejar esta excepción específicamente para proporcionar información adecuada al usuario.

Implementación Extendida Propuesta

Aunque la implementación actual es minimalista, podría extenderse para incluir información adicional:

```
public with sharing class ProductHandlerNotImplementedException extends Exception {

    private String featureName;
    private String expectedAvailability;
    private String alternativePath;

    /**
     * Constructor por defecto
     */
    public ProductHandlerNotImplementedException() {
        super();
    }

    /**
     * Constructor con mensaje de error
     * @param message Mensaje descriptivo del error
     */
    public ProductHandlerNotImplementedException(String message) {
        super(message);
    }

    /**
     * Constructor con información detallada
     * @param message Mensaje descriptivo del error
     * @param featureName Nombre de la característica no implementada
     * @param expectedAvailability Fecha o versión esperada para la disponibilidad
     * @param alternativePath Ruta o método alternativo a utilizar mientras tanto
     */
    public ProductHandlerNotImplementedException(
        String message,
        String featureName,
        String expectedAvailability,
        String alternativePath
    ) {
        super(message);
        this.featureName = featureName;
        this.expectedAvailability = expectedAvailability;
        this.alternativePath = alternativePath;
    }

    /**
     * Constructor con excepción causa
```

```
     * @param message Mensaje descriptivo del error
     * @param cause Excepción que causó esta excepción
     */
    public ProductHandlerNotImplementedException(String message, Exception cause) {
        super(message, cause);
    }

    /**
     * Obtiene el nombre de la característica no implementada
     * @return Nombre de la característica
     */
    public String getFeatureName() {
        return this.featureName;
    }

    /**
     * Obtiene la fecha o versión esperada para la disponibilidad
     * @return Fecha o versión esperada
     */
    public String getExpectedAvailability() {
        return this.expectedAvailability;
    }

    /**
     * Obtiene la ruta o método alternativo sugerido
     * @return Ruta o método alternativo
     */
    public String getAlternativePath() {
        return this.alternativePath;
    }

    /**
     * Genera un mensaje completo con toda la información disponible
     * @return Mensaje completo
     */
    public String getFullMessage() {
        String fullMessage = getMessage();

        if (String.isNotBlank(featureName)) {
            fullMessage += '\nCaracterística: ' + featureName;
        }

        if (String.isNotBlank(expectedAvailability)) {
            fullMessage += '\nDisponibilidad esperada: ' + expectedAvailability;
        }

        if (String.isNotBlank(alternativePath)) {
            fullMessage += '\nAlternativa: ' + alternativePath;
        }

        return fullMessage;
    }
}
```

Ejemplo de Uso de la Implementación Extendida

```
try {
    throw new ProductHandlerNotImplementedException(
        'La generación automática de cotizaciones para productos de servicio no está implementada',
        'AutoQuoteService',
        'Q3 2023',
        'Utilizar el proceso manual a través de QuoteWizardController.generateManualQuote()'
    );
} catch (ProductHandlerNotImplementedException e) {
    // Registrar el error completo para depuración
    System.debug(e.getFullMessage());

    // Mostrar un mensaje amigable al usuario
    String userMessage = 'La funcionalidad solicitada aún no está disponible';

    if (String.isNotBlank(e.getExpectedAvailability())) {
        userMessage += ' (esperada para ' + e.getExpectedAvailability() + ')';
    }

    if (String.isNotBlank(e.getAlternativePath())) {
        userMessage += '. Mientras tanto, puede ' + e.getAlternativePath();
    }

    ApexPages.addMessage(new ApexPages.Message(
        ApexPages.Severity.INFO,
        userMessage
    ));
}
```

Consideraciones de Diseño

Ventajas de la Implementación Actual

1. **Simplicidad**: La implementación minimalista es fácil de entender y utilizar.
2. **Enfoque Específico**: Se centra exclusivamente en representar un error de "no implementado".
3. **Integración con el Manejo de Excepciones Existente**: Se integra perfectamente con los mecanismos estándar de manejo de excepciones de Apex.

Posibles Mejoras

1. **Información Adicional**: Podría extenderse para incluir información más detallada como se muestra en la implementación extendida propuesta.
2. **Métodos de Utilidad**: Podrían añadirse métodos de utilidad para facilitar la creación y el manejo de esta excepción.
3. **Integración con Sistemas de Registro**: Podría añadirse funcionalidad para registrar automáticamente estas excepciones en un sistema de seguimiento.

Notas Adicionales

1. La clase utiliza el modificador `with sharing`, lo que significa que respetará las reglas de compartición de Salesforce durante su ejecución. Esto es apropiado para excepciones que se utilizan en contextos donde la seguridad es importante.
2. La excepción es parte de un ecosistema más amplio de manejo de productos, trabajando en conjunto con otras clases relacionadas.

3. El nombre de la clase indica claramente su propósito, lo que es una buena práctica para mejorar la legibilidad y mantenibilidad del código.
4. Es una práctica común en desarrollo de software crear excepciones específicas para diferentes tipos de errores, como se hace aquí para el caso de funcionalidades no implementadas.

ProductHandlerImplementationException

Descripción General

`ProductHandlerImplementationException` es una clase global de excepción personalizada que extiende la clase `Exception` estándar de Apex. Esta clase está diseñada específicamente para representar errores que ocurren durante la implementación y ejecución de manejadores de productos, proporcionando información adicional como códigos de error y códigos de estado HTTP.

Autor

Jean Carlos Melendez

Última Modificación

4 de febrero de 2025

Estructura

```
global class ProductHandlerImplementationException extends Exception {
   String msg;
   String errorCode;
   Integer errorStatusCode;

   global ProductHandlerImplementationException(String msg, String errorCode, Integer errorStatusCode) {
      this.msg = msg;
      this.errorCode = errorCode;
      this.errorStatusCode = errorStatusCode;
   }

   global override String getMessage() {
      return this.msg;
   }

   global String getErrorCode() {
      return this.errorCode;
   }

   global Integer getStatusCode() {
      return this.errorStatusCode;
   }
}
```

Propiedades

msg

- **Tipo**: `String`
- **Descripción**: Mensaje que describe el error.

errorCode

- **Tipo**: `String`
- **Descripción**: Código de error específico que categoriza el tipo de error.

errorStatusCode

- **Tipo**: `Integer`
- **Descripción**: Código de estado HTTP correspondiente al error, útil para mapear excepciones a respuestas HTTP.

Constructor

ProductHandlerImplementationException(String msg, String errorCode, Integer errorStatusCode)

Descripción

Constructor que inicializa una nueva instancia de la excepción con un mensaje, código de error y código de estado.

Parámetros

- `msg` (String): Mensaje descriptivo del error.
- `errorCode` (String): Código de error específico.
- `errorStatusCode` (Integer): Código de estado HTTP correspondiente.

Código

```
global ProductHandlerImplementationException(String msg, String errorCode, Integer errorStatusCode) {
   this.msg = msg;
   this.errorCode = errorCode;
   this.errorStatusCode = errorStatusCode;
}
```

Métodos

getMessage()

Descripción

Método sobrescrito que devuelve el mensaje de error asociado con la excepción.

Parámetros

Ninguno.

Retorno

- `String`: El mensaje de error.

Código

```
global override String getMessage() {
   return this.msg;
}
```

getErrorCode()

Descripción

Método que devuelve el código de error específico asociado con la excepción.

Parámetros

Ninguno.

Retorno

- `String`: El código de error.

Código

```
global String getErrorCode() {
   return this.errorCode;
}
```

getStatusCode()

Descripción

Método que devuelve el código de estado HTTP asociado con la excepción.

Parámetros

Ninguno.

Retorno

- `Integer`: El código de estado HTTP.

Código

```
global Integer getStatusCode() {
   return this.errorStatusCode;
}
```

Códigos de Error Comunes

A continuación se presenta una tabla de códigos de error comunes que se utilizan con esta excepción:

Código de Error	Descripción	Código de Estado Típico
`BAD_REQUEST`	La solicitud es inválida o carece de parámetros requeridos.	400
`UNAUTHORIZED`	La solicitud requiere autenticación o la autenticación proporcionada es insuficiente.	401
`FORBIDDEN`	La autenticación es válida pero no tiene permisos para el recurso.	403
`NOT_FOUND`	El recurso o producto solicitado no se encuentra.	404
`UNPROCESSABLE_ENTITY`	La solicitud es válida pero no se puede procesar debido a errores semánticos.	422
`INTERNAL_ERROR`	Ocurrió un error interno inesperado.	500
`SERVICE_UNAVAILABLE`	El servicio externo requerido no está disponible.	503

Ejemplos de Uso

Ejemplo 1: Validación de Campos Requeridos

```
public class ProductValidator {
    public static void validateRequiredFields(Map<String, Object> productData) {
        if (productData == null) {
            throw new ProductHandlerImplementationException(
                'Datos del producto no proporcionados',
```

```
                'BAD_REQUEST',
                400
            );
        }

        if (!productData.containsKey('productCode') || String.isBlank((String)productData.get('productCode')))
{
            throw new ProductHandlerImplementationException(
                'El código del producto es requerido',
                'UNPROCESSABLE_ENTITY',
                422
            );
        }

        if (!productData.containsKey('price') || productData.get('price') == null) {
            throw new ProductHandlerImplementationException(
                'El precio del producto es requerido',
                'UNPROCESSABLE_ENTITY',
                422
            );
        }
    }
}
```

Ejemplo 2: Verificación de Existencia de Producto

```
public class ProductFinder {
    public static Product2 findProductById(Id productId) {
        List<Product2> products = [SELECT Id, Name, ProductCode, IsActive FROM Product2 WHERE Id = :productId
LIMIT 1];

        if (products.isEmpty()) {
            throw new ProductHandlerImplementationException(
                'Producto no encontrado con ID: ' + productId,
                'NOT_FOUND',
                404
            );
        }

        Product2 product = products[0];

        if (!product.IsActive) {
            throw new ProductHandlerImplementationException(
                'El producto está inactivo: ' + product.Name,
                'UNPROCESSABLE_ENTITY',
                422
            );
        }

        return product;
    }
}
```

Ejemplo 3: Manejo de Errores en un Controlador REST

```
@RestResource(urlMapping='/products/*')
global class ProductRESTService {

    @HttpGet
    global static void getProduct() {
        RestRequest req = RestContext.request;
        RestResponse res = RestContext.response;

        try {
            String productId = req.requestURI.substring(req.requestURI.lastIndexOf('/') + 1);

            if (String.isBlank(productId)) {
                throw new ProductHandlerImplementationException(
                    'ID de producto no proporcionado',
                    'BAD_REQUEST',
                    400
                );
            }

            Product2 product = ProductFinder.findProductById(productId);

            Map<String, Object> result = new Map<String, Object>{
                'success' => true,
                'product' => new Map<String, Object>{
                    'id' => product.Id,
                    'name' => product.Name,
                    'code' => product.ProductCode
                }
            };

            res.responseBody = Blob.valueOf(JSON.serialize(result));
            res.statusCode = 200;
        } catch (ProductHandlerImplementationException e) {
            handleException(e, res);
        } catch (Exception e) {
            // Convertir excepciones estándar a nuestra excepción personalizada
            ProductHandlerImplementationException customEx = new ProductHandlerImplementationException(
                'Error inesperado: ' + e.getMessage(),
                'INTERNAL_ERROR',
                500
            );
            handleException(customEx, res);
        }
    }

    private static void handleException(ProductHandlerImplementationException e, RestResponse res) {
        Map<String, Object> errorResponse = new Map<String, Object>{
            'success' => false,
            'errorCode' => e.getErrorCode(),
            'message' => e.getMessage()
        };

        res.responseBody = Blob.valueOf(JSON.serialize(errorResponse));
        res.statusCode = e.getStatusCode();
```

```
    }
}
```

Ejemplo 4: Uso en Manejadores de Productos

```
public class AutoInsuranceProductHandler extends ProductHandler {

    public override Map<String, Object> process(
        Map<String, Object> options,
        Map<String, Object> input,
        Map<String, Object> output
    ) {
        try {
            // Validar datos de entrada
            validateInput(input);

            // Procesar el producto
            // ...

            return output;
        } catch (Exception e) {
            if (e instanceof ProductHandlerImplementationException) {
                throw e;
            } else {
                throw new ProductHandlerImplementationException(
                    'Error procesando el producto Auto Insurance: ' + e.getMessage(),
                    'INTERNAL_ERROR',
                    500
                );
            }
        }
    }

    private void validateInput(Map<String, Object> input) {
        if (!input.containsKey('vehicleData')) {
            throw new ProductHandlerImplementationException(
                'Datos del vehículo no proporcionados',
                'UNPROCESSABLE_ENTITY',
                422
            );
        }

        Map<String, Object> vehicleData = (Map<String, Object>)input.get('vehicleData');

        if (!vehicleData.containsKey('make') || String.isBlank((String)vehicleData.get('make'))) {
            throw new ProductHandlerImplementationException(
                'La marca del vehículo es requerida',
                'UNPROCESSABLE_ENTITY',
                422
            );
        }

        if (!vehicleData.containsKey('model') || String.isBlank((String)vehicleData.get('model'))) {
            throw new ProductHandlerImplementationException(
                'El modelo del vehículo es requerido',
                'UNPROCESSABLE_ENTITY',
                422
```

```
            );
        }

        if (!vehicleData.containsKey('year') || vehicleData.get('year') == null) {
            throw new ProductHandlerImplementationException(
                'El año del vehículo es requerido',
                'UNPROCESSABLE_ENTITY',
                422
            );
        }
    }
}
```

Captura y Manejo de la Excepción

```
try {
    // Código que podría lanzar la excepción
    Map<String, Object> productData = new Map<String, Object>();
    ProductValidator.validateRequiredFields(productData);
} catch (ProductHandlerImplementationException e) {
    System.debug('Error: ' + e.getMessage());
    System.debug('Código de error: ' + e.getErrorCode());
    System.debug('Código de estado: ' + e.getStatusCode());

    // Manejar el error según su código
    switch on e.getErrorCode() {
        when 'BAD_REQUEST' {
            // Lógica para manejar errores de solicitud incorrecta
        }
        when 'UNPROCESSABLE_ENTITY' {
            // Lógica para manejar errores de validación
        }
        when 'NOT_FOUND' {
            // Lógica para manejar recursos no encontrados
        }
        when else {
            // Lógica para manejar otros errores
        }
    }
}
```

Uso en Pruebas Unitarias

```
@IsTest
private class ProductHandlerImplementationExceptionTest {

    @IsTest
    static void testConstructorAndGetters() {
        // Configurar
        String message = 'Error de prueba';
        String errorCode = 'TEST_ERROR';
        Integer statusCode = 418; // I'm a teapot

        // Ejecutar
        ProductHandlerImplementationException ex = new ProductHandlerImplementationException(
```

```
            message, errorCode, statusCode
        );

        // Verificar
        System.assertEquals(message, ex.getMessage(), 'El mensaje debe coincidir');
        System.assertEquals(errorCode, ex.getErrorCode(), 'El código de error debe coincidir');
        System.assertEquals(statusCode, ex.getStatusCode(), 'El código de estado debe coincidir');
    }

    @IsTest
    static void testExceptionInProductValidator() {
        // Configurar
        Map<String, Object> emptyData = new Map<String, Object>();

        // Ejecutar y verificar
        try {
            ProductValidator.validateRequiredFields(emptyData);
            System.assert(false, 'Debería haber lanzado una excepción');
        } catch (ProductHandlerImplementationException e) {
            System.assertEquals('UNPROCESSABLE_ENTITY', e.getErrorCode(), 'Debería ser un error de entidad no
procesable');
            System.assertEquals(422, e.getStatusCode(), 'El código de estado debería ser 422');
        }
    }
}
```

Mejores Prácticas

1. **Códigos de Error Consistentes**: Utilizar un conjunto consistente de códigos de error en toda la aplicación.
2. **Mensajes Descriptivos**: Proporcionar mensajes de error claros y descriptivos.
3. **Códigos de Estado HTTP Apropiados**: Usar códigos de estado HTTP estándar que se alineen con el tipo de error.
4. **Manejo Centralizado**: Implementar un manejo centralizado de excepciones en puntos de entrada como servicios REST.
5. **Registro de Errores**: Registrar los errores para análisis y depuración futuros.

Integración con Sistemas más Amplios

La clase `ProductHandlerImplementationException` está diseñada para integrarse con el sistema más amplio de manejo de productos:

1. **ProductHandler y Derivados**: Clases como `QuoteProductHandler`, `IssueProductHandler`, etc., utilizan esta excepción para gestionar errores.
2. **Servicios REST**: Los servicios REST pueden capturar y transformar estas excepciones en respuestas HTTP estructuradas.
3. **Procedimientos de Integración**: Los procedimientos de integración pueden capturar estas excepciones y manejarlas adecuadamente.

Consideraciones de Rendimiento

1. **Creación de Instancias**: La creación de objetos de excepción tiene un costo en términos de rendimiento y debe utilizarse para condiciones de error genuinas, no para control de flujo normal.
2. **Propagación de Excepciones**: Propagar excepciones a través de múltiples niveles de la pila de llamadas puede tener un impacto en el rendimiento. Considerar capturar y manejar excepciones lo más cerca posible de su origen cuando sea apropiado.

Notas Adicionales

1. La clase está marcada como `global`, lo que indica que está diseñada para ser accesible desde diferentes paquetes o namespaces.
2. La implementación actual no guarda la excepción causa, lo que podría ser una mejora para considerar en futuras versiones.
3. Los códigos de estado HTTP utilizados (como 422 para `UNPROCESSABLE_ENTITY`) se alinean con los estándares HTTP, lo que facilita la integración con servicios web.
4. Esta excepción proporciona una estructura estandarizada para manejo de errores, lo que mejora la consistencia en toda la aplicación.

Documentación Técnica - Objeto `Subconjunto Registro de Evento`

Este objeto está destinado a registrar subconjuntos o trazabilidad de eventos internos, permitiendo una integración ordenada, monitoreo estructurado y separación de responsabilidades entre plataformas o procesos involucrados.

Información General

Atributo	Valor
Nombre API	`Subconjunto_Registro_de_Evento__c`
Etiqueta(Label)	Subconjunto Registro de Evento
Etiqueta(Label) Plural	Subconjunto Registros de Eventos
Estado de implementación	`Deployed`
Namespace de acceso	`SFCore`
Modelo de uso compartido interno	`ReadWrite`
Modelo de uso compartido externo	`Private`

📄 Campos Personalizados

🟦 Data__c

- **Etiqueta(Label):** Dato
- **Tipo:** Área de texto larga (`LongTextArea`)
- **Longitud máxima:** 32.768 caracteres
- **Líneas visibles:** 3
- **Uso:** Almacena los datos de entrada para cada Handler (proceso de manejo) o integración (proceso de integración).
- **Requiere valor:** No

🗓️ Date__c

- **Etiqueta(Label):** Fecha
- **Tipo:** Fecha y hora (`DateTime`)
- **Valor predeterminado:** `NOW()`
- **Uso:** Registra la fecha y hora del almacenamiento del subconjunto.
- **Requiere valor:** Sí

📋 Details__c

- **Etiqueta(Label):** Detalle
- **Tipo:** Área de texto larga (`LongTextArea`)
- **Longitud máxima:** 131.072 caracteres
- **Líneas visibles:** 3
- **Uso:** Almacena descripciones detalladas o trazabilidad del evento.
- **Requiere valor:** No

🧭 Level__c

- **Etiqueta(Label):** Nivel
- **Tipo:** Texto (`Text`)
- **Longitud máxima:** 30 caracteres
- **Uso:** Nivel o severidad del evento (ej. INFO, WARNING, ERROR, SUCCESS).
- **Requiere valor:** Sí

💻 Platform__c

- **Etiqueta(Label):** Plataforma
- **Tipo:** Texto (`Text`)
- **Longitud máxima:** 50 caracteres
- **Uso:** Plataforma tecnológica desde la cual se origina el evento. (ej, Salesfoce, Formulario placa, etc)
- **Requiere valor:** No

⚙️ Process__c

- **Etiqueta(Label):** Proceso
- **Tipo:** Texto (`Text`)
- **Longitud máxima:** 80 caracteres
- **Uso:** Identificador del proceso responsable del evento (Proceso de Lead, Tarifa, Emision, Fasecolda, etc).
- **Requiere valor:** No

🛒 Product__c

- **Etiqueta(Label):** Producto
- **Tipo:** Texto (`Text`)
- **Longitud máxima:** 30 caracteres
- **Uso:** Producto asociado al evento (ej. Motos, Autos, Viajes).
- **Requiere valor:** No

💬 Response__c

- **Etiqueta(Label):** Response

- **Tipo:** Área de texto larga (`LongTextArea`)
- **Longitud máxima:** 32.768 caracteres
- **Líneas visibles:** 3
- **Uso:** Guarda la respuesta de un servicio externo o proceso interno.
- **Requiere valor:** No

🏷️ Type__c

- **Etiqueta(Label):** Tipo de Registro
- **Tipo:** Texto (`Text`)
- **Longitud máxima:** 30 caracteres
- **Uso:** Define el tipo específico del registro (Handler, Integración).
- **Requiere valor:** No

👤 UserId__c

- **Etiqueta(Label):** ID de Usuario
- **Tipo:** Texto (`Text`)
- **Longitud máxima:** 30 caracteres
- **Uso:** Identificador del usuario responsable del evento.
- **Requiere valor:** No

Ejemplo Practico

Al Crear un lead de viajes y dejar el registro de Log para su trazabilidad, tenemos dentro del registro del objeto lo siguiente:

- **Registro de Evento Name:** Evento de LEAD
- **Fecha:** Fecha de creación
- **ID de Usuario:** ID de Usuario
- **Producto:** Viajes
- **Plataforma:** Salesforce
- **Dato:**--Datos de entrada del servicio -- "{\"Stage\":\"KNOWING\",\"options\": {},\"SESION_ID\":\"1747677805101\",\"assessorCode\":\"4999\",\"productName\":\"Viajes\",\"accept_tos\":true,\"email\":\"jo@test.us\",\"
- **Response:**--Respuesta del servicio -- {\"type\":\"Lead\",\"url\":\"/services/data/v64.0/sobjects/Lead/00QU900000Cl3LKMAZ\"},\"Status\":\"0_Abierto\",\"Id\":\"00QU90000 11-01\",\"SURALastUpgPersonDataProccess__c\":\"2020-10-29\",\"RecordTypeId\":\"012D60000045fd1IAA\",\"SURAProducerId__c\":\"0YxD60000008UqJKAU\"}"
- **Detalle:** manejador de producto Viajes para proceso LEAD
- **Proceso:** Lead
- **Tipo de registro:** Handler(Proceso)
- **Nivel:** success

🧠 Observaciones Técnicas

- Utiliza diseño compacto del sistema (`SYSTEM`).
- Puede utilizarse en reportes y vía API o Streaming API.
- No está indexado para búsqueda global.

LeadProductHandler

Descripción General

`LeadProductHandler` es una clase virtual global que extiende `ProductHandler`. Esta clase está diseñada para manejar el procesamiento de leads como productos, siguiendo un patrón de diseño que facilita la creación y configuración flexible de objetos Lead en Salesforce.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

21 de febrero de 2025 por Jean Carlos Melendez

Constantes

PROCESS_NAME

- **Valor**: `"KNOWING"`
- **Descripción**: Constante que define el nombre del proceso asociado a este handler.

Constructor

LeadProductHandler(String productName)

Descripción

Constructor que inicializa un nuevo handler con un nombre de producto específico, llamando al constructor de la clase base.

Parámetros

- `productName` (String): Nombre del producto asociado a este handler.

Código

```
global LeadProductHandler(String productName) {
  super(productName);
}
```

Métodos

setObservers()

Descripción

Método sobrescrito que configura los observadores para el handler, específicamente añadiendo un `LeadObserver`.

Retorno

- `void`

Código

```
public override void setObservers() {
  this.addObserver(new LeadObserver());
}
```

process(Map<String, Object> options, Map<String, Object> input, Map<String, Object> output)

Descripción

Método principal sobrescrito que procesa la creación de un Lead utilizando los datos de entrada proporcionados, aplicando opciones adicionales y almacenando el resultado en el mapa de salida.

Parámetros

- `options` (Map<String, Object>): Opciones adicionales para el proceso.
- `input` (Map<String, Object>): Datos de entrada para la creación del Lead.
- `output` (Map<String, Object>): Mapa de salida donde se almacenará el resultado.

Retorno

- `Map<String, Object>`: Mapa con los datos del Lead creado.

Proceso

1. Verifica si se debe omitir el comportamiento predeterminado según las opciones.
2. Valida que ningún valor en el mapa de entrada sea nulo.
3. Establece el nombre del proceso utilizando la constante `PROCESS_NAME`.
4. Obtiene los configuradores de Lead del mapa de entrada.
5. Utiliza `LeadBuilderFoundation` con un patrón builder para crear un objeto Lead.
6. Guarda el Lead creado en el mapa de salida.
7. Establece el proceso de respuesta.
8. Maneja excepciones de DML y generales, lanzando `ProductHandlerImplementationException` con el código 422.

Código

```
public override Map<String, Object> process(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
) {
  if (options?.containsKey('skipDefaultBehavior')) {
    return input;
  }

  for (String key : input.keySet()) {
    if (input?.get(key) == null) {
      Core.debug(
        key +
        ' must not be empty, throwing product handler implementation custom exception'
      );
      throw new ProductHandlerImplementationException(
        key + ' must not be empty',
        'UNPROCESSABLE_ENTITY',
        422
      );
    }
  }
  setProcessName(PROCESS_NAME);
```

```
  List<LeadConfigurator> configurators = new List<LeadConfigurator>();
  configurators = (List<LeadConfigurator>) input?.get('configurators');
  try {
    Lead newLead = LeadBuilderFoundation.createLead(
      new LeadBuilderFoundation()
        .setStatus((String) input?.get('status') ?? '0_Abierto')
        .setId((Id) input?.get('leadId'))
        .setLastName((String) input?.get('lastName'))
        .setFirstName((String) input?.get('firstName'))
        .setEmail((String) input?.get('email'))
        .setMobile((String) input?.get('mobile_phone'))
        .setPhone((String) input?.get('phone'))
        .addConfigurators(configurators ?? new List<LeadConfigurator>())
        .build()
    );

    Core.debug('Finishing lead trip process...' + newLead);
    output.put('resultProcess', newLead);
    this.setResponseProcess((newLead));
    return output;

    // TODO: Implement exception handling Pipe
  } catch (DmlException dmlEx) {
    System.debug(
      'DML Exception while processing lead: ' + dmlEx.getDmlMessage(0)
    );
    throw new ProductHandlerImplementationException(
      dmlEx.getMessage(),
      'UNPROCESSABLE_ENTITY',
      422
    );
  } catch (Exception e) {
    //System.debug(e.getCause().toString() + ' ' + e.getMessage());
    throw new ProductHandlerImplementationException(
      e.getMessage(),
      'UNPROCESSABLE_ENTITY',
      422
    );
  }
}
```

isValidForLeadRequirements(Map<String, Object> data)

Descripción

Método virtual que valida los requerimientos de negocio para la creación de un Lead. Este método está diseñado para ser sobrescrito por clases derivadas concretas que implementarán lógicas específicas de validación.

Parámetros

- `data` (Map<String, Object>): Datos a validar.

Retorno

- `Boolean`: `true` si los datos cumplen con los requisitos, `false` en caso contrario. La implementación base siempre retorna `true`.

Código

```
global virtual Boolean isValidForLeadRequirements(Map<String, Object> data) {
  /**
   * Se sobreescribira el codigo que evaluará la logica
   * y los requerimientos de negocios para creación de Lead...
   **/
  return true;
}
```

Excepciones

ProductHandlerImplementationException

Esta excepción se lanza en las siguientes situaciones:

1. Cuando un valor requerido en el mapa de entrada es nulo.
2. Cuando ocurre una excepción de DML durante la creación del Lead.
3. Cuando ocurre cualquier otra excepción durante el proceso.

En todos los casos, la excepción se lanza con un código HTTP 422 (UNPROCESSABLE_ENTITY).

Dependencias

- `ProductHandler`: Clase base que `LeadProductHandler` extiende.
- `LeadObserver`: Clase observadora añadida en el método `setObservers()`.
- `Core`: Clase utilitaria que proporciona el método `debug`.
- `ProductHandlerImplementationException`: Excepción personalizada lanzada cuando hay errores en el proceso.
- `LeadBuilderFoundation`: Clase que implementa el patrón Builder para crear objetos Lead.
- `LeadConfigurator`: Interfaz o clase que define configuradores para personalizar objetos Lead.
- `Lead`: Objeto estándar de Salesforce utilizado en el proceso.

Patrones de Diseño Implementados

Observer

A través del método `setObservers()`, la clase implementa el patrón Observer, permitiendo que otros objetos (específicamente, `LeadObserver`) sean notificados de eventos durante el procesamiento.

Builder

Aunque no implementado directamente en esta clase, `LeadProductHandler` utiliza el patrón Builder a través de `LeadBuilderFoundation` para crear objetos Lead de manera flexible.

Template Method

Como clase virtual con métodos que pueden ser sobrescritos, `LeadProductHandler` implementa el patrón Template Method, definiendo el esqueleto de un algoritmo pero permitiendo que las subclases sobrescriban pasos específicos.

Ejemplo de Uso

```
// Crear una instancia del handler
LeadProductHandler handler = new LeadProductHandler('LeadGeneration');

// Preparar los datos de entrada
```

```
Map<String, Object> input = new Map<String, Object>{
  'lastName' => 'Pérez',
  'firstName' => 'Juan',
  'email' => 'juan.perez@example.com',
  'phone' => '555-1234',
  'mobile_phone' => '555-5678',
  'status' => '0_Abierto'
};

// Configuradores personalizados (opcional)
List<LeadConfigurator> configurators = new List<LeadConfigurator>{
  new CompanyNameConfigurator('ACME Inc.'),
  new LeadSourceConfigurator('Web')
};
input.put('configurators', configurators);

// Mapas para opciones y salida
Map<String, Object> options = new Map<String, Object>();
Map<String, Object> output = new Map<String, Object>();

try {
  // Procesar la creación del Lead
  Map<String, Object> result = handler.process(options, input, output);

  // Obtener el Lead creado
  Lead newLead = (Lead)result.get('resultProcess');
  System.debug('Lead creado: ' + newLead.Id);
} catch (ProductHandlerImplementationException e) {
  System.debug('Error: ' + e.getMessage());
}
```

Extensión de la Clase

```
public class PremiumLeadHandler extends LeadProductHandler {

    public PremiumLeadHandler() {
        super('PremiumLead');
    }

    // Sobrescribir para implementar validaciones específicas
    global override Boolean isValidForLeadRequirements(Map<String, Object> data) {
        // Validar que sea un lead premium (ejemplo: debe tener email y teléfono)
        String email = (String)data.get('email');
        String phone = (String)data.get('phone');

        return !String.isBlank(email) && !String.isBlank(phone);
    }

    // Opcionalmente sobrescribir process para añadir lógica específica
    public override Map<String, Object> process(
        Map<String, Object> options,
        Map<String, Object> input,
        Map<String, Object> output
    ) {
        // Validar requisitos específicos antes de procesar
        if (!isValidForLeadRequirements(input)) {
```

```
            throw new ProductHandlerImplementationException(
                'Este lead no cumple con los requisitos para ser premium',
                'UNPROCESSABLE_ENTITY',
                422
            );
        }

        // Añadir configurador específico para leads premium
        List<LeadConfigurator> configurators = (List<LeadConfigurator>)input.get('configurators');
        if (configurators == null) {
            configurators = new List<LeadConfigurator>();
            input.put('configurators', configurators);
        }

        configurators.add(new PremiumLeadConfigurator());

        // Llamar al proceso base
        return super.process(options, input, output);
    }
}
```

Notas Adicionales

1. La clase usa el operador de navegación segura (`?.`) y el operador de coalescencia nula (`??`) para evitar excepciones de referencia nula, lo que indica que está destinada a ejecutarse en versiones recientes de Apex.
2. Hay un comentario TODO para implementar un manejo de excepciones con "Pipe", lo que sugiere que podría haber planes para mejorar el manejo de errores en el futuro.
3. El método `process()` valida que todos los valores de entrada no sean nulos, lo que podría ser restrictivo en algunos casos. Considere implementar una lista de campos requeridos específicos en lugar de exigir que todos los campos tengan un valor.
4. La clase implementa un mecanismo para omitir su comportamiento predeterminado si la opción `skipDefaultBehavior` está presente, lo que proporciona flexibilidad para casos de uso especiales.

IssueProductHandler

Descripción General

`IssueProductHandler` es una clase virtual global que extiende `ProductHandler` y está diseñada para manejar la emisión de productos de seguros a través de procedimientos de integración. Esta clase implementa la lógica necesaria para validar datos, asignar cuentas a cotizaciones y gestionar el proceso de emisión de pólizas.

Autor

Jean Carlos Melendez \<jean@cloudblue.us\>

Última Modificación

21 de febrero de 2025

Constantes

Nombre	Tipo	Descripción
`FLAG_PLATFORM_NAME`	`String`	Nombre del indicador de plataforma: 'sura_foundation_platform_events'
`IP_PREPROCESS`	`String`	Nombre del procedimiento de preprocesamiento: 'sfcore_preandpostquotingprocess'
`PROCESS_NAME`	`String`	Nombre del proceso: 'Issuing'
`OPTION_SKIP_BEHAVIOR`	`String`	Opción para omitir el comportamiento predeterminado: 'skipDefaultBehavior'
`ISSUING_IPROCEDURE`	`String`	Nombre del procedimiento de integración para emisión: 'SURAFoundation_IssuingProcedure'
`ERROR_REQUIRED_FIELD`	`String`	Mensaje de error para campos requeridos: ' es requerido'
`REQUIRED_FIELD_FOR_ISSUE_CORE`	`List<String>`	Lista de campos requeridos para la emisión: 'quoteId', 'effectiveDate', 'producerId'

Variables Globales

Nombre	Tipo	Descripción
`EXECUTOR`	`IntegrationProcedureExecutor`	Ejecutor de procedimientos de integración

Constructores

IssueProductHandler(String productName, IntegrationProcedureExecutor executor)

Descripción

Constructor que inicializa el handler con un nombre de producto específico y un ejecutor de procedimientos de integración personalizado.

Parámetros

- `productName` (String): Nombre del producto a gestionar.
- `executor` (IntegrationProcedureExecutor): Ejecutor personalizado para los procedimientos de integración.

Código

```
global IssueProductHandler(
  String productName,
  IntegrationProcedureExecutor executor
) {
  super(productName);
  this.EXECUTOR = executor;
}
```

IssueProductHandler(String productName)

Descripción

Constructor alternativo que inicializa el handler con un nombre de producto específico y un ejecutor mock por defecto.

Parámetros

- `productName` (String): Nombre del producto a gestionar.

Código

```
global IssueProductHandler(String productName) {
  super(productName);
  this.EXECUTOR = new MockIntegrationProcedureExecutor();
}
```

Métodos Públicos y Globales

onPre(Map<String, Object> options, Map<String, Object> input, Map<String, Object> output)

Descripción

Método virtual sobrescrito que se ejecuta antes del proceso principal. Realiza validaciones iniciales y asigna cuentas a la cotización.

Parámetros

- `options` (Map<String, Object>): Opciones adicionales para el proceso.
- `input` (Map<String, Object>): Datos de entrada para el proceso.
- `output` (Map<String, Object>): Mapa de salida donde se guardarán los resultados.

Retorno

- `Map<String, Object>`: Los datos de entrada, posiblemente modificados durante el pre-procesamiento.

Código

```
global virtual override Map<String, Object> onPre(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
) {
  validateInput(input, new List<String>{ 'leadId', 'quoteId' });
  assignAccountsToQuote(
    listInsuredPartyAccounts(input),
    (Id) input.get('quoteId')
```

```
  );
  return input;
}
```

assignAccountsToQuote(List<Map<String, Object>> parties, Id quoteId)

Descripción

Método virtual que asigna cuentas de partes aseguradas a una cotización.

Parámetros

- `parties` (List<Map<String, Object>>): Lista de partes aseguradas a asignar.
- `quoteId` (Id): ID de la cotización a la que se asignarán las cuentas.

Retorno

- `void`

Código

```
global virtual void assignAccountsToQuote(
  List<Map<String, Object>> parties,
  Id quoteId
) {
  SObject quote = QuoteUtils.getQuoteById(quoteId);
  new QuoteUtils().setInsuredPartyToQuoteAndOpp(parties, quote);
}
```

listInsuredPartyAccounts(Map<String, Object> input)

Descripción

Método virtual que obtiene la lista de cuentas de partes aseguradas a partir de los datos de entrada.

Parámetros

- `input` (Map<String, Object>): Datos de entrada que contienen información sobre las partes aseguradas.

Retorno

- `List<Map<String, Object>>`: Lista de mapas que representan las cuentas de partes aseguradas.

Código

```
global virtual List<Map<String, Object>> listInsuredPartyAccounts(
  Map<String, Object> input
) {
  return new LeadUtils().convertInsuredPartyFromLeadToAccount(input);
}
```

onPos(Map<String, Object> options, Map<String, Object> input, Map<String, Object> output)

Descripción

Método virtual sobrescrito que se ejecuta después del proceso principal. Actualiza el estado de la oportunidad a "Poliza emitida".

Parámetros

- `options` (Map<String, Object>): Opciones adicionales para el proceso.
- `input` (Map<String, Object>): Datos de entrada para el proceso.

- `output` (Map<String, Object>): Mapa de salida con los resultados del proceso principal.

Retorno

- `Map<String, Object>`: Los datos de entrada, posiblemente modificados durante el post-procesamiento.

Código

```
global virtual override Map<String, Object> onPos(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
) {
  String opportunityId = (String) ((Map<String, Object>) input
      ?.get('resultProcess'))
    ?.get('OpportunityId');

  if (opportunityId == null) {
    throw new ProductHandlerImplementationException(
      'OpportunityId is null',
      'UNPROCESSABLE_ENTITY',
      422
    );
  }
  Opportunity opp = [
    SELECT Id, StageName
    FROM Opportunity
    WHERE Id = :opportunityId
  ];

  opp.StageName = 'Poliza emitida';
  update opp;
  return input;
}
```

process(Map<String, Object> options, Map<String, Object> input, Map<String, Object> output)

Descripción

Método sobrescrito que orquesta el proceso principal de emisión de producto.

Parámetros

- `options` (Map<String, Object>): Opciones adicionales para el proceso.
- `input` (Map<String, Object>): Datos de entrada para el proceso.
- `output` (Map<String, Object>): Mapa de salida donde se guardarán los resultados.

Retorno

- `Map<String, Object>`: Mapa con los datos del producto emitido.

Código

```
global override Map<String, Object> process(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
) {
```

```
  if (options?.containsKey(OPTION_SKIP_BEHAVIOR)) {
    return input;
  }

  return executeIntegrationProcedure(this.preIssuingValidation(input));
}
```

executeIntegrationProcedure(Map<String, Object> ipInput)

Descripción

Ejecuta el procedimiento de integración para la emisión del producto.

Parámetros

- `ipInput` (Map<String, Object>): Datos de entrada para el procedimiento de integración.

Retorno

- `Map<String, Object>`: Mapa con los resultados del procedimiento de integración.

Código

```
global Map<String, Object> executeIntegrationProcedure(
  Map<String, Object> ipInput
) {
  Map<String, Object> output = new Map<String, Object>{
    'resultProcess' => EXECUTOR.execute(ISSUING_IPROCEDURE, ipInput)
  };
  this.setResponseProcess(output);
  return output;
}
```

Métodos Privados

validateInput(Map<String, Object> input, List<String> requiredFields)

Descripción

Valida que los campos requeridos estén presentes en el mapa de entrada.

Parámetros

- `input` (Map<String, Object>): Datos de entrada a validar.
- `requiredFields` (List
  ): Lista de nombres de campos requeridos.

Retorno

- `void`

Excepciones

- `ProductHandlerImplementationException`: Si algún campo requerido está ausente o vacío.

Código

```
private void validateInput(
  Map<String, Object> input,
  List<String> requiredFields
```

```
) {
  for (String field : requiredFields) {
    if (String.isBlank(String.valueOf(input?.get(field)))) {
      throw new ProductHandlerImplementationException(
        field + ERROR_REQUIRED_FIELD,
        'UNPROCESSABLE_ENTITY',
        422
      );
    }
  }
}
```

preIssuingValidation(Map<String, Object> input)

Descripción

Realiza validaciones específicas para el preprocesamiento de emisión, verificando la presencia de campos requeridos y las relaciones entre cotización, oportunidad y cuenta.

Parámetros

- `input` (Map<String, Object>): Datos de entrada a validar.

Retorno

- `Map<String, Object>`: Los datos de entrada validados.

Excepciones

- `ProductHandlerImplementationException`: Si no se encuentra el ID de oportunidad en la cotización o el ID de cuenta en la oportunidad.

Código

```
private Map<String, Object> preIssuingValidation(Map<String, Object> input) {
  validateInput(input, REQUIRED_FIELD_FOR_ISSUE_CORE);
  SObject quote = QuoteUtils.getQuoteById((Id) input.get('quoteId'));
  Id opportunityId = (Id) quote.get('OpportunityId');

  if (opportunityId == null) {
    throw new ProductHandlerImplementationException(
      'No OpportunityId found in Quote',
      'UNPROCESSABLE_ENTITY',
      422
    );
  }

  Opportunity opp = [
    SELECT Id, AccountId
    FROM Opportunity
    WHERE Id = :opportunityId
    LIMIT 1
  ];

  if (opp.AccountId == null) {
    throw new ProductHandlerImplementationException(
      'No AccountId found in Opportunity',
      'UNPROCESSABLE_ENTITY',
      422
```

```
    );
  }

  return input;
}
```

Excepciones

ProductHandlerImplementationException

Esta excepción personalizada se lanza en las siguientes situaciones:

1. Cuando un campo requerido está ausente o vacío.
2. Cuando no se encuentra el ID de oportunidad en la cotización.
3. Cuando no se encuentra el ID de cuenta en la oportunidad.
4. Cuando el ID de oportunidad es nulo en los resultados del proceso.

Dependencias

- `ProductHandler`: Clase base que `IssueProductHandler` extiende.
- `IntegrationProcedureExecutor`: Interfaz o clase que define el método `execute` para ejecutar procedimientos de integración.
- `MockIntegrationProcedureExecutor`: Implementación mock de `IntegrationProcedureExecutor` utilizada en el constructor alternativo.
- `QuoteUtils`: Clase utilitaria para operaciones con cotizaciones.
- `LeadUtils`: Clase utilitaria para operaciones con leads.
- `ProductHandlerImplementationException`: Excepción personalizada lanzada cuando hay errores en el proceso.

Flujo de Proceso

El proceso de emisión de producto sigue el siguiente flujo:

1. **Pre-procesamiento (`onPre`)**:
   - Validación de campos requeridos ('leadId', 'quoteId').
   - Obtención de la lista de cuentas de partes aseguradas.
   - Asignación de cuentas a la cotización.
2. **Proceso principal (`process`)**:
   - Validación específica para emisión (verificación de campos y relaciones).
   - Ejecución del procedimiento de integración para emisión.
3. **Post-procesamiento (`onPos`)**:
   - Actualización del estado de la oportunidad a "Poliza emitida".

Ejemplo de Uso

```
// Crear el ejecutor de procedimientos de integración
IntegrationProcedureExecutor executor = new SomeIntegrationProcedureExecutor();
```

```
// Instanciar el handler
IssueProductHandler handler = new IssueProductHandler('AutoInsurance', executor);

// Preparar los datos de entrada
Map<String, Object> input = new Map<String, Object>{
  'leadId' => '00Q0x000000XXXXX',
  'quoteId' => '0Q00x000000XXXXX',
  'effectiveDate' => Date.today(),
  'producerId' => '0010x000000XXXXX'
};

// Mapas para opciones y salida
Map<String, Object> options = new Map<String, Object>();
Map<String, Object> output = new Map<String, Object>();

try {
  // Iniciar el flujo de emisión
  Map<String, Object> result = handler.onPre(options, input, output);
  result = handler.process(options, result, output);
  result = handler.onPos(options, result, output);

  // Verificar el resultado
  System.debug('Emisión completada. Resultado: ' + result);
} catch (ProductHandlerImplementationException e) {
  System.debug('Error en el proceso de emisión: ' + e.getMessage());
}
```

Extensión de la Clase

```
public class SpecializedIssueHandler extends IssueProductHandler {

    public SpecializedIssueHandler(String productName) {
        super(productName);
    }

    // Personalizar el comportamiento previo a la emisión
    global override Map<String, Object> onPre(
        Map<String, Object> options,
        Map<String, Object> input,
        Map<String, Object> output
    ) {
        // Realizar validaciones adicionales específicas del producto
        if (!String.valueOf(input.get('productLine')).equals('SpecialLine')) {
            throw new ProductHandlerImplementationException(
                'Este handler solo maneja la línea de producto "SpecialLine"',
                'UNPROCESSABLE_ENTITY',
                422
            );
        }

        // Llamar al método base
        return super.onPre(options, input, output);
    }

    // Personalizar el comportamiento posterior a la emisión
```

```
    global override Map<String, Object> onPos(
        Map<String, Object> options,
        Map<String, Object> input,
        Map<String, Object> output
    ) {
        // Llamar al método base primero
        Map<String, Object> result = super.onPos(options, input, output);

        // Realizar acciones adicionales después de la emisión
        String opportunityId = (String) ((Map<String, Object>)
input.get('resultProcess')).get('OpportunityId');
        if (opportunityId != null) {
            // Crear una tarea de seguimiento, por ejemplo
            Task followUpTask = new Task(
                WhatId = opportunityId,
                Subject = 'Seguimiento post-emisión',
                ActivityDate = Date.today().addDays(7),
                Status = 'Pendiente',
                Priority = 'Normal'
            );
            insert followUpTask;
        }

        return result;
    }
}
```

Notas Adicionales

1. La clase utiliza el operador de navegación segura (`?.`) para evitar excepciones de referencia nula al acceder a valores en mapas.
2. La clase implementa un mecanismo para omitir su comportamiento predeterminado si la opción `skipDefaultBehavior` está presente.
3. El método `executeIntegrationProcedure` encapsula la ejecución del procedimiento de integración, facilitando la prueba unitaria y la extensibilidad.
4. La clase realiza validaciones exhaustivas en diferentes etapas del proceso para garantizar la integridad de los datos.
5. La estructura de la clase facilita la personalización mediante herencia, permitiendo sobrescribir los métodos virtuales para comportamientos específicos.

LeadConfigurator

Descripción General

`LeadConfigurator` es una interfaz global que define un contrato para clases que configuran objetos Lead en Salesforce. Esta interfaz forma parte del patrón de diseño Strategy, permitiendo encapsular diferentes algoritmos de configuración para Leads que pueden ser utilizados intercambiablemente.

Autor

Jean Carlos Melendez

Última Modificación

30 de enero de 2025

Definición de la Interfaz

```
global interface LeadConfigurator {
  void configure(Lead lead);
}
```

Métodos

configure(Lead lead)

Descripción

Método que debe ser implementado por las clases que implementan esta interfaz. Su propósito es aplicar una configuración específica a un objeto Lead.

Parámetros

- `lead` (Lead): El objeto Lead que será configurado.

Retorno

- `void`: Este método no devuelve ningún valor.

Propósito y Aplicaciones

El propósito principal de esta interfaz es proporcionar un mecanismo flexible para configurar objetos Lead de diferentes maneras sin modificar el código que utiliza estas configuraciones. Esto es particularmente útil en escenarios como:

1. **Configuración de Leads por canal**: Aplicar configuraciones específicas según el canal de origen del Lead (web, teléfono, evento, etc.).
2. **Configuración por campañas**: Configurar Leads con información relacionada a campañas específicas.
3. **Configuración por producto de interés**: Aplicar lógica personalizada según el producto o servicio de interés del Lead.
4. **Normalización de datos**: Implementar diferentes estrategias para limpiar y normalizar datos de Leads.
5. **Asignación de equipos o territorios**: Configurar asignaciones basadas en reglas personalizadas.

Patrones de Diseño Relacionados

Strategy

La interfaz `LeadConfigurator` es un ejemplo clásico del patrón de diseño Strategy, que permite definir una familia de algoritmos (en este caso, algoritmos de configuración de Leads), encapsularlos y hacerlos intercambiables. Este patrón permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

Composite

Cuando se utilizan múltiples implementaciones de `LeadConfigurator` juntas para configurar un Lead, se está empleando indirectamente el patrón Composite, permitiendo tratar tanto configuraciones individuales como grupos de configuraciones de manera uniforme.

Decorator

Las implementaciones de esta interfaz pueden actuar como decoradores, añadiendo funcionalidades adicionales o modificando propiedades existentes de un objeto Lead sin cambiar su estructura básica.

Relación con LeadBuilderFoundation

Esta interfaz está diseñada para trabajar en conjunto con la clase `LeadBuilderFoundation`, que implementa el patrón Builder para construir objetos Lead. La clase `LeadBuilderFoundation` tiene un método `addConfigurators` que acepta una lista de objetos que implementan esta interfaz, y luego los aplica secuencialmente en su método `build`.

```
// Fragmento de código de LeadBuilderFoundation
global Lead build() {
  if (!this.configurators.isEmpty()) {
    for (LeadConfigurator configurator : configurators) {
      configurator.configure(lead);
    }
  }
  return lead;
}
```

Ejemplos de Implementación

Configurador de Fuente de Lead

```
public class LeadSourceConfigurator implements LeadConfigurator {
    private String source;
    private String sourceDetail;

    public LeadSourceConfigurator(String source, String sourceDetail) {
        this.source = source;
        this.sourceDetail = sourceDetail;
    }

    public void configure(Lead lead) {
        lead.LeadSource = this.source;
        lead.Source_Detail__c = this.sourceDetail;
    }
}
```

Configurador de Datos de Contacto

```
public class ContactInfoConfigurator implements LeadConfigurator {
    private String email;
    private String phone;
    private String country;

    public ContactInfoConfigurator(String email, String phone, String country) {
```

```
        this.email = email;
        this.phone = phone;
        this.country = country;
    }

    public void configure(Lead lead) {
        lead.Email = this.email;
        lead.Phone = this.phone;
        lead.Country = this.country;

        // Normalizar el número de teléfono según el país
        if (this.country == 'USA' || this.country == 'US') {
            lead.Phone = normalizeUSPhone(this.phone);
        }
    }

    private String normalizeUSPhone(String phone) {
        // Implementación de normalización de número telefónico de EE.UU.
        // ...
        return phone;
    }
}
```

Configurador para Asignación de Propietario

```
public class OwnerAssignmentConfigurator implements LeadConfigurator {
    private String productInterest;

    public OwnerAssignmentConfigurator(String productInterest) {
        this.productInterest = productInterest;
    }

    public void configure(Lead lead) {
        lead.Product_Interest__c = this.productInterest;

        // Asignar propietario basado en el producto de interés
        Id ownerId = getOwnerIdForProduct(this.productInterest);
        if (ownerId != null) {
            lead.OwnerId = ownerId;
        }
    }

    private Id getOwnerIdForProduct(String product) {
        // Lógica para determinar el propietario adecuado basado en el producto
        // Podría consultar una configuración personalizada, una jerarquía de roles, etc.
        // ...
        return null; // Devolver el ID del propietario apropiado
    }
}
```

Configurador Compuesto

```
public class CompositeLeadConfigurator implements LeadConfigurator {
    private List<LeadConfigurator> configurators;

    public CompositeLeadConfigurator() {
```

```
        this.configurators = new List<LeadConfigurator>();
    }

    public CompositeLeadConfigurator addConfigurator(LeadConfigurator configurator) {
        this.configurators.add(configurator);
        return this;
    }

    public void configure(Lead lead) {
        for (LeadConfigurator configurator : configurators) {
            configurator.configure(lead);
        }
    }
}
```

Ejemplo de Uso con LeadBuilderFoundation

```
// Crear configuradores específicos
LeadSourceConfigurator sourceConfig = new LeadSourceConfigurator('Web', 'Landing Page');
ContactInfoConfigurator contactConfig = new ContactInfoConfigurator('john.doe@example.com', '555-1234', 'US');
OwnerAssignmentConfigurator ownerConfig = new OwnerAssignmentConfigurator('Cloud Services');

// Añadir configuradores al builder
List<LeadConfigurator> configurators = new List<LeadConfigurator>{
    sourceConfig,
    contactConfig,
    ownerConfig
};

// Crear y configurar un Lead usando el builder
Lead newLead = new LeadBuilderFoundation()
    .setFirstName('John')
    .setLastName('Doe')
    .addConfigurators(configurators)
    .build();

// Alternativamente, crear y configurar y insertar un Lead
Lead insertedLead = new LeadBuilderFoundation()
    .setFirstName('Jane')
    .setLastName('Smith')
    .addConfigurators(configurators)
    .buildAndInsert();
```

Ejemplo Avanzado con Lógica Personalizada

```
public class CampaignResponderConfigurator implements LeadConfigurator {
    private Id campaignId;
    private Boolean attended;
    private Date responseDate;

    public CampaignResponderConfigurator(Id campaignId, Boolean attended, Date responseDate) {
        this.campaignId = campaignId;
        this.attended = attended;
        this.responseDate = responseDate;
    }
```

```
    public void configure(Lead lead) {
        // Configurar campos relacionados con la campaña
        lead.Campaign_Source__c = this.campaignId;

        // Después de insertar el Lead, crear un CampaignMember
        if (lead.Id != null) {
            try {
                CampaignMember member = new CampaignMember(
                    CampaignId = this.campaignId,
                    LeadId = lead.Id,
                    Status = this.attended ? 'Attended' : 'Responded',
                    HasResponded = true
                );

                if (this.responseDate != null) {
                    member.FirstRespondedDate = this.responseDate;
                }

                insert member;
            } catch (Exception e) {
                // Manejar errores o registrar excepciones
                System.debug('Error al crear CampaignMember: ' + e.getMessage());
            }
        }
    }
}
```

Consideraciones para Testing

Al probar clases que implementan la interfaz `LeadConfigurator`, es importante:

1. **Verificar campos individuales**: Comprobar que cada campo afectado por el configurador tenga el valor esperado.
2. **Probar combinaciones**: Verificar que múltiples configuradores funcionen correctamente juntos sin conflictos.
3. **Simular dependencias**: Utilizar mocks o datos de prueba para cualquier consulta o lógica externa.
4. **Verificar efectos secundarios**: Si el configurador realiza DML u otras operaciones, asegurarse de que se ejecuten correctamente.

Ejemplo de una clase de prueba:

```
@isTest
private class LeadSourceConfiguratorTest {
    @isTest
    static void testConfigureLead() {
        // Preparar datos
        String source = 'Web';
        String detail = 'Social Media';
        LeadSourceConfigurator configurator = new LeadSourceConfigurator(source, detail);

        Lead testLead = new Lead(
            FirstName = 'Test',
            LastName = 'User'
        );

        // Ejecutar la configuración
        Test.startTest();
```

```
        configurator.configure(testLead);
        Test.stopTest();

        // Verificar resultados
        System.assertEquals(source, testLead.LeadSource, 'Lead source should be set correctly');
        System.assertEquals(detail, testLead.Source_Detail__c, 'Source detail should be set correctly');
    }
}
```

Mejores Prácticas

1. **Principio de Responsabilidad Única**: Cada implementación de `LeadConfigurator` debe tener una responsabilidad única y específica.
2. **Inmutabilidad**: Las implementaciones deberían ser inmutables cuando sea posible, recibiendo todos los datos necesarios en el constructor.
3. **Manejo de errores robusto**: Implementar manejo de excepciones adecuado para evitar fallos en cascada.
4. **Documentación clara**: Documentar qué campos modifica cada configurador para facilitar su uso y mantenimiento.
5. **Evitar efectos secundarios**: Minimizar las operaciones DML u otros efectos secundarios dentro de los configuradores.
6. **Parametrización**: Diseñar configuradores parametrizados en lugar de codificar valores fijos.

Notas Adicionales

1. La naturaleza global de la interfaz sugiere que está diseñada para ser accesible desde diferentes paquetes o namespaces.
2. Esta interfaz facilita la aplicación del principio abierto/cerrado, permitiendo extender la funcionalidad sin modificar el código existente.
3. Al ser parte de un sistema más amplio que incluye `LeadBuilderFoundation`, esta interfaz contribuye a una arquitectura más modular y mantenible para la gestión de Leads.

LeadBuilderFoundation

Descripción General

`LeadBuilderFoundation` es una clase global que implementa el patrón de diseño Builder para crear y configurar objetos Lead de manera fluida en Salesforce. Esta clase permite encadenar métodos para establecer diferentes atributos del Lead y aplicar configuradores personalizados antes de construir e insertar el registro.

Autor

Felipe Correa \<felipe.correa@nespon.com\>

Última Modificación

30 de enero de 2025 por Jean Carlos Melendez

Atributos

lead

- **Tipo**: `Lead`
- **Accesibilidad**: `global`
- **Descripción**: El objeto Lead que se está construyendo.

configurators

- **Tipo**: `List<LeadConfigurator>`
- **Accesibilidad**: `global`
- **Descripción**: Lista de configuradores que aplicarán personalizaciones adicionales al Lead.

Constructor

LeadBuilderFoundation()

Descripción

Constructor que inicializa un nuevo objeto Lead vacío.

Parámetros

Ninguno.

Código

```
global LeadBuilderFoundation() {
  lead = new Lead();
}
```

Métodos de Configuración Fluida

Estos métodos permiten establecer valores para campos específicos del Lead y devuelven la instancia del builder, permitiendo el encadenamiento de métodos.

setId(Id id)

Descripción

Establece el Id del Lead.

Parámetros

- `id` (Id): Id del Lead.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation setId(Id id) {
  lead.Id = id;
  return this;
}
```

setStatus(String status)

Descripción

Establece el estado del Lead.

Parámetros

- `status` (String): Estado del Lead.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation setStatus(String status) {
  lead.Status = status;
  return this;
}
```

setLastName(String lastName)

Descripción

Establece el apellido del Lead.

Parámetros

- `lastName` (String): Apellido del Lead.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation setLastName(String lastName) {
  lead.LastName = lastName;
  return this;
}
```

setFirstName(String firstName)

Descripción

Establece el nombre del Lead.

Parámetros

- `firstName` (String): Nombre del Lead.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation setFirstName(String firstName) {
  lead.FirstName = firstName;
  return this;
}
```

setEmail(String email)

Descripción

Establece el correo electrónico del Lead.

Parámetros

- `email` (String): Correo electrónico del Lead.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation setEmail(String email) {
  lead.Email = email;
  return this;
}
```

setMobile(String mobile)

Descripción

Establece el número de teléfono móvil del Lead.

Parámetros

- `mobile` (String): Número de teléfono móvil del Lead.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation setMobile(String mobile) {
  lead.MobilePhone = mobile;
  return this;
}
```

setPhone(String phone)

Descripción

Establece el número de teléfono del Lead.

Parámetros

- `phone` (String): Número de teléfono del Lead.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation setPhone(String phone) {
  lead.Phone = phone;
  return this;
}
```

addConfigurators(List<LeadConfigurator> configuratorsList)

Descripción

Añade una lista de configuradores para personalizar el Lead.

Parámetros

- `configuratorsList` (List

  ): Lista de configuradores a añadir.

Retorno

- `LeadBuilderFoundation`: La instancia actual del builder.

Código

```
global LeadBuilderFoundation addConfigurators(
  List<LeadConfigurator> configuratorsList
) {
  if (!configuratorsList.isEmpty()) {
    this.configurators.addAll(configuratorsList);
  }
  return this;
}
```

Métodos de Construcción

build()

Descripción

Construye el objeto Lead aplicando todos los configuradores añadidos.

Retorno

- `Lead`: El objeto Lead configurado.

Proceso

1. Verifica si hay configuradores en la lista.
2. Si existen, aplica cada configurador al objeto Lead.

3. Devuelve el objeto Lead configurado.

Código

```
global Lead build() {
  if (!this.configurators.isEmpty()) {
    for (LeadConfigurator configurator : configurators) {
      configurator.configure(lead);
    }
  }
  return lead;
}
```

buildAndInsert()

Descripción

Construye el objeto Lead aplicando todos los configuradores y lo inserta en la base de datos.

Retorno

- `Lead`: El objeto Lead configurado e insertado.

Proceso

1. Verifica si hay configuradores en la lista.
2. Si existen, aplica cada configurador al objeto Lead.
3. Realiza un upsert del Lead en la base de datos.
4. Devuelve el objeto Lead insertado.

Código

```
global Lead buildAndInsert() {
  if (!this.configurators.isEmpty()) {
    for (LeadConfigurator configurator : configurators) {
      configurator.configure(lead);
    }
  }
  upsert lead;
  return lead;
}
```

Métodos Estáticos de Servicio

createLead(Lead lead)

Descripción

Método estático que inserta un Lead utilizando reglas de asignación.

Parámetros

- `lead` (Lead): El objeto Lead a insertar.

Retorno

- `Lead`: El objeto Lead insertado.

Código

```
global static Lead createLead(Lead lead) {
  return insertLeadWithAssignmentRules(lead);
}
```

upsertLead(Lead lead)

Descripción

Método estático que realiza un upsert de un Lead utilizando reglas de asignación.

Parámetros

- `lead` (Lead): El objeto Lead para realizar upsert.

Retorno

- `Lead`: El objeto Lead después del upsert.

Proceso

1. Configura las opciones DML para utilizar la regla de asignación predeterminada.
2. Establece estas opciones en el objeto Lead.
3. Realiza un upsert del Lead en la base de datos.
4. Devuelve el objeto Lead actualizado.

Código

```
global static Lead upsertLead(Lead lead) {
  Database.DMLOptions dmlOptn = new Database.DMLOptions();
  dmlOptn.assignmentRuleHeader.useDefaultRule = true;
  lead.setOptions(dmlOptn);
  Database.upsert(lead);
  return lead;
}
```

Métodos Privados

insertLeadWithAssignmentRules(Lead lead)

Descripción

Método auxiliar privado que inserta un Lead utilizando reglas de asignación predeterminadas.

Parámetros

- `lead` (Lead): El objeto Lead a insertar.

Retorno

- `Lead`: El objeto Lead insertado.

Proceso

1. Configura las opciones DML para utilizar la regla de asignación predeterminada.
2. Establece estas opciones en el objeto Lead.
3. Realiza un upsert del Lead en la base de datos.

4. Devuelve el objeto Lead insertado.

Código

```
private static Lead insertLeadWithAssignmentRules(Lead lead) {
  Database.DMLOptions dmlOptn = new Database.DMLOptions();
  dmlOptn.assignmentRuleHeader.useDefaultRule = true;
  lead.setOptions(dmlOptn);
  Database.upsert(lead);
  return lead;
}
```

Dependencias

- `Lead`: Objeto estándar de Salesforce para almacenar prospectos.
- `LeadConfigurator`: Interfaz o clase que define el método `configure(Lead)` para personalizar Leads.
- `Database.DMLOptions`: Clase de Salesforce para configurar opciones de operaciones DML.

Patrón de Diseño Implementado

Builder

La clase implementa el patrón de diseño Builder, que permite construir objetos complejos paso a paso. Las principales características de este patrón en la implementación son:

1. **Métodos de configuración fluida**: Todos los métodos `set*` devuelven la instancia del builder, permitiendo el encadenamiento de métodos.
2. **Separación de construcción y representación**: El proceso de construcción está separado de la representación final del objeto Lead.
3. **Proceso paso a paso**: El objeto Lead se construye gradualmente añadiendo propiedades específicas.
4. **Configuración extensible**: El método `addConfigurators` permite añadir comportamientos personalizados a través de implementaciones de `LeadConfigurator`.

Ejemplo de Uso

Uso Básico

```
// Crear un Lead con el Builder
Lead newLead = new LeadBuilderFoundation()
    .setFirstName('John')
    .setLastName('Doe')
    .setEmail('john.doe@example.com')
    .setPhone('555-1234')
    .setMobile('555-5678')
    .setStatus('Open')
    .build();

// Insertar el Lead creado usando el servicio estático
Lead insertedLead = LeadBuilderFoundation.createLead(newLead);
```

Uso con Configuradores Personalizados

```
// Definir un configurador personalizado
public class CompanyNameConfigurator implements LeadConfigurator {
```

```
    private String companyName;

    public CompanyNameConfigurator(String companyName) {
        this.companyName = companyName;
    }

    public void configure(Lead lead) {
        lead.Company = this.companyName;
    }
}

// Definir otro configurador
public class LeadSourceConfigurator implements LeadConfigurator {
    private String source;

    public LeadSourceConfigurator(String source) {
        this.source = source;
    }

    public void configure(Lead lead) {
        lead.LeadSource = this.source;
    }
}

// Crear y usar una lista de configuradores
List<LeadConfigurator> configurators = new List<LeadConfigurator>{
    new CompanyNameConfigurator('ACME Inc.'),
    new LeadSourceConfigurator('Web')
};

// Crear y insertar un Lead con configuradores personalizados
Lead lead = new LeadBuilderFoundation()
    .setFirstName('Jane')
    .setLastName('Smith')
    .setEmail('jane.smith@example.com')
    .addConfigurators(configurators)
    .buildAndInsert();
```

Uso con ID Existente (Actualización)

```
// Actualizar un Lead existente
Id existingLeadId = '00Q0x000000XXXXX';
Lead updatedLead = new LeadBuilderFoundation()
    .setId(existingLeadId)
    .setStatus('Qualified')
    .build();

// Realizar upsert del Lead
Lead result = LeadBuilderFoundation.upsertLead(updatedLead);
```

Notas Adicionales

1. La clase utiliza el método `upsert` para permitir tanto la inserción como la actualización de registros de Lead, dependiendo de si se proporciona un Id.
2. Los configuradores personalizados (`LeadConfigurator`) permiten extender la funcionalidad del builder sin modificar su código base.

3. El uso de reglas de asignación asegura que los Leads se asignen correctamente según las reglas configuradas en Salesforce.
4. El patrón builder implementado mejora la legibilidad del código y simplifica la creación de objetos Lead con múltiples atributos.
5. La clase está marcada como `global`, lo que permite su uso en paquetes gestionados y su acceso desde otros namespaces.
6. Hay una duplicación de código entre `insertLeadWithAssignmentRules` y `upsertLead` que podría refactorizarse para mejorar la mantenibilidad.

LeadUtils

Esta clase proporciona utilidades para la gestión y manipulación de objetos Lead, facilitando la búsqueda, recuperación y conversión de datos relacionados con leads.

📋 Información General

Atributo	Valor
Nombre API	`LeadUtils`
Contexto	`global virtual`
Grupo	`utils`
Desarrollador original	Jean Carlos Melendez
Última modificación	30-01-2025 (Jean Carlos Melendez)

🧩 Principales características

Esta clase proporciona métodos utilitarios para:

- Búsqueda de leads por criterios específicos (email y teléfono móvil)
- Recuperación de leads por ID
- Conversión de leads a cuentas para partes aseguradas

🔄 Métodos principales

🔍 searchLeadIdByEmailAndMobilePhone(String email, String mobilePhone)

```
global static Id searchLeadIdByEmailAndMobilePhone(String email, String mobilePhone)
```

- **Descripción**: Busca un lead por su correo electrónico y número de teléfono móvil.
- **Parámetros**:
  - `email` (String): Correo electrónico del lead.
  - `mobilePhone` (String): Número de teléfono móvil del lead.
- **Retorna**:
  - `Id`: ID del lead encontrado, o null si no se encuentra ningún lead.
- **Comportamiento**:
  1. Realiza una consulta SOQL para buscar leads que coincidan con el correo electrónico y teléfono móvil proporcionados.
  2. Si encuentra al menos un lead, devuelve el ID del primer lead encontrado.
  3. Si no encuentra ningún lead, devuelve null.
  4. Registra en el log de depuración el ID del lead encontrado.

getLeadById(Id leadId)

```
global static Lead getLeadById(Id leadId)
```

- **Descripción**: Recupera un objeto Lead por su ID.
- **Parámetros**:
  - `leadId` (Id): ID del lead a recuperar.
- **Retorna**:
  - `Lead`: Objeto Lead correspondiente al ID proporcionado.
- **Comportamiento**:
  1. Realiza una consulta SOQL para buscar el lead con el ID proporcionado.
  2. Devuelve el objeto Lead encontrado.
  3. La consulta está limitada a 1 registro para optimizar el rendimiento.

convertInsuredPartyFromLeadToAccount(Map<String, Object> input)

```
global virtual List<Map<String, Object>> convertInsuredPartyFromLeadToAccount(Map<String, Object> input)
```

- **Descripción**: Convierte un lead en una estructura de datos para una parte asegurada (cuenta).
- **Parámetros**:
  - `input` (Map<String, Object>): Mapa con los datos de entrada, debe contener una clave 'leadId' con el ID del lead a convertir.
- **Retorna**:
  - `List<Map<String, Object>>`: Lista de mapas con la estructura de datos para las partes aseguradas.
- **Comportamiento**:
  1. Extrae el ID del lead del mapa de entrada.
  2. Consulta los datos del lead, incluyendo nombre, apellido, correo electrónico, teléfono móvil, ciudad y calle.
  3. Crea una estructura de datos para la parte asegurada utilizando el patrón Builder (AccountBuilder).
  4. Registra en el log de depuración la estructura de datos creada.
  5. Devuelve la lista de partes aseguradas.
- **Notas**:
  - Este método es virtual, por lo que puede ser sobrescrito en clases que extiendan LeadUtils.
  - Utiliza el patrón Builder para la construcción del objeto Account.

Ejemplo práctico

```
// Búsqueda de un lead por correo electrónico y teléfono
Id leadId = LeadUtils.searchLeadIdByEmailAndMobilePhone('ejemplo@email.com', '5551234567');
if (leadId != null) {
    // Recuperación del lead completo
    Lead leadEncontrado = LeadUtils.getLeadById(leadId);

    // Conversión del lead a estructura de asegurado
```

```
    Map<String, Object> entrada = new Map<String, Object>{'leadId' => leadId};
    List<Map<String, Object>> partesAseguradas = new LeadUtils().convertInsuredPartyFromLeadToAccount(entrada);

    // Procesar las partes aseguradas
    for (Map<String, Object> parte : partesAseguradas) {
        Account cuentaAsegurado = (Account)parte.get('Asegurado');
        // Hacer algo con la cuenta...
    }
}
```

🪀 Dependencias

- **AccountBuilder**: Patrón Builder utilizado para construir objetos Account a partir de datos de Lead.
- **Objeto Lead**: Interactúa directamente con el objeto estándar Lead de Salesforce.

💉 Consideraciones técnicas

- Los métodos `searchLeadIdByEmailAndMobilePhone` y `getLeadById` son estáticos, por lo que pueden ser llamados directamente desde la clase sin necesidad de instanciarla.
- El método `convertInsuredPartyFromLeadToAccount` es virtual y no estático, lo que permite sobrescribirlo en clases que extiendan LeadUtils.
- Las consultas SOQL están optimizadas con límites para evitar problemas de rendimiento.
- La clase utiliza el patrón Builder para la construcción de objetos Account, lo que facilita la creación de objetos complejos.

LeadProductHandler

Este handler gestiona el procesamiento de leads para diferentes productos, aplicando la lógica de negocio específica y preparando los datos para su procesamiento posterior.

📋 Información General

Atributo	Valor
Nombre API	`LeadProductHandler`
Extiende	`ProductHandler`
Contexto	`global virtual`
Proceso	`LEAD`
Desarrollador original	Soulberto Lorenzo
Última modificación	25-04-2025 (Jean Carlos Melendez)

🧩 Principales características

📦 Constantes

- **PROCESS_NAME**: Define el nombre del proceso como "LEAD".

🔄 Métodos principales

👷 Constructor

```
global LeadProductHandler(String productName)
```

- **Descripción**: Inicializa la instancia del handler con el nombre del producto.
- **Parámetros**:
  - `productName` (String): Nombre del producto asociado al lead.

👁️ setObservers()

```
public override void setObservers()
```

- **Descripción**: Configura los observadores para este handler.
- **Implementación**: Añade un `LeadObserver` a la lista de observadores.

🏷️ setProcessName(String processName)

```
global virtual override void setProcessName(String processName)
```

- **Descripción**: Establece el nombre del proceso.
- **Parámetros**:
  - `processName` (String): Nombre del proceso a establecer.

🕹️ setResponseProcess(Object responseProcess)

```
global virtual override void setResponseProcess(Object responseProcess)
```

- **Descripción**: Establece la respuesta del proceso.
- **Parámetros**:
  - `responseProcess` (Object): Objeto con la respuesta del proceso.

🕹️ setRequestInput(Object requestInput)

```
global virtual override void setRequestInput(Object requestInput)
```

- **Descripción**: Establece los datos de entrada para el proceso.
- **Parámetros**:
  - `requestInput` (Object): Objeto con los datos de entrada.

⚙️ process(Map<String, Object> options, Map<String, Object> input, Map<String, Object> output)

```
public override Map<String, Object> process(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
)
```

- **Descripción**: Método principal que ejecuta el procesamiento del lead.
- **Parámetros**:
  - `options` (Map<String, Object>): Opciones de configuración para el proceso.
  - `input` (Map<String, Object>): Datos de entrada para el proceso.
  - `output` (Map<String, Object>): Mapa donde se almacena el resultado.
- **Retorna**:
  - Map<String, Object>: El mapa de salida con el resultado del proceso.
- **Comportamiento**:
  1. Verifica si debe omitirse el comportamiento predeterminado.
  2. Establece el nombre del proceso.
  3. Obtiene los configuradores de lead del mapa de entrada.
  4. Crea un nuevo lead utilizando el patrón Builder.
  5. Registra la finalización del proceso y guarda el resultado.
  6. Maneja excepciones DML y generales, lanzando excepciones personalizadas.

✅ isValidForLeadRequirements(Map<String, Object> data)

```
global virtual Boolean isValidForLeadRequirements(Map<String, Object> data)
```

- **Descripción**: Valida que los datos cumplan con los requisitos de negocio para la creación de un lead.
- **Parámetros**:
  - `data` (Map<String, Object>): Datos a validar.
- **Retorna**:
  - Boolean: `true` si los datos son válidos, `false` en caso contrario.
- **Notas**: Este método está diseñado para ser sobrescrito en implementaciones específicas.

🔄 Flujo de ejecución

1. Se instancia el handler con el nombre del producto.
2. Se configuran los observadores llamando a `setObservers()`.
3. Se invoca el método `process()` con los mapas correspondientes.
4. Se validan los datos y se crea el lead utilizando el builder.
5. Se registra el resultado y se devuelve en el mapa de salida.

🧠 Puntos de extensión

Este handler puede ser extendido de las siguientes maneras:

1. **Sobrescribir el método `isValidForLeadRequirements()`**: Para implementar validaciones específicas de negocio.
2. **Crear un handler que extienda `LeadProductHandler`**: Para personalizar comportamientos específicos por producto.
3. **Agregar configuradores adicionales**: Mediante la lista `configurators` para modificar la creación del lead.

📊 Ejemplo práctico

```
// Creación de un handler para leads de producto Viajes
LeadProductHandler viajesHandler = new LeadProductHandler('Viajes');

// Configuración de los datos de entrada
Map<String, Object> input = new Map<String, Object>{
  'lastName' => 'Pérez',
  'firstName' => 'Juan',
  'email' => 'juan.perez@example.com',
  'mobile_phone' => '3124567890',
  'status' => '0_Abierto'
};

// Procesamiento del lead
Map<String, Object> output = new Map<String, Object>();
Map<String, Object> options = new Map<String, Object>();
output = viajesHandler.process(options, input, output);
```

```
// Obtención del resultado
Lead createdLead = (Lead)output.get('resultProcess');
```

🧪 Consideraciones técnicas

- Utiliza el patrón Observer para notificar sobre eventos durante el procesamiento.
- Implementa el patrón Builder para la construcción del objeto Lead.
- Utiliza manejo de excepciones personalizadas para informar errores específicos.
- Los errores se devuelven con un código HTTP 422 (UNPROCESSABLE_ENTITY).

🔍 Log y trazabilidad

El proceso registra información en el objeto `Subconjunto_Registro_de_Evento__c` para permitir la trazabilidad completa, incluyendo:

- Datos de entrada
- Respuesta del proceso
- Detalles de la ejecución
- Usuario responsable

RUNTWrapper

Descripción General

`RUNTWrapper` es una clase diseñada para manejar y estructurar los datos provenientes del Registro Único Nacional de Tránsito (RUNT) de Colombia. Esta clase encapsula información detallada sobre vehículos, sus propietarios y características técnicas.

Autor

- **Autor:** felipe.correa@nespon.com
- **Proyecto:** Foundation Salesforce - Suratech - UH-17585
- **Última modificación:** 11-06-2024

Anotaciones

- `@NamespaceAccessible`: Permite que la clase sea accesible desde otros namespaces, facilitando su uso en paquetes gestionados o en integraciones.

Estructura de la Clase

Propiedades Principales

La clase contiene numerosas propiedades que representan los diferentes atributos de un vehículo registrado en el RUNT:

Propiedad	Descripción
codigoResultado	Código del resultado de la consulta
fecha	Fecha de la consulta o registro
idUsuario	Identificador del usuario
noRegistro	Número de registro
noLicenciaTransito	Número de licencia de tránsito
fechaExpedicionLicTransito	Fecha de expedición de la licencia de tránsito
estadoDelVehiculo	Estado actual del vehículo
idTipoServicio	ID del tipo de servicio del vehículo
tipoServicio	Descripción del tipo de servicio
idClaseVehiculo	ID de la clase del vehículo
claseVehiculo	Descripción de la clase del vehículo
idMarca	ID de la marca del vehículo
marca	Nombre de la marca del vehículo
idLinea	ID de la línea del vehículo
linea	Descripción de la línea del vehículo
modelo	Año modelo del vehículo
idColor	ID del color del vehículo
color	Descripción del color del vehículo
noMotor	Número de motor
noChasis	Número de chasis
noVin	Número VIN (Vehicle Identification Number)
cilindraje	Cilindraje del vehículo
divipola	Código de División Político-Administrativa
idTipoCarroceria	ID del tipo de carrocería

tipoCarroceria	Descripción del tipo de carrocería
fechaMatricula	Fecha de matrícula del vehículo
tieneGravamenes	Indica si el vehículo tiene gravámenes
organismoTransito	Organismo de tránsito donde está registrado
prendas	Información sobre prendas asociadas al vehículo
prendario	Información adicional sobre estado prendario
clasificacion	Clasificación del vehículo
esRegrabadoMotor	Indica si el motor ha sido regrabado
esRegrabadoChasis	Indica si el chasis ha sido regrabado
esRegrabadoSerie	Indica si la serie ha sido regrabada
esRegrabadoVin	Indica si el VIN ha sido regrabado
tipoServicioNombre	Nombre del tipo de servicio
nroMotor	Número de motor (duplicado de noMotor)
capacidadCarga	Capacidad de carga del vehículo
pesoBrutoVehicular	Peso bruto vehicular
noEjes	Número de ejes del vehículo
noPlaca	Número de placa del vehículo
repotenciado	Indica si el vehículo ha sido repotenciado
diasMatriculado	Días transcurridos desde la matrícula
idPaisOrigen	ID del país de origen del vehículo
paisOrigen	Nombre del país de origen
capacidadPasajerosSentados	Capacidad de pasajeros sentados
idTipoImportacion	ID del tipo de importación
tipoImportacion	Descripción del tipo de importación
idTipoCombustible	ID del tipo de combustible

tipoCombustible	Descripción del tipo de combustible
nombreAcreedor	Nombre del acreedor en caso de prenda
datosTecnicos	Objeto que contiene datos técnicos adicionales
normalizacion	Objeto con información de normalización
propietarios	Lista de propietarios generales
propietariosActuales	Lista de propietarios actuales con información detallada
codigoUUID	Código UUID de la consulta

Clases Internas

DatosTecnicos

Contiene información técnica adicional del vehículo:

Propiedad	Descripción
capacidadCarga	Capacidad de carga en kilogramos
pesoBrutoVehicular	Peso bruto del vehículo en kilogramos
noEjes	Número de ejes
puertas	Número de puertas
idTipoCombustible	ID del tipo de combustible
tipoCombustible	Descripción del tipo de combustible

Normalizacion

Contiene información sobre el estado de normalización del vehículo:

Propiedad	Descripción
deficienciaMatriculaInicial	Indica deficiencias en la matrícula inicial
vehiculoNormalizado	Indica si el vehículo ha sido normalizado

Propietario

Representa información general sobre los propietarios:

Propiedad	Descripción
cantidad	Cantidad de propietarios
tipoPropietario	Tipo de propietario

PropietarioActual

Contiene información detallada sobre los propietarios actuales:

Propiedad	Descripción
`id`	ID del propietario
`idTipoDocumento`	ID del tipo de documento
`tipoDocumento`	Tipo de documento de identidad
`noDocumento`	Número de documento
`nombreCompleto`	Nombre completo del propietario
`primerNombre`	Primer nombre
`segundoNombre`	Segundo nombre
`primerApellido`	Primer apellido
`segundoApellido`	Segundo apellido
`tipoPropiedad`	Tipo de propiedad (numérico)
`detallePropiedad`	Detalle del tipo de propiedad
`fechaNacimiento`	Fecha de nacimiento
`aux`	Campo auxiliar

Métodos

newInstance

```
public static RUNTWrapper newInstance(Map<String, Object> response)
```

Descripción: Método estático que crea una nueva instancia de `RUNTWrapper` a partir de una respuesta recibida, típicamente de una llamada API.

Parámetros:

- `response` (Map<String, Object>): Mapa que contiene la respuesta, donde se espera que haya una clave 'body' con los datos a deserializar.

Retorna: Una instancia de `RUNTWrapper` con los datos deserializados de la respuesta.

Funcionalidad:

1. Serializa el objeto almacenado en la clave 'body' del mapa de respuesta
2. Deserializa el JSON resultante a la clase `RUNTWrapper`
3. Devuelve la instancia creada

Uso Típico

Esta clase se utiliza principalmente para:

1. Recibir y estructurar datos de consultas al RUNT
2. Facilitar el acceso a información detallada de vehículos colombianos
3. Servir como modelo de datos para integraciones con el sistema RUNT

Consideraciones

- La clase contiene algunas propiedades que parecen duplicadas (como `noMotor` y `nroMotor`), lo que podría indicar variaciones en la nomenclatura del sistema de origen.
- Al ser `@NamespaceAccessible`, esta clase está diseñada para ser utilizada en contextos de paquetes gestionados o integraciones externas.

RUNTServiceHandler

Descripción General

`RUNTServiceHandler` es una clase virtual global que extiende `PlatformEventIntegrationObserver`. Esta clase está diseñada para manejar la integración con el servicio RUNT (Registro Único Nacional de Tránsito), permitiendo consultar información de vehículos a través de un servicio externo.

Autor

Felipe Correa \<felipe.correa@nespon.com\>

Proyecto

Foundation Salesforce - Suratech - UH-17752

Última Modificación

20 de febrero de 2025

Constantes

SERVICE_METHOD

- **Valor**: `"GET"`
- **Descripción**: Define el método HTTP utilizado para la comunicación con el servicio externo (GET).

SURATECH_MDT_CONFIG_NAME

- **Valor**: `"runt_endpoint"`
- **Descripción**: Nombre del registro de metadatos personalizado que contiene la configuración de conexión al servicio RUNT.

Métodos

preProcess(Map<String, Object> input, Map<String, Object> options)

Descripción

Método virtual que realiza el pre-procesamiento de la solicitud antes de invocar el servicio externo. Publica un evento de plataforma si está habilitado.

Parámetros

- `input` (Map<String, Object>): Datos de entrada para la solicitud.
- `options` (Map<String, Object>): Opciones adicionales para el procesamiento.

Retorno

- `Map<String, Object>`: Los datos de entrada, posiblemente modificados durante el pre-procesamiento.

Código

```
global virtual Map<String, Object> preProcess(
  Map<String, Object> input,
  Map<String, Object> options
) {
```

```
  Core.debug('Starting RUNT Integration Service Pre Process');
  if (
    options.containsKey('PlatformEventEnabled') &&
    ((Boolean) options.get('PlatformEventEnabled')) == true
  )
    publishPlatformEventIntegration('pre', JSON.serialize(input));
  return input;
}
```

invokeProcess(Map<String, Object> input, Map<String, Object> output, Map<String, Object> options)

Descripción

Método principal que orquesta el flujo completo de integración con el servicio RUNT. Obtiene la configuración de metadatos, realiza validaciones, emite la solicitud al servicio externo, procesa la respuesta y maneja excepciones.

Parámetros

- `input` (Map<String, Object>): Datos de entrada para la solicitud.
- `output` (Map<String, Object>): Mapa donde se almacenará el resultado de la operación.
- `options` (Map<String, Object>): Opciones adicionales para el procesamiento.

Retorno

- `Map<String, Object>`: Mapa con la respuesta procesada o `null` en caso de error.

Proceso

1. Obtiene la configuración del servicio desde metadatos personalizados.
2. Marca si los eventos de plataforma están habilitados para este servicio.
3. Crea un manejador de notificaciones externas.
4. Ejecuta el pre-procesamiento.
5. Publica un evento de plataforma para el inicio del proceso.
6. Emite la solicitud al servicio externo mediante el manejador de notificaciones.
7. Procesa la respuesta HTTP.
8. Publica un evento de plataforma con el resultado.
9. Ejecuta el post-procesamiento.
10. Maneja excepciones si ocurren durante el proceso.

Código

```
global Map<String, Object> invokeProcess(
  Map<String, Object> input,
  Map<String, Object> output,
  Map<String, Object> options
) {
  SF_IntegrationSetting__mdt config = Test.isRunningTest()
    ? ExternalServiceMetadataIntegrationMock.getIntegrationSettings(
        SURATECH_MDT_CONFIG_NAME
      )
    : SF_IntegrationSetting__mdt.getInstance(SURATECH_MDT_CONFIG_NAME);
  if (config == null) {
```

```
    throw new ExternalServiceException(
      'No existing "' +
        SURATECH_MDT_CONFIG_NAME +
        '" custom metadata type record at Integration Setting'
    );
  }

  //Marcamos la activacion si debemos ejecutar Eventos de Plataformas para este servicio.
  options.put('PlatformEventEnabled', config.Platform_Events_Enabled__c);

  ExternalNotificationHandler enh = new ExternalNotificationHandler();
  Map<String, Object> payload = new Map<String, Object>();

  try {
    payload = this.preProcess(input, options);
    Core.debug('Starting RUNT Integration Service In Process');
    publishPlatformEventIntegration('in', JSON.serialize(input));
    //Validations
    // ExternalServiceValidationUtil.plateValidation(
    //   (String) input.get('plate')
    // );

    System.HTTPResponse response = enh.emit(
      SERVICE_METHOD,
      config,
      String.valueOf(input.get('plate')),
      payload
    );

    Map<String, Object> responseMap = new Map<String, Object>{
      'body' => (Map<String, Object>) JSON.deserializeUntyped(
        response.getBody()
      ),
      'status' => response.getStatus(),
      'statusCode' => response.getStatusCode()
    };

    this.publishPlatFormEvent(
      response.getStatusCode(),
      JSON.serialize(payload),
      JSON.serialize(response.getBody())
    );
    return this.postProcess(responseMap, options);
  } catch (Exception e) {
    enh.controlException(
      'RUNT Service remote service',
      e,
      JSON.serialize(payload)
    );
    return null;
  }
}
```

postProcess(Map<String, Object> response, Map<String, Object> options)

Descripción

Método virtual que realiza el post-procesamiento de la respuesta después de invocar el servicio externo. Publica un evento de plataforma si está habilitado.

Parámetros

- `response` (Map<String, Object>): Datos de respuesta del servicio externo.
- `options` (Map<String, Object>): Opciones adicionales para el procesamiento.

Retorno

- `Map<String, Object>`: La respuesta, posiblemente modificada durante el post-procesamiento.

Código

```
global virtual Map<String, Object> postProcess(
  Map<String, Object> response,
  Map<String, Object> options
) {
  Core.debug('Starting RUNT Integration Service Post Process');
  if (
    options.containsKey('PlatformEventEnabled') &&
    ((Boolean) options.get('PlatformEventEnabled')) == true
  )
    publishPlatformEventIntegration('post', JSON.serialize(response));
  return response;
}
```

publishPlatformEventIntegration(String moment, String data)

Descripción

Publica diferentes tipos de eventos de plataforma según el momento del proceso de integración.

Parámetros

- `moment` (String): Momento del proceso ('pre', 'post', 'in').
- `data` (String): Datos serializados para incluir en el evento.

Proceso

Utiliza una declaración switch para determinar qué tipo de evento de plataforma publicar:

- 'pre': Evento de pre-procesamiento.
- 'post': Evento de post-procesamiento.
- 'in': Evento durante el procesamiento.

Código

```
global void publishPlatformEventIntegration(String moment, String data) {
  switch on moment.toLowerCase() {
    when 'pre' {
      publishInitPrePlatformEvent(data, 'RUNT Integration Service');
    }
    when 'post' {
      publishInitPosPlatformEvent(data, 'RUNT Integration Service');
    }
    when 'in' {
```

```
      publishPlatFormEvent(200, data, 'Starting RUNT In process');
    }
  }
}
```

Excepciones

ExternalServiceException

Esta excepción se lanza cuando no se encuentra la configuración del servicio RUNT en los metadatos personalizados.

Dependencias

- `PlatformEventIntegrationObserver`: Clase base que proporciona funcionalidades para publicar eventos de plataforma.
- `SF_IntegrationSetting__mdt`: Tipo de metadatos personalizado que contiene la configuración para la integración.
- `ExternalServiceMetadataIntegrationMock`: Clase mock utilizada durante las pruebas para simular la configuración de integración.
- `ExternalNotificationHandler`: Clase que maneja la emisión de solicitudes a servicios externos y el control de excepciones.
- `Core`: Clase de utilidad que proporciona el método `debug` para registro.
- `ExternalServiceException`: Excepción personalizada lanzada cuando hay problemas con la configuración del servicio.
- `ExternalServiceValidationUtil`: Clase de utilidad para validar datos (actualmente comentada).

Características Notables

Eventos de Plataforma

La clase implementa una integración completa con eventos de plataforma de Salesforce, permitiendo el seguimiento y monitoreo del proceso de integración en diferentes etapas:

- Pre-procesamiento
- Durante el procesamiento
- Post-procesamiento

Manejo de Configuración

La configuración del servicio se obtiene dinámicamente de un registro de metadatos personalizado (`SF_IntegrationSetting__mdt`), facilitando cambios en la configuración sin modificar el código.

Manejo de Pruebas

Incorpora lógica especial para entornos de prueba, utilizando una clase mock (`ExternalServiceMetadataIntegrationMock`) para simular la configuración durante las pruebas.

Virtualidad

Los métodos principales (`preProcess` y `postProcess`) son declarados como virtuales, permitiendo que las clases derivadas personalicen el comportamiento sin modificar el flujo general.

Ejemplo de Uso

```
// Crear una instancia del handler
RUNTServiceHandler handler = new RUNTServiceHandler();

// Preparar los datos de entrada
Map<String, Object> input = new Map<String, Object>{
```

```
  'plate' => 'ABC123'  // Placa del vehículo a consultar
};

// Mapas para opciones y salida
Map<String, Object> options = new Map<String, Object>();
Map<String, Object> output = new Map<String, Object>();

try {
  // Invocar el proceso de integración
  Map<String, Object> result = handler.invokeProcess(input, output, options);

  if (result != null) {
    // Procesar el resultado
    Map<String, Object> body = (Map<String, Object>)result.get('body');
    Integer statusCode = (Integer)result.get('statusCode');

    if (statusCode == 200) {
      // Extraer información del vehículo del resultado
      System.debug('Información del vehículo: ' + body);

      // Realizar operaciones con la información obtenida
      // ...
    } else {
      System.debug('Error en la consulta. Código: ' + statusCode);
    }
  }
} catch (Exception e) {
  System.debug('Error en la integración: ' + e.getMessage());
}
```

Extensión de la Clase

```
public class CustomRUNTServiceHandler extends RUNTServiceHandler {

    // Personalizar el pre-procesamiento
    global override Map<String, Object> preProcess(
        Map<String, Object> input,
        Map<String, Object> options
    ) {
        // Realizar validaciones adicionales
        String plate = (String)input.get('plate');
        if (plate != null) {
            // Convertir la placa a mayúsculas
            input.put('plate', plate.toUpperCase());

            // Validar el formato de la placa
            ExternalServiceValidationUtil.plateValidation(plate);
        }

        // Agregar información adicional
        input.put('requestTimestamp', System.now().getTime());
        input.put('requestUser', UserInfo.getUserId());

        // Llamar al método base
        return super.preProcess(input, options);
    }
```

```
    // Personalizar el post-procesamiento
    global override Map<String, Object> postProcess(
        Map<String, Object> response,
        Map<String, Object> options
    ) {
        // Transformar o enriquecer la respuesta
        if (response != null && response.containsKey('body')) {
            Map<String, Object> body = (Map<String, Object>)response.get('body');

            // Agregar información adicional
            body.put('processedTimestamp', System.now().getTime());

            // Transformar datos específicos si es necesario
            // ...
        }

        // Llamar al método base
        return super.postProcess(response, options);
    }
}
```

Notas Adicionales

1. La clase contiene código comentado que sugiere una implementación anterior o alternativa, incluyendo constantes para el nombre del procedimiento de integración y el nombre del servicio, así como una validación de placa de vehículo.
2. La clase está diseñada siguiendo un patrón de tres fases (pre-proceso, proceso, post-proceso), permitiendo personalizar cada fase a través de la herencia.
3. La integración con eventos de plataforma facilita el monitoreo y depuración del proceso de integración.
4. La clase es consciente del contexto de prueba (`Test.isRunningTest()`) y utiliza mocks para simular la configuración durante las pruebas.
5. La clase no realiza actualmente la validación de la placa del vehículo, aunque hay código comentado que sugiere que esta validación podría ser necesaria en algún momento.

LocalEmisionNotificationHandler

Descripción General

`LocalEmisionNotificationHandler` es una clase que gestiona la integración para notificaciones de emisión local. Extiende `PlatformEventIntegrationObserver` e implementa las interfaces `Queueable` y `Database.AllowsCallouts`, lo que le permite ser ejecutada como trabajo en cola asíncrono y realizar llamadas HTTP a servicios externos.

Autor y Modificaciones

- **Autor Original:** felipe.correa@nespon.com
- **Proyecto:** Foundation Salesforce - Suratech - UH-17589
- **Última modificación:** 03-03-2025
- **Modificado por:** fabian.lopez@nespon.com

Anotaciones

- `global virtual`: La clase es accesible desde cualquier namespace y puede ser extendida por otras clases.

Constantes

Constante	Valor	Descripción
`SERVICE_METHOD`	`'POST'`	Método HTTP utilizado para las llamadas al servicio externo
`SURATECH_MDT_CONFIG_NAME`	`'emisionlocal_endpoint'`	Nombre del registro de metadatos personalizado que contiene la configuración de integración

Propiedades

Propiedad	Tipo	Descripción
`input`	`Map<String, Object>`	Datos de entrada para el proceso de integración
`output`	`Map<String, Object>`	Datos de salida del proceso de integración
`options`	`Map<String, Object>`	Opciones de configuración para el proceso
`policyId`	`String`	Identificador de la póliza relacionada (declarado pero no utilizado en el código)

Constructor

```
global LocalEmisionNotificationHandler(
  Map<String, Object> input,
  Map<String, Object> output,
  Map<String, Object> options
)
```

Descripción: Inicializa una nueva instancia de la clase con los parámetros proporcionados.

Parámetros:

- `input`: Datos de entrada para el proceso
- `output`: Contenedor para los datos de salida
- `options`: Opciones de configuración

Métodos

execute

```
global void execute(QueueableContext context)
```

Descripción: Implementación del método requerido por la interfaz `Queueable`. Inicia el proceso de integración cuando el trabajo en cola es ejecutado.

Parámetros:

- `context`: Contexto de la ejecución en cola

preProcess

```
global virtual Map<String, Object> preProcess(
  Map<String, Object> input,
  Map<String, Object> options
)
```

Descripción: Método virtual que realiza el pre-procesamiento de los datos de entrada antes de la integración.

Parámetros:

- `input`: Datos de entrada para procesar
- `options`: Opciones de configuración

Retorna: Los datos de entrada posiblemente modificados

Funcionalidad:

1. Registra un mensaje de depuración
2. Si las opciones incluyen 'PlatformEventEnabled' establecido como verdadero, publica un evento de plataforma con los datos de entrada
3. Devuelve los datos de entrada (potencialmente modificados)

invokeProcess

```
global Map<String, Object> invokeProcess(
  Map<String, Object> input,
  Map<String, Object> output,
  Map<String, Object> options
)
```

Descripción: Método principal que orquesta todo el proceso de integración.

Parámetros:

- `input`: Datos de entrada para el proceso
- `output`: Contenedor para los datos de salida
- `options`: Opciones de configuración

Retorna: Mapa con la respuesta procesada del servicio externo o null en caso de error

Funcionalidad:

1. Obtiene la configuración de integración desde metadatos personalizados
2. Establece la configuración para eventos de plataforma
3. Ejecuta el pre-procesamiento de los datos
4. Valida que el producto esté incluido en los datos de entrada
5. Añade la dirección IP externa de Salesforce a los datos de entrada
6. Estructura la carga útil en el formato esperado por el servicio
7. Invoca el servicio externo a través de `ExternalNotificationHandler`
8. Procesa la respuesta y publica un evento de plataforma con los resultados
9. Ejecuta el post-procesamiento de la respuesta
10. Maneja cualquier excepción que ocurra durante el proceso

postProcess

```
global virtual Map<String, Object> postProcess(
  Map<String, Object> response,
  Map<String, Object> options
)
```

Descripción: Método virtual que realiza el post-procesamiento de la respuesta del servicio.

Parámetros:

- `response`: Respuesta del servicio externo
- `options`: Opciones de configuración

Retorna: La respuesta posiblemente modificada

Funcionalidad:

1. Registra un mensaje de depuración
2. Si las opciones incluyen 'PlatformEventEnabled' establecido como verdadero, publica un evento de plataforma con los datos de la respuesta
3. Devuelve la respuesta (potencialmente modificada)

getExternalIPAddress

```
public static String getExternalIPAddress()
```

Descripción: Método estático que obtiene la dirección IP externa del servidor de Salesforce.

Retorna: String con la dirección IP externa

Funcionalidad:

1. Registra los límites actuales a través de `Core.getSymmaryLimits()`
2. Realiza una llamada HTTP GET al servicio ipify.org
3. Registra y devuelve la respuesta que contiene la dirección IP

publishPlatformEventIntegration

```
global void publishPlatformEventIntegration(String moment, String data)
```

Descripción: Publica diferentes tipos de eventos de plataforma según el momento del proceso.

Parámetros:

- `moment`: Momento del proceso ('pre', 'post', 'in')
- `data`: Datos a incluir en el evento

Funcionalidad: Utiliza un switch para determinar qué tipo de evento publicar:

- 'pre': Publica un evento de pre-proceso inicial
- 'post': Publica un evento de post-proceso inicial
- 'in': Publica un evento general con código de estado 200

Flujo de Trabajo Principal

1. La clase puede ser instanciada y ejecutada como un trabajo en cola (Queueable)
2. Cuando se ejecuta, sigue un patrón de pre-proceso → proceso principal → post-proceso
3. Durante el proceso principal:
   - Obtiene configuración desde metadatos personalizados
   - Valida los datos de entrada
   - Enriquece los datos con información adicional (como la IP externa)
   - Realiza llamadas a servicios externos
   - Maneja las respuestas y las excepciones

Integración con Eventos de Plataforma

La clase utiliza eventos de plataforma para registrar y comunicar diferentes etapas del proceso de integración:

- Eventos de pre-proceso ('pre')
- Eventos durante el proceso ('in')
- Eventos de post-proceso ('post')

Dependencias

- `PlatformEventIntegrationObserver`: Clase base que proporciona funcionalidad para eventos de plataforma
- `ExternalNotificationHandler`: Clase para interactuar con servicios externos

- `Core`: Clase de utilidad para depuración y registro
- `SF_IntegrationSetting__mdt`: Metadatos personalizados para configuración de integración

Casos de Uso

Esta clase se utiliza principalmente para:

1. Enviar notificaciones sobre emisiones locales a servicios externos
2. Integrar Salesforce con sistemas externos para la emisión de pólizas o documentos
3. Proporcionar un flujo de trabajo asíncrono para operaciones de integración que pueden tomar tiempo

Consideraciones y Mejores Prácticas

- La clase está diseñada para ser extendida (virtual), permitiendo personalizar el comportamiento en clases derivadas
- Utiliza un patrón asíncrono para evitar problemas de límites de tiempo en Salesforce
- Implementa manejo de excepciones para garantizar que los errores sean registrados adecuadamente
- Usa eventos de plataforma para proporcionar visibilidad sobre el proceso de integración
- El nombre de configuración de metadatos sugiere variantes para diferentes productos (comentario indica "_Arrendamientos, _Viajes, _Autos, _Motos")

IntegrationProcedureExecutor

Descripción General

`IntegrationProcedureExecutor` es una interfaz global que define el contrato para la ejecución de procedimientos de integración. Esta interfaz establece un método estándar para invocar procedimientos de integración con parámetros de entrada y obtener resultados estructurados.

Autor

Jean Carlos Melendez

Última Modificación

28 de enero de 2025

Definición de la Interfaz

```
global interface IntegrationProcedureExecutor {
  Map<String, Object> execute(String procedureName, Map<String, Object> input);
}
```

Métodos

execute(String procedureName, Map<String, Object> input)

Descripción

Este método ejecuta un procedimiento de integración especificado, pasando los parámetros de entrada proporcionados y devolviendo los resultados del procedimiento.

Parámetros

- `procedureName` (String): Nombre del procedimiento de integración a ejecutar.
- `input` (Map<String, Object>): Mapa con los parámetros de entrada para el procedimiento.

Retorno

- `Map<String, Object>`: Mapa con los resultados del procedimiento de integración.

Propósito y Aplicaciones

El propósito principal de esta interfaz es proporcionar una abstracción para la ejecución de procedimientos de integración, ofreciendo:

1. **Desacoplamiento**: Separar el código que necesita invocar procedimientos de integración de la implementación específica de cómo se ejecutan estos procedimientos.
2. **Flexibilidad**: Permitir múltiples implementaciones para diferentes contextos (por ejemplo, producción, pruebas, mock).
3. **Consistencia**: Establecer un contrato estándar para la ejecución de procedimientos de integración en toda la aplicación.
4. **Testabilidad**: Facilitar la creación de mocks para pruebas unitarias.

Esta interfaz es especialmente útil en escenarios como:

- **Sistemas de Integración**: Facilitar la comunicación con sistemas externos a través de procedimientos de integración

estandarizados.

- **Orquestación de Procesos**: Coordinar diferentes pasos en flujos de trabajo complejos que pueden involucrar múltiples sistemas.
- **Abstracción de Detalles Técnicos**: Ocultar los detalles específicos de cómo se ejecutan los procedimientos de integración a los componentes que los utilizan.
- **Cambio de Implementaciones**: Permitir cambiar la forma en que se ejecutan los procedimientos de integración sin afectar a los componentes que los invocan.

Ejemplos de Implementación

Implementación para OmniScript/Integration Procedures de Vlocity

```
global class VlocityIntegrationProcedureExecutor implements IntegrationProcedureExecutor {

    global Map<String, Object> execute(String procedureName, Map<String, Object> input) {
        try {
            // Inicializar el resultado
            Map<String, Object> result = new Map<String, Object>();

            // Verificar parámetros
            if (String.isBlank(procedureName)) {
                throw new IntegrationProcedureException('El nombre del procedimiento no puede estar vacío');
            }

            if (input == null) {
                input = new Map<String, Object>();
            }

            // Crear la petición para el procedimiento de integración de Vlocity
            Map<String, Object> request = new Map<String, Object>{
                'procedureName' => procedureName,
                'input' => input
            };

            // Invocar el procedimiento de integración a través de la API de Vlocity
            Map<String, Object> response = (Map<String, Object>)
vlocity_ins.IntegrationProcedureService.runIntegrationProcedureQueueable(
                procedureName,
                input
            );

            // Verificar la respuesta
            if (response == null) {
                throw new IntegrationProcedureException('No se recibió respuesta del procedimiento de
integración');
            }

            return response;
        } catch (Exception e) {
            System.debug(LoggingLevel.ERROR, 'Error ejecutando procedimiento de integración: ' + procedureName
+ '. Error: ' + e.getMessage());
            throw new IntegrationProcedureException('Error ejecutando procedimiento: ' + e.getMessage(), e);
        }
    }

    // Clase de excepción personalizada
```

```
    global class IntegrationProcedureException extends Exception {}
}
```

Implementación a través de Remote Actions

```
global class RemoteActionIntegrationProcedureExecutor implements IntegrationProcedureExecutor {

    global Map<String, Object> execute(String procedureName, Map<String, Object> input) {
        try {
            // Validar parámetros
            if (String.isBlank(procedureName)) {
                throw new IntegrationProcedureException('El nombre del procedimiento no puede estar vacío');
            }

            if (input == null) {
                input = new Map<String, Object>();
            }

            // Preparar parámetros para Remote Action
            String namespace = 'vlocity_ins';
            String classname = 'IntegrationProcedureService';
            String methodName = 'invokeIntegrationProcedure';
            String options = '{}';

            // Convertir el mapa de entrada a JSON
            String inputJson = JSON.serialize(input);

            // Invocar Remote Action
            Object result = vlocity_ins.RemoteActionController.invokeMethod(
                namespace,
                classname,
                methodName,
                procedureName,
                inputJson,
                options
            );

            // Procesar resultado
            if (result == null) {
                return new Map<String, Object>();
            }

            // Convertir el resultado a un mapa
            Map<String, Object> resultMap;
            if (result instanceof String) {
                resultMap = (Map<String, Object>) JSON.deserializeUntyped((String) result);
            } else if (result instanceof Map<String, Object>) {
                resultMap = (Map<String, Object>) result;
            } else {
                throw new IntegrationProcedureException('Resultado inesperado del tipo: ' +
result.getClass().getName());
            }

            return resultMap;
        } catch (Exception e) {
            System.debug(LoggingLevel.ERROR, 'Error ejecutando procedimiento de integración: ' + procedureName
+ '. Error: ' + e.getMessage());
```

```
            throw new IntegrationProcedureException('Error ejecutando procedimiento: ' + e.getMessage(), e);
        }
    }

    // Clase de excepción personalizada
    global class IntegrationProcedureException extends Exception {}
}
```

Implementación Mock para Pruebas

```
@IsTest
global class MockIntegrationProcedureExecutor implements IntegrationProcedureExecutor {

    // Mapa para almacenar respuestas mockificadas por nombre de procedimiento
    private static Map<String, Map<String, Object>> mockResponses = new Map<String, Map<String, Object>>();

    // Bandera para simular errores
    private static Boolean throwError = false;
    private static String errorMessage = 'Error simulado en procedimiento de integración';

    global Map<String, Object> execute(String procedureName, Map<String, Object> input) {
        // Registrar la llamada para verificación en las pruebas
        recordExecution(procedureName, input);

        // Simular error si está configurado
        if (throwError) {
            throw new IntegrationProcedureException(errorMessage);
        }

        // Devolver respuesta mockificada si existe
        if (mockResponses.containsKey(procedureName)) {
            return mockResponses.get(procedureName);
        }

        // Respuesta por defecto
        return new Map<String, Object>{
            'success' => true,
            'procedureName' => procedureName,
            'mockInput' => input,
            'result' => 'Mock result for ' + procedureName
        };
    }

    // Métodos para configurar el comportamiento del mock

    global static void setMockResponse(String procedureName, Map<String, Object> response) {
        mockResponses.put(procedureName, response);
    }

    global static void clearMockResponses() {
        mockResponses.clear();
    }

    global static void setThrowError(Boolean shouldThrow, String message) {
        throwError = shouldThrow;
        if (String.isNotBlank(message)) {
            errorMessage = message;
```

```
        }
    }

    // Registro de ejecuciones para verificación en pruebas
    private static List<ExecutionRecord> executionRecords = new List<ExecutionRecord>();

    private void recordExecution(String procedureName, Map<String, Object> input) {
        executionRecords.add(new ExecutionRecord(procedureName, input));
    }

    global static List<ExecutionRecord> getExecutionRecords() {
        return executionRecords;
    }

    global static void clearExecutionRecords() {
        executionRecords.clear();
    }

    // Clase interna para registrar ejecuciones
    global class ExecutionRecord {
        public String procedureName;
        public Map<String, Object> input;

        public ExecutionRecord(String procedureName, Map<String, Object> input) {
            this.procedureName = procedureName;
            this.input = input;
        }
    }

    // Clase de excepción personalizada
    global class IntegrationProcedureException extends Exception {}
}
```

Ejemplo de Uso

Uso Básico en un Servicio

```
public class QuotingService {

    private final IntegrationProcedureExecutor ipExecutor;

    // Constructor con inyección de dependencia
    public QuotingService(IntegrationProcedureExecutor ipExecutor) {
        this.ipExecutor = ipExecutor;
    }

    // Constructor sin parámetros para uso general
    public QuotingService() {
        this.ipExecutor = new VlocityIntegrationProcedureExecutor();
    }

    // Método para calcular prima de un producto
    public Map<String, Object> calculatePremium(String productCode, Map<String, Object> customerData) {
        try {
            // Preparar los datos de entrada
            Map<String, Object> input = new Map<String, Object>{
```

```
                'productCode' => productCode,
                'customerData' => customerData,
                'calculationDate' => System.now()
            };

            // Ejecutar el procedimiento de integración
            Map<String, Object> result = ipExecutor.execute('CalculatePremium', input);

            // Procesar y validar resultados
            if (!result.containsKey('premium')) {
                throw new QuotingException('La respuesta del cálculo de prima no contiene el campo "premium"');
            }

            return result;
        } catch (Exception e) {
            throw new QuotingException('Error al calcular prima: ' + e.getMessage(), e);
        }
    }

    // Clase de excepción personalizada
    public class QuotingException extends Exception {}
}
```

Uso con Factory para Obtener la Implementación Adecuada

```
public class IntegrationProcedureExecutorFactory {

    // Singleton instance
    private static IntegrationProcedureExecutorFactory instance;

    // Obtener instancia singleton
    public static IntegrationProcedureExecutorFactory getInstance() {
        if (instance == null) {
            instance = new IntegrationProcedureExecutorFactory();
        }
        return instance;
    }

    // Obtener el ejecutor adecuado basado en configuración o contexto
    public IntegrationProcedureExecutor getExecutor() {
        // En un entorno de prueba, usar el mock
        if (Test.isRunningTest()) {
            return new MockIntegrationProcedureExecutor();
        }

        // Verificar si se debe usar una implementación alternativa
        SF_IntegrationConfig__mdt config = SF_IntegrationConfig__mdt.getInstance('IPExecutorConfig');
        if (config != null && String.isNotBlank(config.AlternativeImplementation__c)) {
            return
(IntegrationProcedureExecutor)Type.forName(config.AlternativeImplementation__c).newInstance();
        }

        // Por defecto, usar la implementación estándar
        return new VlocityIntegrationProcedureExecutor();
    }
}
```

```
// Uso con factory
public class RatingService {

    public Map<String, Object> rateProduct(String productId, Map<String, Object> ratingFactors) {
        IntegrationProcedureExecutor executor =
IntegrationProcedureExecutorFactory.getInstance().getExecutor();

        Map<String, Object> input = new Map<String, Object>{
            'productId' => productId,
            'ratingFactors' => ratingFactors
        };

        return executor.execute('RateProduct', input);
    }
}
```

Uso en Pruebas Unitarias

```
@IsTest
private class QuotingServiceTest {

    @IsTest
    static void testCalculatePremium() {
        // Configurar el mock
        MockIntegrationProcedureExecutor mockExecutor = new MockIntegrationProcedureExecutor();

        // Configurar respuesta esperada
        Map<String, Object> mockResponse = new Map<String, Object>{
            'premium' => 1200.50,
            'discounts' => new Map<String, Object>{
                'goodDriver' => 100.00,
                'multiPolicy' => 50.00
            },
            'totalPremium' => 1050.50,
            'currency' => 'USD'
        };

        MockIntegrationProcedureExecutor.setMockResponse('CalculatePremium', mockResponse);

        // Crear instancia del servicio con el mock
        QuotingService service = new QuotingService(mockExecutor);

        // Preparar datos de prueba
        String productCode = 'AUTO-001';
        Map<String, Object> customerData = new Map<String, Object>{
            'age' => 35,
            'vehicleValue' => 25000,
            'drivingHistory' => 'Clean'
        };

        // Ejecutar el método a probar
        Test.startTest();
        Map<String, Object> result = service.calculatePremium(productCode, customerData);
        Test.stopTest();

        // Verificar resultados
        System.assertEquals(1200.50, result.get('premium'), 'Premium should match mock value');
```

```
        System.assertEquals(1050.50, result.get('totalPremium'), 'Total premium should match mock value');
        System.assertEquals('USD', result.get('currency'), 'Currency should match mock value');

        // Verificar que se llamó al procedimiento correcto con los parámetros esperados
        List<MockIntegrationProcedureExecutor.ExecutionRecord> executions =
MockIntegrationProcedureExecutor.getExecutionRecords();
        System.assertEquals(1, executions.size(), 'Should have executed exactly one procedure');
        System.assertEquals('CalculatePremium', executions[0].procedureName, 'Should have called the correct
procedure');
        System.assertEquals(productCode, executions[0].input.get('productCode'), 'Should have passed the
correct product code');
    }

    @IsTest
    static void testCalculatePremiumError() {
        // Configurar el mock para lanzar error
        MockIntegrationProcedureExecutor mockExecutor = new MockIntegrationProcedureExecutor();
        MockIntegrationProcedureExecutor.setThrowError(true, 'Error de cálculo simulado');

        // Crear instancia del servicio con el mock
        QuotingService service = new QuotingService(mockExecutor);

        // Preparar datos de prueba
        String productCode = 'AUTO-001';
        Map<String, Object> customerData = new Map<String, Object>{
            'age' => 35,
            'vehicleValue' => 25000
        };

        // Ejecutar el método y verificar que se lance la excepción esperada
        Test.startTest();
        try {
            service.calculatePremium(productCode, customerData);
            System.assert(false, 'Should have thrown an exception');
        } catch (QuotingService.QuotingException e) {
            System.assert(e.getMessage().contains('Error de cálculo simulado'), 'Exception should contain the
mock error message');
        }
        Test.stopTest();
    }
}
```

Consideraciones de Diseño

Ventajas

1. **Desacoplamiento**: Los componentes que necesitan ejecutar procedimientos de integración no dependen de una implementación específica.
2. **Flexibilidad**: Se pueden crear diferentes implementaciones para diferentes contextos o requisitos.
3. **Testabilidad**: Facilita la creación de mocks para pruebas unitarias.
4. **Consistencia**: Proporciona un contrato estándar para la ejecución de procedimientos de integración.

Desafíos

1. **Manejo de Errores**: Las implementaciones deben proporcionar un manejo coherente de errores para mantener un comportamiento predecible.
2. **Tipado Dinámico**: El uso de `Map<String, Object>` proporciona flexibilidad pero requiere verificaciones adicionales de tipo en tiempo de ejecución.
3. **Sincronización**: La interfaz actual es síncrona, lo que podría no ser adecuado para procedimientos de larga duración.

Patrones de Diseño Relacionados

Inyección de Dependencias

La interfaz `IntegrationProcedureExecutor` facilita la inyección de dependencias, permitiendo que los componentes reciban la implementación específica a través de constructores o setters.

Estrategia (Strategy)

Permite definir una familia de algoritmos (diferentes formas de ejecutar procedimientos de integración), encapsularlos y hacerlos intercambiables.

Fábrica (Factory)

Puede utilizarse junto con un patrón de fábrica para proporcionar la implementación adecuada basada en el contexto.

Adaptador (Adapter)

Algunas implementaciones pueden actuar como adaptadores, permitiendo que diferentes sistemas de procedimientos de integración se utilicen de manera uniforme.

Relación con Otras Interfaces y Clases

Esta interfaz está relacionada y es utilizada por varias clases en el sistema, como:

1. **QuoteProductHandler**: Utiliza un `IntegrationProcedureExecutor` para ejecutar procedimientos relacionados con cotizaciones.
2. **IssueProductHandler**: Emplea un ejecutor para gestionar la emisión de productos.
3. **MockIntegrationProcedureExecutor**: Implementación de prueba que simula la ejecución de procedimientos de integración.

Mejores Prácticas

1. **Manejo Adecuado de Errores**: Implementar un manejo robusto de errores en todas las implementaciones.
2. **Registro de Actividad**: Incluir registro de actividad para facilitar el seguimiento y la depuración.
3. **Validación de Parámetros**: Validar los parámetros de entrada para evitar errores inesperados.
4. **Implementaciones con Caché**: Considerar la implementación de mecanismos de caché para procedimientos frecuentemente utilizados.
5. **Timeout y Reintentos**: Implementar mecanismos de timeout y reintentos para manejar situaciones de fallo transitorio.

Notas Adicionales

1. Esta interfaz está marcada como `global`, lo que indica que está diseñada para ser accesible desde diferentes paquetes o namespaces.
2. El uso de mapas genéricos (`Map<String, Object>`) proporciona flexibilidad pero requiere un manejo cuidadoso de los tipos en tiempo de ejecución.
3. Esta interfaz forma parte de un sistema más amplio para la integración y orquestación de procesos, trabajando en conjunto con otras interfaces y clases.

4. Para procedimientos de larga duración, podría ser necesario considerar una variante asíncrona de esta interfaz.

InsServiceAdapter

Descripción General

`InsServiceAdapter` es una clase que implementa la interfaz `Adapter` y proporciona funcionalidad para interactuar con servicios de Vlocity Insurance. Esta clase actúa como un adaptador que facilita la comunicación entre el sistema Salesforce y los servicios de seguros externos, específicamente para acceder a productos calificados (rated products).

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

2 de septiembre de 2024

Decoradores y Accesibilidad

La clase está marcada con `@namespaceAccessible`, lo que indica que está diseñada para ser accesible desde diferentes espacios de nombres (namespaces). Esto es especialmente relevante en contextos de paquetes gestionados donde se necesita que componentes específicos sean accesibles desde fuera del namespace del paquete.

Estructura de la Clase

```
@namespaceAccessible
public class InsServiceAdapter implements Adapter {
  @namespaceAccessible
  public Map<String, Object> adaptee() {
    return new Map<String, Object>();
  }

  @namespaceAccessible
  public static Boolean useInsAdapter() {
    // Implementación del método
  }
}
```

Interfaz Implementada

Adapter

La clase implementa la interfaz `Adapter`. Aunque no se muestra la definición de esta interfaz en el código proporcionado, se puede inferir que requiere al menos un método `adaptee()` que devuelve un `Map<String, Object>`.

Métodos

adaptee()

Descripción

Método que implementa la interfaz `Adapter` y devuelve un mapa vacío. Según el comentario comentado, estaba previsto que devolviera una instancia de `URLGenerationNotificationHandler`, pero actualmente devuelve un mapa vacío.

Parámetros

Ninguno.

Retorno

- `Map<String, Object>`: Un mapa vacío.

Código

```
@namespaceAccessible
public Map<String, Object> adaptee() {
  // return new URLGenerationNotificationHandler();
  return new Map<String, Object>();
}
```

useInsAdapter()

Descripción

Método estático que intenta realizar una llamada HTTP a un servicio de Vlocity Insurance para obtener productos calificados. Determina si el adaptador de seguros debe ser utilizado basándose en el éxito de esta llamada.

Parámetros

Ninguno.

Retorno

- `Boolean`: `true` si la llamada al servicio fue exitosa (estado 200), `false` en caso contrario.

Proceso

1. Inicializa una variable booleana `isSuccess` como `false`.
2. Crea una nueva solicitud HTTP.
3. Consulta el nombre de la instancia de la organización actual.
4. Configura el endpoint de la solicitud usando una credencial nombrada para el servicio de integración de Vlocity.
5. Establece el método HTTP como 'GET'.
6. Crea una instancia de Http y envía la solicitud.
7. Analiza la respuesta:
   - Si el código de estado es 200, registra el cuerpo de la respuesta y establece `isSuccess` como `true`.
   - Si el código de estado no es 200, registra el estado de error y mantiene `isSuccess` como `false`.
8. Devuelve el valor de `isSuccess`.

Código

```
@namespaceAccessible
public static Boolean useInsAdapter() {
  Boolean isSuccess = false;
  HttpRequest insServiceRequest = new HttpRequest();

  String instanceName = [SELECT InstanceName FROM Organization LIMIT 1]
  .InstanceName;
  System.debug(instanceName);

  // insServiceRequest.setEndpoint(
  //   'callout:salesforce_ins_service_adaptar/services/apexrest/vlocity_insurance/ratedproducts'
  // );
  insServiceRequest.setEndpoint(

'callout:Suratech_Salesforce_Vlocity_Integration_Credential/services/apexrest/vlocity_insurance/ratedproducts'
  );
```

```
  insServiceRequest.setMethod('GET');

  Http http = new Http();
  HttpResponse res = http.send(insServiceRequest);

  if (res.getStatusCode() == 200) {
    System.debug('Response: ' + res.getBody());
    isSuccess = true;
  } else {
    System.debug('Error: ' + res.getStatus());
    isSuccess = false;
  }

  return isSuccess;
}
```

Observaciones sobre el Código

Puntos a Destacar

1. **Implementación Parcial o Transitoria**: El método `adaptee()` parece estar en un estado transitorio, ya que contiene código comentado y devuelve un mapa vacío.
2. **Uso de Credencial Nombrada**: La clase utiliza una credencial nombrada (`Suratech_Salesforce_Vlocity_Integration_Credential`) para la autenticación segura con el servicio externo.
3. **Consulta de Organización**: La clase consulta la información de la organización actual, aunque actualmente solo utiliza esta información para registro de depuración.

Posibles Mejoras

1. **Implementación Completa de `adaptee()`**: Completar la implementación del método `adaptee()` para que devuelva la instancia apropiada o un mapa con datos relevantes.
2. **Manejo de Excepciones**: Añadir manejo de excepciones para capturar y gestionar adecuadamente posibles errores durante la llamada HTTP.
3. **Parametrización**: Considerar hacer que el endpoint sea configurable, posiblemente a través de metadatos personalizados, en lugar de codificarlo directamente.
4. **Timeout y Reintentos**: Implementar lógica de timeout y reintentos para manejar situaciones donde el servicio externo no responde o lo hace lentamente.
5. **Utilización de la Información de Instancia**: Hacer un uso más significativo de la información de instancia que se está consultando.

Versión Mejorada Propuesta

```
/**
 * @description       : Adaptador para servicios de seguros de Vlocity
 * @author            : Soulberto Lorenzo <soulberto@cloudblue.us>
 * @group             : Adapters
 * @last modified on  : 09-02-2024
 * @last modified by  : Soulberto Lorenzo <soulberto@cloudblue.us>
 **/
@namespaceAccessible
public class InsServiceAdapter implements Adapter {
  // Constantes para configuración
```

```
  private static final String VLOCITY_ENDPOINT_PATH = '/services/apexrest/vlocity_insurance/ratedproducts';
  private static final String NAMED_CREDENTIAL = 'Suratech_Salesforce_Vlocity_Integration_Credential';
  private static final Integer DEFAULT_TIMEOUT = 30000; // 30 segundos

  /**
   * @description Implementa la interfaz Adapter devolviendo un adaptador para servicios de seguros
   * @author Soulberto Lorenzo <soulberto@cloudblue.us> | 09-02-2024
   * @return Map<String, Object> Mapa con información del adaptador
   **/
  @namespaceAccessible
  public Map<String, Object> adaptee() {
    Map<String, Object> adapterInfo = new Map<String, Object>();

    try {
      // Verificar disponibilidad del servicio
      Boolean serviceAvailable = checkServiceAvailability();

      // Añadir información relevante al mapa
      adapterInfo.put('type', 'InsuranceServiceAdapter');
      adapterInfo.put('serviceAvailable', serviceAvailable);
      adapterInfo.put('instanceName', getOrganizationInstanceName());

      // Si el servicio está disponible, añadir información adicional
      if (serviceAvailable) {
        // Aquí se podría añadir más información específica del servicio
        // Por ejemplo, versión de API, capacidades disponibles, etc.
      }
    } catch (Exception e) {
      // Registrar error y añadir información de error al mapa
      System.debug(LoggingLevel.ERROR, 'Error en adaptee(): ' + e.getMessage());
      adapterInfo.put('error', e.getMessage());
      adapterInfo.put('errorType', e.getTypeName());
    }

    return adapterInfo;
  }

  /**
   * @description Verifica si se debe utilizar el adaptador de seguros basándose en la disponibilidad del
servicio
   * @author Soulberto Lorenzo <soulberto@cloudblue.us> | 09-02-2024
   * @return Boolean true si el servicio está disponible, false en caso contrario
   **/
  @namespaceAccessible
  public static Boolean useInsAdapter() {
    return checkServiceAvailability();
  }

  /**
   * @description Verifica la disponibilidad del servicio de seguros
   * @return Boolean true si el servicio está disponible, false en caso contrario
   **/
  private static Boolean checkServiceAvailability() {
    try {
      HttpRequest insServiceRequest = new HttpRequest();

      String endpoint = 'callout:' + NAMED_CREDENTIAL + VLOCITY_ENDPOINT_PATH;
```

```
      System.debug('Verificando disponibilidad del servicio: ' + endpoint);

      insServiceRequest.setEndpoint(endpoint);
      insServiceRequest.setMethod('GET');
      insServiceRequest.setTimeout(DEFAULT_TIMEOUT);

      Http http = new Http();
      HttpResponse res = http.send(insServiceRequest);

      Integer statusCode = res.getStatusCode();
      System.debug('Código de estado de respuesta: ' + statusCode);

      if (statusCode == 200) {
        System.debug('Servicio disponible. Respuesta: ' + res.getBody());
        return true;
      } else {
        System.debug('Servicio no disponible. Estado: ' + res.getStatus());
        logServiceError(statusCode, res.getStatus(), res.getBody());
        return false;
      }
    } catch (Exception e) {
      System.debug(LoggingLevel.ERROR, 'Error al verificar disponibilidad del servicio: ' + e.getMessage());
      return false;
    }
  }

  /**
   * @description Obtiene el nombre de la instancia de la organización actual
   * @return String Nombre de la instancia
   **/
  private static String getOrganizationInstanceName() {
    try {
      Organization org = [SELECT InstanceName FROM Organization LIMIT 1];
      return org.InstanceName;
    } catch (Exception e) {
      System.debug(LoggingLevel.ERROR, 'Error al obtener el nombre de la instancia: ' + e.getMessage());
      return '';
    }
  }

  /**
   * @description Registra información detallada sobre errores del servicio
   * @param statusCode Código de estado HTTP
   * @param status Descripción del estado
   * @param responseBody Cuerpo de la respuesta
   **/
  private static void logServiceError(Integer statusCode, String status, String responseBody) {
    // Aquí se podría implementar lógica para registrar errores en un objeto personalizado,
    // enviar notificaciones, o realizar otras acciones según la política de gestión de errores

    System.debug(LoggingLevel.ERROR, 'Error en servicio de seguros - Código: ' + statusCode +
                 ', Estado: ' + status + ', Respuesta: ' + responseBody);
  }

  /**
   * @description Realiza una llamada al servicio de seguros para obtener productos calificados
   * @param parameters Parámetros de la solicitud
```

```
   * @return Map<String, Object> Respuesta del servicio
   * @throws InsServiceException Si hay un error en la llamada al servicio
   **/
  @namespaceAccessible
  public static Map<String, Object> getRatedProducts(Map<String, Object> parameters) {
    try {
      // Verificar disponibilidad del servicio
      if (!checkServiceAvailability()) {
        throw new InsServiceException('El servicio de seguros no está disponible');
      }

      // Implementar lógica para obtener productos calificados
      // ...

      return new Map<String, Object>(); // Placeholder
    } catch (Exception e) {
      throw new InsServiceException('Error al obtener productos calificados: ' + e.getMessage(), e);
    }
  }

  /**
   * Clase de excepción personalizada para errores del servicio de seguros
   */
  public class InsServiceException extends Exception {}
}
```

Ejemplo de Uso

Uso Básico para Verificar Disponibilidad

```
// Verificar si el adaptador de seguros debe ser utilizado
Boolean useInsAdapter = InsServiceAdapter.useInsAdapter();

if (useInsAdapter) {
    System.debug('El servicio de seguros está disponible, se utilizará el adaptador');
    // Lógica adicional cuando el servicio está disponible
} else {
    System.debug('El servicio de seguros no está disponible, se utilizará una alternativa');
    // Lógica alternativa cuando el servicio no está disponible
}
```

Uso del Adaptador

```
// Crear una instancia del adaptador
InsServiceAdapter adapter = new InsServiceAdapter();

// Obtener información del adaptador
Map<String, Object> adapterInfo = adapter.adaptee();

// Verificar si el servicio está disponible
Boolean serviceAvailable = (Boolean)adapterInfo.get('serviceAvailable');

if (serviceAvailable) {
    // Utilizar el adaptador para obtener productos calificados
    Map<String, Object> parameters = new Map<String, Object>{
        'productType' => 'Auto',
```

```
        'clientId' => '12345'
    };

    try {
        Map<String, Object> ratedProducts = InsServiceAdapter.getRatedProducts(parameters);
        // Procesar los productos calificados
        // ...
    } catch (InsServiceAdapter.InsServiceException e) {
        System.debug('Error al obtener productos calificados: ' + e.getMessage());
        // Manejar el error
    }
} else {
    System.debug('El servicio no está disponible: ' + adapterInfo.get('error'));
    // Manejar la falta de disponibilidad del servicio
}
```

Contexto y Uso Típico

Vlocity Insurance y Salesforce

Esta clase probablemente forma parte de una integración entre Salesforce y Vlocity, una solución específica para la industria de seguros que ahora es parte de Salesforce Industries. Vlocity proporciona componentes y funcionalidad específica para varias industrias, incluyendo seguros.

Casos de Uso Típicos

1. **Cotización de Seguros**: Obtener productos calificados con precios y coberturas específicas para un cliente.
2. **Configuración de Productos**: Acceder a reglas de configuración y opciones para productos de seguros.
3. **Verificación de Elegibilidad**: Determinar si un cliente es elegible para ciertos productos o coberturas.
4. **Cálculo de Primas**: Obtener cálculos de primas basados en información específica del cliente.

Patrones de Diseño Utilizados

Patrón Adaptador (Adapter)

La clase implementa el patrón Adaptador, que permite que interfaces incompatibles trabajen juntas. En este caso, adapta la comunicación con el servicio de Vlocity Insurance a una interfaz común definida por `Adapter`.

Patrón de Comprobación de Disponibilidad (Availability Check)

El método `useInsAdapter()` implementa un patrón de comprobación de disponibilidad, que verifica si un servicio externo está disponible antes de intentar utilizarlo para operaciones críticas.

Consideraciones de Seguridad

1. **Credenciales Nombradas**: El uso de credenciales nombradas es una buena práctica para manejar la autenticación, ya que abstrae las credenciales del código.
2. **Registro de Datos Sensibles**: El código actual registra el cuerpo completo de la respuesta, lo que podría contener información sensible. Considerar filtrar o anonimizar datos sensibles en los registros.

Consideraciones de Rendimiento

1. **Timeout**: El código original no establece un timeout específico para la llamada HTTP, lo que podría resultar en esperas prolongadas si el servicio externo no responde.

2. **Consulta de Organización**: La consulta a la tabla Organization se realiza en cada llamada al método `useInsAdapter()`. Para uso frecuente, considerar cachear esta información.

Notas Adicionales

1. La clase parece estar en desarrollo o en transición, como lo indican los comentarios y la implementación incompleta de `adaptee()`.
2. La anotación `@namespaceAccessible` sugiere que esta clase está diseñada para ser parte de un paquete gestionado que necesita exponer funcionalidad a otros paquetes o al código de la organización.
3. La clase no incluye manejo de excepciones, lo que podría llevar a comportamientos inesperados si la llamada HTTP falla por razones distintas a un código de estado no 200 (por ejemplo, timeout, problemas de DNS, etc.).
4. El código actual no incluye documentación completa ni pruebas unitarias, que serían recomendables para una clase de esta naturaleza.

GenerateURLServiceHandler

Descripción General

`GenerateURLServiceHandler` es una clase virtual global que extiende `PlatformEventIntegrationObserver` y está diseñada para manejar la integración con servicios externos para la generación de URLs. Esta clase implementa un patrón de procesamiento en tres fases (pre-proceso, proceso principal y post-proceso) y proporciona capacidades de publicación de eventos de plataforma para monitoreo y auditoría.

Autor

Felipe Correa \<felipe.correa@nespon.com\>

Proyecto

Foundation Salesforce - Suratech - UH-17594

Última Modificación

20 de febrero de 2025 por "ChangeMeIn@UserSettingsUnder.SFDoc"

Estructura de la Clase

```
global virtual class GenerateURLServiceHandler extends PlatformEventIntegrationObserver {
  // Constantes de configuración
  private static final String SERVICE_METHOD = 'POST';
  private static final String SURATECH_MDT_CONFIG_NAME = 'url_endpoint';

  // Métodos de procesamiento
  global virtual Map<String, Object> preProcess(Map<String, Object> input, Map<String, Object> options);
  global Map<String, Object> invokeProcess(Map<String, Object> input, Map<String, Object> output, Map<String,
Object> options);
  global virtual Map<String, Object> postProcess(Map<String, Object> response, Map<String, Object> options);
  global void publishPlatformEventIntegration(String moment, String data);
}
```

Herencia

- **Clase Base**: `PlatformEventIntegrationObserver`
- **Modificadores**: `global virtual`

Constantes

Nombre	Valor	Descripción
`SERVICE_METHOD`	`'POST'`	Método HTTP utilizado para las llamadas al servicio externo
`SURATECH_MDT_CONFIG_NAME`	`'url_endpoint'`	Nombre del registro de metadatos personalizados que contiene la configuración del servicio

Métodos

preProcess(Map<String, Object> input, Map<String, Object> options)

Descripción

Método virtual que se ejecuta antes del procesamiento principal. Realiza la preparación inicial de los datos y opcionalmente publica eventos de plataforma para el monitoreo del proceso.

Parámetros

- `input` (Map<String, Object>): Datos de entrada para el procesamiento.
- `options` (Map<String, Object>): Opciones de configuración para el procesamiento.

Retorno

- `Map<String, Object>`: Los datos de entrada, potencialmente modificados durante el pre-procesamiento.

Proceso

1. Registra el inicio del pre-procesamiento.
2. Verifica si los eventos de plataforma están habilitados en las opciones.
3. Si están habilitados, publica un evento de plataforma con los datos de entrada.
4. Devuelve los datos de entrada sin modificaciones.

Código

```
global virtual Map<String, Object> preProcess(
  Map<String, Object> input,
  Map<String, Object> options
) {
  Core.debug('Starting Generate URL Integration Service Pre Process');
  if (
    options.containsKey('PlatformEventEnabled') &&
    ((Boolean) options.get('PlatformEventEnabled')) == true
  )
    publishPlatformEventIntegration('pre', JSON.serialize(input));
  return input;
}
```

invokeProcess(Map<String, Object> input, Map<String, Object> output, Map<String, Object> options)

Descripción

Método principal que ejecuta la lógica de integración con el servicio externo para la generación de URLs. Maneja la configuración, validación, comunicación con el servicio externo y procesamiento de respuestas.

Parámetros

- `input` (Map<String, Object>): Datos de entrada para el procesamiento.
- `output` (Map<String, Object>): Mapa de salida (no utilizado en la implementación actual).
- `options` (Map<String, Object>): Opciones de configuración para el procesamiento.

Retorno

- `Map<String, Object>`: Respuesta procesada del servicio externo, o `null` en caso de error.

Proceso

1. Obtiene la configuración desde metadatos personalizados.

2. Configura las opciones de eventos de plataforma.

3. Ejecuta el pre-procesamiento.

4. Realiza la llamada HTTP al servicio externo.

5. Procesa la respuesta y la estructura en un formato estándar.

6. Publica eventos de plataforma con los resultados.

7. Ejecuta el post-procesamiento.

8. Maneja excepciones y devuelve resultados.

Código

```
global Map<String, Object> invokeProcess(
  Map<String, Object> input,
  Map<String, Object> output,
  Map<String, Object> options
) {
  // Obtener configuración de metadatos
  SF_IntegrationSetting__mdt config = Test.isRunningTest()
    ? ExternalServiceMetadataIntegrationMock.getIntegrationSettings(SURATECH_MDT_CONFIG_NAME)
    : SF_IntegrationSetting__mdt.getInstance(SURATECH_MDT_CONFIG_NAME);

  if (config == null) {
    throw new ExternalServiceException(
      'No existing "' + SURATECH_MDT_CONFIG_NAME +
      '" custom metadata type record at Integration Setting'
    );
  }

  // Configurar opciones
  options.put('PlatformEventEnabled', config.Platform_Events_Enabled__c);

  ExternalNotificationHandler enh = new ExternalNotificationHandler();
  Map<String, Object> payload = new Map<String, Object>();

  Boolean ignoreHash = options.containsKey('ignoreHash')
    ? (Boolean) options.get('ignoreHash')
    : false;

  try {
    payload = this.preProcess(input, options);
    Core.debug('Starting URL Generation Integration Service In Process');
    publishPlatformEventIntegration('in', JSON.serialize(input));

    // Realizar llamada HTTP
    System.HTTPResponse response = enh.emit(
      SERVICE_METHOD,
      config,
      '',
      (Object) payload
    );

    // Procesar respuesta
    Map<String, Object> mapResponse = new Map<String, Object>{
      'body' => (Map<String, Object>) JSON.deserializeUntyped(response.getBody()),
```

```
      'status' => response.getStatus(),
      'statusCode' => response.getStatusCode()
    };

    String responseString = response.getBody();
    System.debug('Response String: ' + responseString);

    this.publishPlatFormEvent(
      response.getStatusCode(),
      JSON.serialize(payload),
      JSON.serialize(response.getBody())
    );

    return this.postProcess(mapResponse, options);
  } catch (Exception e) {
    enh.controlException(
      'Generate URL remote service',
      e,
      JSON.serialize(payload)
    );
    return null;
  }
}
```

postProcess(Map<String, Object> response, Map<String, Object> options)

Descripción

Método virtual que se ejecuta después del procesamiento principal. Realiza tareas de finalización y opcionalmente publica eventos de plataforma con la respuesta final.

Parámetros

- `response` (Map<String, Object>): Respuesta del procesamiento principal.
- `options` (Map<String, Object>): Opciones de configuración.

Retorno

- `Map<String, Object>`: La respuesta, potencialmente modificada durante el post-procesamiento.

Código

```
global virtual Map<String, Object> postProcess(
  Map<String, Object> response,
  Map<String, Object> options
) {
  Core.debug('Starting Generate URL Integration Service Post Process');
  if (
    options.containsKey('PlatformEventEnabled') &&
    ((Boolean) options.get('PlatformEventEnabled')) == true
  )
    publishPlatformEventIntegration('post', JSON.serialize(response));
  return response;
}
```

publishPlatformEventIntegration(String moment, String data)

Descripción

Método que publica eventos de plataforma en diferentes momentos del procesamiento para proporcionar visibilidad y auditoría del proceso de integración.

Parámetros

- `moment` (String): Momento del procesamiento ('pre', 'in', 'post').
- `data` (String): Datos a incluir en el evento de plataforma.

Retorno

- `void`: Este método no devuelve ningún valor.

Proceso

1. Evalúa el momento especificado usando un switch.
2. Llama al método de publicación de evento apropiado según el momento.
3. Incluye información contextual específica para cada momento.

Código

```
global void publishPlatformEventIntegration(String moment, String data) {
  switch on moment.toLowerCase() {
    when 'pre' {
      publishInitPrePlatformEvent(data, 'Generate URL Integration Service');
    }
    when 'post' {
      publishInitPosPlatformEvent(data, 'Generate URL Integration Service');
    }
    when 'in' {
      publishPlatFormEvent(200, data, 'Starting Generate URL In process');
    }
  }
}
```

Dependencias

- **`PlatformEventIntegrationObserver`**: Clase base que proporciona funcionalidad de eventos de plataforma.
- **`SF_IntegrationSetting__mdt`**: Tipo de metadatos personalizados que almacena la configuración de integración.
- **`ExternalNotificationHandler`**: Clase que maneja las comunicaciones HTTP con servicios externos.
- **`ExternalServiceException`**: Excepción personalizada para errores de servicios externos.
- **`ExternalServiceMetadataIntegrationMock`**: Clase mock para proporcionar configuración durante las pruebas.
- **`Core`**: Clase utilitaria que proporciona funcionalidad de depuración.

Patrón de Diseño Implementado

Template Method

La clase implementa el patrón Template Method a través de los métodos `preProcess`, `invokeProcess` y `postProcess`, definiendo un algoritmo de procesamiento en pasos donde algunos pasos pueden ser sobrescritos por subclases.

Observer

A través de la herencia de `PlatformEventIntegrationObserver`, la clase implementa el patrón Observer para la publicación de eventos de plataforma.

Ejemplos de Uso

Ejemplo 1: Uso Básico del Servicio

```
public class URLGenerationService {

    public static Map<String, Object> generatePaymentURL(Map<String, Object> paymentData) {
        try {
            // Crear instancia del handler
            GenerateURLServiceHandler handler = new GenerateURLServiceHandler();

            // Preparar opciones
            Map<String, Object> options = new Map<String, Object>{
                'PlatformEventEnabled' => true,
                'ignoreHash' => false
            };

            // Preparar datos de entrada
            Map<String, Object> input = new Map<String, Object>{
                'amount' => paymentData.get('amount'),
                'currency' => paymentData.get('currency'),
                'customerInfo' => paymentData.get('customerInfo'),
                'returnUrl' => paymentData.get('returnUrl')
            };

            // Ejecutar el procesamiento
            Map<String, Object> result = handler.invokeProcess(input, new Map<String, Object>(), options);

            if (result != null && result.containsKey('body')) {
                Map<String, Object> responseBody = (Map<String, Object>) result.get('body');
                return new Map<String, Object>{
                    'success' => true,
                    'paymentUrl' => responseBody.get('url'),
                    'transactionId' => responseBody.get('transactionId')
                };
            } else {
                return new Map<String, Object>{
                    'success' => false,
                    'error' => 'No se pudo generar la URL de pago'
                };
            }
        } catch (Exception e) {
            System.debug('Error generando URL de pago: ' + e.getMessage());
            return new Map<String, Object>{
                'success' => false,
                'error' => e.getMessage()
            };
        }
    }
}

// Uso del servicio
Map<String, Object> paymentData = new Map<String, Object>{
    'amount' => 150000,
```

```
    'currency' => 'COP',
    'customerInfo' => new Map<String, Object>{
        'name' => 'Juan Pérez',
        'email' => 'juan@example.com',
        'phone' => '3001234567'
    },
    'returnUrl' => 'https://mysite.com/payment/return'
};

Map<String, Object> result = URLGenerationService.generatePaymentURL(paymentData);

if ((Boolean) result.get('success')) {
    String paymentUrl = (String) result.get('paymentUrl');
    System.debug('URL de pago generada: ' + paymentUrl);
} else {
    System.debug('Error: ' + result.get('error'));
}
```

Ejemplo 2: Extensión de la Clase para Funcionalidad Específica

```
public class EnhancedURLServiceHandler extends GenerateURLServiceHandler {

    // Sobrescribir pre-procesamiento para validaciones adicionales
    global override Map<String, Object> preProcess(
        Map<String, Object> input,
        Map<String, Object> options
    ) {
        Core.debug('Starting Enhanced URL Service Pre Process');

        // Ejecutar validaciones personalizadas
        validateInputData(input);

        // Enriquecer datos de entrada
        input = enrichInputData(input);

        // Llamar al método padre para mantener funcionalidad base
        return super.preProcess(input, options);
    }

    // Sobrescribir post-procesamiento para transformaciones adicionales
    global override Map<String, Object> postProcess(
        Map<String, Object> response,
        Map<String, Object> options
    ) {
        Core.debug('Starting Enhanced URL Service Post Process');

        // Procesar respuesta
        response = transformResponse(response);

        // Crear registros de auditoría
        createAuditRecord(response, options);

        // Llamar al método padre
        return super.postProcess(response, options);
    }

    private void validateInputData(Map<String, Object> input) {
```

```
        if (!input.containsKey('amount') || input.get('amount') == null) {
            throw new CumuloMissingParameterException('El monto es requerido para generar la URL de pago');
        }

        Decimal amount = (Decimal) input.get('amount');
        if (amount <= 0) {
            throw new IllegalArgumentException('El monto debe ser mayor que cero');
        }

        if (!input.containsKey('customerInfo')) {
            throw new CumuloMissingParameterException('La información del cliente es requerida');
        }
    }

    private Map<String, Object> enrichInputData(Map<String, Object> input) {
        // Añadir timestamp
        input.put('timestamp', System.now().getTime());

        // Añadir información de la organización
        input.put('organizationId', UserInfo.getOrganizationId());

        // Generar ID de transacción único
        input.put('transactionRef', generateTransactionReference());

        return input;
    }

    private Map<String, Object> transformResponse(Map<String, Object> response) {
        if (response.containsKey('body')) {
            Map<String, Object> body = (Map<String, Object>) response.get('body');

            // Añadir información adicional a la respuesta
            body.put('processedAt', System.now());
            body.put('processedBy', UserInfo.getUserId());

            response.put('body', body);
        }

        return response;
    }

    private void createAuditRecord(Map<String, Object> response, Map<String, Object> options) {
        try {
            // Crear registro de auditoría en objeto personalizado
            URL_Generation_Audit__c audit = new URL_Generation_Audit__c(
                Response_Status_Code__c = (Integer) response.get('statusCode'),
                Response_Status__c = (String) response.get('status'),
                Platform_Events_Enabled__c = (Boolean) options.get('PlatformEventEnabled'),
                Processing_Date__c = System.now()
            );

            insert audit;
        } catch (Exception e) {
            System.debug('Error creando registro de auditoría: ' + e.getMessage());
        }
    }
```

```
    private String generateTransactionReference() {
        return 'TXN_' + String.valueOf(System.currentTimeMillis()) + '_' +
               String.valueOf(Math.random()).substring(2, 8);
    }
}
```

Ejemplo 3: Uso en Procedimiento de Integración

```
public class PaymentIntegrationProcedure {

    public static Map<String, Object> generatePaymentURL(Map<String, Object> input) {
        try {
            // Extraer datos del input del procedimiento de integración
            Map<String, Object> paymentInfo = (Map<String, Object>) input.get('paymentInfo');
            Map<String, Object> options = (Map<String, Object>) input.get('options');

            if (options == null) {
                options = new Map<String, Object>{
                    'PlatformEventEnabled' => true
                };
            }

            // Crear handler y procesar
            GenerateURLServiceHandler handler = new GenerateURLServiceHandler();
            Map<String, Object> result = handler.invokeProcess(
                paymentInfo,
                new Map<String, Object>(),
                options
            );

            // Estructurar respuesta para el procedimiento de integración
            if (result != null) {
                return new Map<String, Object>{
                    'success' => true,
                    'data' => result,
                    'message' => 'URL de pago generada exitosamente'
                };
            } else {
                return new Map<String, Object>{
                    'success' => false,
                    'data' => null,
                    'message' => 'Error al generar URL de pago'
                };
            }
        } catch (ExternalServiceException e) {
            return new Map<String, Object>{
                'success' => false,
                'data' => null,
                'message' => 'Error de servicio externo: ' + e.getMessage(),
                'errorType' => 'EXTERNAL_SERVICE_ERROR'
            };
        } catch (Exception e) {
            return new Map<String, Object>{
                'success' => false,
                'data' => null,
                'message' => 'Error inesperado: ' + e.getMessage(),
                'errorType' => 'UNEXPECTED_ERROR'
```

```
            };
        }
    }
}
```

Configuración de Metadatos Personalizados

La clase utiliza el registro de metadatos personalizados `SF_IntegrationSetting__mdt` con el nombre `url_endpoint`. Este registro debe contener:

Campo	Descripción
`Platform_Events_Enabled__c`	Boolean que indica si los eventos de plataforma están habilitados
Campos de endpoint	URL del servicio externo, credenciales, timeouts, etc.

Ejemplo de configuración:

```
Label: URL Endpoint Configuration
Developer Name: url_endpoint
Platform_Events_Enabled__c: true
Endpoint__c: https://api.payment-provider.com/v1/generate-url
Timeout__c: 30000
API_Key__c: [encrypted]
```

Mejores Prácticas Implementadas

1. **Separación de Responsabilidades**: La clase separa claramente las fases de pre-procesamiento, procesamiento principal y post-procesamiento.
2. **Configuración Externalizada**: Utiliza metadatos personalizados para la configuración, permitiendo cambios sin modificar código.
3. **Manejo de Errores**: Implementa manejo centralizado de excepciones.
4. **Observabilidad**: Proporciona eventos de plataforma para monitoreo y auditoría.
5. **Testabilidad**: Incluye soporte para mocks durante las pruebas.

Consideraciones de Seguridad

1. **Datos Sensibles**: La clase serializa datos de entrada y respuesta para eventos de plataforma. Asegurar que no se expongan datos sensibles.
2. **Configuración de Metadatos**: Las credenciales y endpoints deben estar adecuadamente protegidos en los metadatos personalizados.
3. **Validación de Entrada**: Aunque hay comentarios sobre validación de esquema, la implementación actual no incluye validaciones robustas.

Consideraciones de Rendimiento

1. **Llamadas HTTP**: Las operaciones son síncronas y pueden afectar el tiempo de respuesta total.
2. **Eventos de Plataforma**: La publicación de múltiples eventos puede tener un impacto en el rendimiento.
3. **Serialización JSON**: La serialización repetida de objetos grandes puede ser costosa.

Notas Adicionales

1. **Código Comentado**: La clase contiene código comentado que sugiere funcionalidades planeadas o alternativas (como `INTEGRATION_PROCEDURE_NAME` y `SERVICE_NAME`).
2. **Parámetro Ignorado**: El parámetro `ignoreHash` se lee pero no se utiliza en la lógica actual.
3. **Métodos Virtuales**: Los métodos `preProcess` y `postProcess` son virtuales, permitiendo su sobrescritura en subclases.
4. **Autor Modificado**: El campo "last modified by" muestra un placeholder, sugiriendo un proceso de configuración incompleto.
5. **Flexibilidad**: La naturaleza virtual de la clase permite extensión y personalización para diferentes tipos de servicios de generación de URL.

ExternalServiceValidationUtil

Descripción General

`ExternalServiceValidationUtil` es una clase de utilidad que proporciona métodos para validar datos utilizados en servicios externos, específicamente enfocada en la validación de placas de vehículos y el registro de eventos de integración.

Autor

Felipe Correa \<felipe.correa@nespon.com\>

Proyecto

Foundation Salesforce - Suratech - UH-17585

Última Modificación

17 de enero de 2025

Estructura y Métodos

Validación de Placas

plateValidation(String plate)

Descripción

Método público que procesa y valida una placa de vehículo. Convierte la placa a mayúsculas y luego realiza la validación.

Parámetros

- `plate` (String): La placa del vehículo a validar.

Excepciones

- `ExternalServiceValidationException`: Si la placa no cumple con los criterios de validación.

Código

```
public static void plateValidation(String plate) {
  plate = parseUpperCase(plate);
  validatePlate(plate);
}
```

parseUpperCase(String param)

Descripción

Convierte una cadena a mayúsculas.

Parámetros

- `param` (String): La cadena a convertir.

Retorno

- `String`: La cadena convertida a mayúsculas.

Código

```
public static String parseUpperCase(String param) {
  return param.toUpperCase();
}
```

validatePlate(String plate)

Descripción

Valida una placa de vehículo según criterios específicos:

- No puede ser nula o vacía
- Debe tener exactamente 6 caracteres
- Debe seguir un patrón específico para automóviles o motocicletas

Parámetros

- `plate` (String): La placa del vehículo a validar.

Excepciones

- `ExternalServiceValidationException`: Si la placa no cumple con alguno de los criterios.

Código

```
public static void validatePlate(String plate) {
  if (plate == null || String.isEmpty(plate))
    throw new ExternalServiceValidationException(
      'The plate cannot be empty or Null!'
    );
  else if (plate.length() != 6) {
    throw new ExternalServiceValidationException(
      'The plate must contain 6 characters! Actual Length = ' + plate.length()
    );
  } else if (!platePattern(plate)) {
    throw new ExternalServiceValidationException(
      'The plate format must be correct!'
    );
  }
}
```

platePattern(String plate)

Descripción

Método privado que verifica si una placa sigue uno de los patrones definidos para automóviles o motocicletas utilizando expresiones regulares.

Parámetros

- `plate` (String): La placa a validar.

Retorno

- `Boolean`: `true` si la placa sigue alguno de los patrones definidos, `false` en caso contrario.

Patrones

- **Automóviles**: `(^[A-Z]([A-Z0-9][0-9]|[A-Z]{2})[0-9]{2}[A-Z0-9]$)`
  - Comienza con una letra mayúscula
  - Seguida por una letra mayúscula o número y un número, o dos letras mayúsculas
  - Seguida por dos números
  - Termina con una letra mayúscula o número
- **Motocicletas**: `(^[A-Z][0-9]{4}[A-Z0-9]$)`
  - Comienza con una letra mayúscula
  - Seguida por cuatro números
  - Termina con una letra mayúscula o número

Código

```
private static boolean platePattern(String plate) {
  String carPattern = '(^[A-Z]([A-Z0-9][0-9]|[A-Z]{2})[0-9]{2}[A-Z0-9]$)';
  String motoPattern = '(^[A-Z][0-9]{4}[A-Z0-9]$)';
  // return validatePattern(plate, carPattern+'|'+motoPattern);
  return Pattern.matches(carPattern + '|' + motoPattern, plate);
}
```

Registro de Eventos

publishServiceEventLog(String type, String details)

Descripción

Publica un registro de evento relacionado con servicios externos si la función de registros de eventos está habilitada a través de `FeatureFlags`.

Parámetros

- `type` (String): El tipo o nivel del evento (por ejemplo, 'ERROR', 'INFO', etc.).
- `details` (String): Detalles del evento a registrar.

Proceso

1. Verifica si la función `sura_foundation_event_logs` está habilitada.
2. Si está habilitada, registra un evento asincrónico con información sobre el servicio externo.

Código

```
public static void publishServiceEventLog(String type, String details) {
  if (new FeatureFlags().evaluate('sura_foundation_event_logs').isEnabled()) {
    Core.debug('Saving ' + type + ' Log...');
    EventLogManager.saveAsync(
      JSON.serialize(
        new List<Map<String, Object>>{
          new Map<String, Object>{
```

```
            'Product' => 'All',
            'Level' => type,
            'Platform' => 'Salesforce',
            'Details' => details,
            'Process' => 'External Service',
            'Type' => 'Integración'
          }
        }
      )
    );
  }
}
```

Dependencias

- `ExternalServiceValidationException`: Excepción personalizada lanzada cuando se incumplen los criterios de validación.
- `FeatureFlags`: Clase que gestiona la habilitación de características a través de indicadores.
- `Core`: Clase de utilidad que proporciona el método `debug` para registro.
- `EventLogManager`: Gestor que proporciona funcionalidades para guardar registros de eventos.

Ejemplo de Uso

Validación de Placa

```
try {
    // Validar una placa de automóvil
    ExternalServiceValidationUtil.plateValidation('ABC123');
    System.debug('La placa es válida.');

    // Proceder con operaciones relacionadas con la placa
    // ...
} catch (ExternalServiceValidationException e) {
    // Manejar error de validación
    System.debug('Error de validación: ' + e.getMessage());

    // Registrar el error
    ExternalServiceValidationUtil.publishServiceEventLog('ERROR', e.getMessage());

    // Informar al usuario
    ApexPages.addMessage(new ApexPages.Message(
        ApexPages.Severity.ERROR,
        'La placa proporcionada no es válida: ' + e.getMessage()
    ));
}
```

Publicación de Evento de Servicio

```
// Registrar un evento informativo
ExternalServiceValidationUtil.publishServiceEventLog(
    'INFO',
    'Consulta exitosa a servicio externo de validación de placas. Placa: ABC123'
);

// Registrar un evento de error
```

```
try {
    // Operación del servicio externo
    // ...
} catch (Exception e) {
    ExternalServiceValidationUtil.publishServiceEventLog(
        'ERROR',
        'Error en servicio externo: ' + e.getMessage()
    );
}
```

Patrones Válidos de Placas

Automóviles (carPattern)

- **Patrón**: `(^[A-Z]([A-Z0-9][0-9]|[A-Z]{2})[0-9]{2}[A-Z0-9]$)`
- **Ejemplos válidos**:
  - `ABC123`: Primera letra (A) + dos letras (BC) + dos números (12) + letra/número (3)
  - `A1D234`: Primera letra (A) + letra/número y número (1D) + dos números (23) + letra/número (4)

Motocicletas (motoPattern)

- **Patrón**: `(^[A-Z][0-9]{4}[A-Z0-9]$)`
- **Ejemplos válidos**:
  - `A1234B`: Primera letra (A) + cuatro números (1234) + letra/número (B)
  - `Z9876X`: Primera letra (Z) + cuatro números (9876) + letra/número (X)

Notas Adicionales

1. La clase contiene código comentado que sugiere una implementación alternativa para la validación de patrones y el guardado sincrónico de eventos. Estas líneas están actualmente inactivas.
2. El método `publishServiceEventLog` utiliza `saveAsync` para guardar los eventos de manera asincrónica, lo que optimiza el rendimiento al no bloquear la transacción actual.
3. La validación de placas está diseñada específicamente para un formato que tiene 6 caracteres, siguiendo patrones diferentes para automóviles y motocicletas.
4. La clase utiliza regiones de código (`#region`) para organizar lógicamente los métodos relacionados, mejorando la legibilidad del código.
5. El uso de `FeatureFlags` permite habilitar o deshabilitar el registro de eventos sin modificar el código, facilitando los despliegues y pruebas.

Posibles Mejoras

1. Añadir documentación JavaDoc completa a todos los métodos.
2. Implementar validaciones adicionales para otros tipos de datos utilizados en servicios externos.
3. Proporcionar métodos de utilidad para validar y formatear otros tipos comunes de información (como números de identificación, correos electrónicos, etc.).

4. Implementar métodos para validar placas de diferentes longitudes o formatos, para adaptarse a posibles cambios en los requisitos de formato.
5. Añadir tests unitarios exhaustivos para cubrir todos los casos posibles de validación.

ExternalServiceValidationException

Descripción General

`ExternalServiceValidationException` es una clase de excepción personalizada que extiende la clase base `Exception` de Apex. Esta excepción está diseñada específicamente para representar errores de validación que ocurren durante la interacción con servicios externos.

Estructura de la Clase

```
public class ExternalServiceValidationException extends Exception {
}
```

Características

Herencia

La clase `ExternalServiceValidationException` hereda de la clase base `Exception` de Apex, lo que significa que hereda todos los métodos y propiedades estándar de las excepciones de Apex.

Accesibilidad

- **Modificador de Acceso**: `public`
- **Descripción**: La excepción es accesible desde cualquier contexto dentro de la organización.

Funcionalidad

Esta excepción personalizada proporciona una forma específica de indicar errores relacionados con la validación de servicios externos. Al utilizar una clase de excepción dedicada, el código que interactúa con servicios externos puede capturar y manejar específicamente los errores de validación, separándolos de otros tipos de excepciones.

Métodos Heredados

Al extender la clase base `Exception`, esta clase hereda los siguientes métodos importantes:

getMessage()

- **Retorno**: `String`
- **Descripción**: Devuelve el mensaje de error asociado con la excepción.

getTypeName()

- **Retorno**: `String`
- **Descripción**: Devuelve el nombre completo de la clase de excepción.

getCause()

- **Retorno**: `Exception`
- **Descripción**: Devuelve la excepción causante, si existe.

getLineNumber()

- **Retorno**: `Integer`

- **Descripción**: Devuelve el número de línea donde se produjo la excepción.

getStackTraceString()

- **Retorno**: `String`
- **Descripción**: Devuelve la traza de la pila como una cadena.

Constructores Implícitos

Aunque no están explícitamente definidos en el código, la clase hereda los siguientes constructores de la clase base `Exception`:

ExternalServiceValidationException()

- **Descripción**: Constructor por defecto que crea una instancia de la excepción sin mensaje.

ExternalServiceValidationException(String message)

- **Descripción**: Constructor que crea una instancia de la excepción con un mensaje específico.
- **Parámetros**:
  - `message` (String): Mensaje descriptivo de la excepción.

ExternalServiceValidationException(Exception cause)

- **Descripción**: Constructor que crea una instancia de la excepción con una excepción causante.
- **Parámetros**:
  - `cause` (Exception): Excepción causante.

ExternalServiceValidationException(String message, Exception cause)

- **Descripción**: Constructor que crea una instancia de la excepción con un mensaje y una excepción causante.
- **Parámetros**:
  - `message` (String): Mensaje descriptivo de la excepción.
  - `cause` (Exception): Excepción causante.

Ejemplo de Uso

```
public class ExternalServiceConnector {

    /**
     * @description Valida y procesa datos con un servicio externo
     * @param data Los datos a procesar
     * @return Los datos procesados
     * @throws ExternalServiceValidationException Si los datos no pasan la validación
     */
    public Map<String, Object> processWithExternalService(Map<String, Object> data) {
        // Validar los datos antes de enviarlos al servicio externo
        if (!isValidForExternalService(data)) {
            throw new ExternalServiceValidationException(
                'Los datos proporcionados no cumplen con los requisitos del servicio externo');
        }
```

```
        try {
            // Código para interactuar con el servicio externo
            HttpRequest req = new HttpRequest();
            req.setEndpoint('https://api.ejemplo.com/procesar');
            req.setMethod('POST');
            req.setHeader('Content-Type', 'application/json');
            req.setBody(JSON.serialize(data));

            Http http = new Http();
            HttpResponse res = http.send(req);

            // Validar la respuesta
            if (res.getStatusCode() != 200) {
                Map<String, Object> errorResponse = (Map<String,
Object>)JSON.deserializeUntyped(res.getBody());
                throw new ExternalServiceValidationException(
                    'Error del servicio externo: ' + errorResponse.get('error_message'));
            }

            return (Map<String, Object>)JSON.deserializeUntyped(res.getBody());
        } catch (Exception e) {
            if (e instanceof ExternalServiceValidationException) {
                throw e; // Relanzar excepciones de validación
            }
            throw new ExternalServiceValidationException(
                'Error al procesar con el servicio externo: ' + e.getMessage(), e);
        }
    }

    /**
     * @description Valida si los datos cumplen con los requisitos del servicio externo
     * @param data Los datos a validar
     * @return true si los datos son válidos, false en caso contrario
     */
    private Boolean isValidForExternalService(Map<String, Object> data) {
        // Implementación de la lógica de validación
        if (data == null || data.isEmpty()) {
            return false;
        }

        // Verificar campos requeridos
        String[] requiredFields = new String[] {'id', 'nombre', 'tipo'};
        for (String field : requiredFields) {
            if (!data.containsKey(field) || data.get(field) == null) {
                return false;
            }
        }

        return true;
    }
}
```

Manejo de la Excepción

```
try {
    Map<String, Object> data = new Map<String, Object> {
```

```
        'id' => '12345',
        'nombre' => 'Ejemplo',
        // Falta el campo 'tipo'
    };

    ExternalServiceConnector connector = new ExternalServiceConnector();
    Map<String, Object> result = connector.processWithExternalService(data);

    // Procesar el resultado exitoso
    System.debug('Procesamiento exitoso: ' + result);
} catch (ExternalServiceValidationException e) {
    // Manejar específicamente los errores de validación
    System.debug('Error de validación: ' + e.getMessage());

    // Registrar el error
    LoggerUtility.logError('ExternalServiceValidation', e.getMessage(), e.getStackTraceString());

    // Notificar al usuario con un mensaje amigable
    ApexPages.addMessage(new ApexPages.Message(
        ApexPages.Severity.ERROR,
        'No se pudo completar la operación debido a un problema de validación con el servicio externo. ' +
        'Por favor, verifique los datos proporcionados.'));
} catch (Exception e) {
    // Manejar otros tipos de excepciones
    System.debug('Error general: ' + e.getMessage());
    LoggerUtility.logError('ExternalServiceGeneral', e.getMessage(), e.getStackTraceString());
}
```

Buenas Prácticas

1. **Mensajes descriptivos**: Al lanzar una `ExternalServiceValidationException`, proporcione mensajes descriptivos que ayuden a identificar la causa exacta del error de validación.
2. **Captura específica**: Capture específicamente esta excepción cuando sea relevante para el contexto de validación de servicios externos.
3. **Jerarquía de excepciones**: Considere crear una jerarquía de excepciones más detallada si necesita distinguir entre diferentes tipos de errores de validación de servicios externos.
4. **Encapsular excepciones originales**: Cuando sea apropiado, encapsule la excepción original como causa al lanzar una `ExternalServiceValidationException`, para mantener la información completa del error.
5. **Registro adecuado**: Asegúrese de registrar adecuadamente las excepciones para facilitar la depuración y el análisis de problemas.

Notas Adicionales

1. Esta excepción personalizada ayuda a mejorar la legibilidad y mantenibilidad del código al proporcionar un tipo específico para los errores de validación de servicios externos.
2. Aunque la implementación actual es mínima, la clase podría ampliarse en el futuro para incluir propiedades adicionales específicas de validación de servicios externos, como códigos de error, detalles de campo, o información sobre el servicio externo específico.
3. En una arquitectura compleja que interactúe con múltiples servicios externos, podría ser beneficioso extender esta excepción base para crear excepciones más específicas para cada servicio.

ExternalServiceException

Descripción General

`ExternalServiceException` es una clase de excepción personalizada que extiende la clase `Exception` estándar de Apex. Esta clase está diseñada para representar errores específicos que ocurren durante la interacción con servicios externos.

Estructura

```
public class ExternalServiceException extends Exception {

}
```

La clase no define métodos o propiedades adicionales más allá de los heredados de la clase base `Exception`.

Herencia

- **Clase Base**: `Exception`
- **Paquete**: Estándar de Salesforce

Propósito

El propósito principal de esta clase es proporcionar un tipo de excepción específico para errores relacionados con servicios externos, permitiendo:

1. **Categorización de Errores**: Distinguir errores de servicios externos de otros tipos de excepciones en el sistema.
2. **Manejo Específico**: Permitir que el código que captura excepciones pueda identificar y manejar específicamente errores de servicios externos.
3. **Claridad en el Código**: Mejorar la legibilidad del código al indicar explícitamente cuando un error está relacionado con un servicio externo.

Métodos Heredados

Al extender la clase `Exception` estándar, `ExternalServiceException` hereda los siguientes métodos y propiedades:

Constructores

- **`ExternalServiceException()`**: Constructor por defecto.
- **`ExternalServiceException(String message)`**: Constructor que acepta un mensaje de error.
- **`ExternalServiceException(Exception cause)`**: Constructor que acepta otra excepción como causa.
- **`ExternalServiceException(String message, Exception cause)`**: Constructor que acepta un mensaje y otra excepción como causa.

Métodos

- **`getMessage()`**: Devuelve el mensaje de error asociado con la excepción.
- **`getCause()`**: Devuelve la excepción que causó esta excepción, si existe.
- **`getTypeName()`**: Devuelve el nombre del tipo de excepción.
- **`setMessage(String message)`**: Establece el mensaje de error para la excepción.

- `getLineNumber()`: Devuelve el número de línea donde se generó la excepción.
- `getStackTraceString()`: Devuelve una representación de texto del seguimiento de la pila.

Uso en el Sistema

Esta excepción es utilizada por varias clases en el sistema que interactúan con servicios externos, tales como:

1. `ExternalNotificationHandler`: La utiliza para envolver excepciones específicas durante las llamadas a servicios externos.
2. `RUNTServiceHandler`: La lanza cuando hay problemas para obtener la configuración del servicio RUNT.
3. **Otras clases de integración**: Cualquier clase que realice llamadas a servicios web, APIs externas o sistemas integrados.

Ejemplo de Lanzamiento

```
try {
    // Intentar realizar una operación con un servicio externo
    HttpRequest req = new HttpRequest();
    req.setEndpoint('https://api.example.com/resource');
    req.setMethod('GET');

    Http http = new Http();
    HttpResponse res = http.send(req);

    if (res.getStatusCode() != 200) {
        throw new ExternalServiceException(
            'Error en el servicio externo. Código: ' + res.getStatusCode() +
            '. Respuesta: ' + res.getBody()
        );
    }

    // Procesar la respuesta exitosa
} catch (CalloutException e) {
    throw new ExternalServiceException(
        'Error de comunicación con el servicio externo: ' + e.getMessage(),
        e
    );
} catch (Exception e) {
    // Manejar otras excepciones
    throw new ExternalServiceException(
        'Error inesperado al interactuar con el servicio externo: ' + e.getMessage(),
        e
    );
}
```

Ejemplo de Captura

```
try {
    // Llamar a un método que podría lanzar diferentes tipos de excepciones
    processExternalServiceRequest();
} catch (ExternalServiceException e) {
    // Manejo específico para errores de servicios externos
    System.debug('Error de servicio externo: ' + e.getMessage());

    // Registrar el error en un objeto personalizado
```

```
    External_Service_Error__c errorRecord = new External_Service_Error__c(
        Error_Message__c = e.getMessage(),
        Stack_Trace__c = e.getStackTraceString(),
        Timestamp__c = Datetime.now()
    );
    insert errorRecord;

    // Notificar al administrador
    sendErrorNotification('Error de servicio externo', e.getMessage());
} catch (Exception e) {
    // Manejo genérico para otros tipos de excepciones
    System.debug('Error general: ' + e.getMessage());
}
```

Extensión Personalizada

Aunque la implementación actual es minimalista, la clase puede ser extendida para incluir información adicional específica de servicios externos:

```
public class EnhancedExternalServiceException extends ExternalServiceException {
    private Integer statusCode;
    private String endpoint;
    private String requestBody;
    private String responseBody;

    public EnhancedExternalServiceException(String message) {
        super(message);
    }

    public EnhancedExternalServiceException(String message, Integer statusCode, String endpoint) {
        super(message);
        this.statusCode = statusCode;
        this.endpoint = endpoint;
    }

    public EnhancedExternalServiceException(
        String message,
        Integer statusCode,
        String endpoint,
        String requestBody,
        String responseBody
    ) {
        super(message);
        this.statusCode = statusCode;
        this.endpoint = endpoint;
        this.requestBody = requestBody;
        this.responseBody = responseBody;
    }

    public Integer getStatusCode() {
        return this.statusCode;
    }

    public String getEndpoint() {
        return this.endpoint;
    }
```

```
    public String getRequestBody() {
        return this.requestBody;
    }

    public String getResponseBody() {
        return this.responseBody;
    }
}
```

Mejores Prácticas

1. **Mensajes Descriptivos**: Incluir información específica en los mensajes de error, como códigos de estado, endpoints y respuestas.
2. **Captura Selectiva**: Capturar esta excepción específicamente cuando se quiera manejar errores de servicios externos de manera diferenciada.
3. **Información de Contexto**: Al lanzar la excepción, proporcionar información de contexto que pueda ayudar a diagnosticar el problema.
4. **Encadenamiento de Excepciones**: Utilizar el constructor que acepta una excepción causa para preservar el seguimiento de la pila original.
5. **Registro Adecuado**: Registrar las excepciones capturadas para su posterior análisis y solución de problemas.

Puntos a Considerar

1. **Simplicidad vs. Especialización**: La implementación actual es muy simple, lo que la hace flexible pero menos especializada. Considerar si se necesita una versión más especializada con campos adicionales.
2. **Seguridad**: Tener cuidado con la información sensible que se incluye en los mensajes de excepción, especialmente si estos se muestran a los usuarios finales.
3. **Rendimiento**: La creación de objetos de excepción tiene un costo, por lo que deben utilizarse apropiadamente y no como mecanismo de control de flujo normal.

Notas Adicionales

1. Esta clase forma parte del ecosistema de integración de la organización y se utiliza en conjunto con otras clases como `ExternalNotificationHandler` y `RUNTServiceHandler`.
2. La simplicidad de la implementación sugiere que la clase se utiliza principalmente para categorización de errores más que para transportar información adicional específica.
3. En un entorno de producción, considerar la implementación de mecanismos de registro y notificación automatizados para errores de servicios externos.

ExternalNotificationHandler

Descripción General

`ExternalNotificationHandler` es una clase virtual que extiende `PlatformEventIntegrationObserver` y está diseñada para manejar notificaciones a servicios externos. Esta clase implementa un mecanismo sofisticado de llamadas HTTP a servicios externos con capacidades de reintento, control de tiempos de espera, y registro de eventos de plataforma para monitoreo y depuración.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

9 de mayo de 2025 por Esneyder Zabala

Constantes y Variables Estáticas

Nombre	Tipo	Descripción
`NAMED_CREDENTIAL`	`String`	Nombre de la credencial con nombre utilizada para autenticación: 'Suratech_Integration_Bus_Credentials'
`skipLogDefault`	`Boolean`	Controla si se deben registrar logs de eventos. Valor por defecto: `false`
`retryQuantity`	`Integer`	Número de reintentos realizados. Valor inicial: `1`
`familyProductControl`	`String`	Familia de producto para el control de logs. Valor por defecto: 'All'
`accumulatedRetryTime`	`Long`	Tiempo acumulado en milisegundos para los reintentos
`MAX_RETRY_TIME`	`Long`	Tiempo máximo permitido para los reintentos, obtenido desde un metadata personalizado
`bodyInput`	`String`	Almacena el cuerpo de la solicitud para su uso en logs

Variables de Instancia

Nombre	Tipo	Descripción
`startTimeGlobal`	`Long`	Tiempo de inicio global para medir el tiempo total de ejecución

Inicialización Estática

La clase inicializa `MAX_RETRY_TIME` a través de un bloque estático que obtiene el valor desde un registro de metadatos personalizado `SF_ConfigTimeRetryIntegrations__mdt`.

```
static {
  MAX_RETRY_TIME = getMaxRetryTimeFromMetadata();
}
```

Métodos de Acceso (Getters y Setters)

Método	Descripción
`setBodyInput(String value)`	Establece el valor del cuerpo de la solicitud para su uso en logs
`getBodyInput()`	Obtiene el valor del cuerpo de la solicitud
`setFamilyProduct(String value)`	Establece la familia de producto para control de logs
`getFamilyProduct()`	Obtiene la familia de producto actual
`setretryQuantity(Integer value)`	Establece el número de reintentos (método estático)
`getretryQuantity()`	Obtiene el número actual de reintentos (método estático)
`setskipLogDefault(Boolean value)`	Establece si se deben omitir los logs por defecto
`getskipLogDefault()`	Obtiene el valor actual de omisión de logs

Métodos Principales

emit(String methodType, SF_IntegrationSetting__mdt config, String serviceParameters, Object payload, String family)

Descripción

Emite una solicitud HTTP a un servicio externo utilizando la configuración proporcionada.

Parámetros

- `methodType` (String): Tipo de método HTTP (GET, POST, etc.).
- `config` (SF_IntegrationSetting__mdt): Configuración de integración desde metadatos personalizados.
- `serviceParameters` (String): Parámetros adicionales para la URL del servicio.
- `payload` (Object): Carga útil para enviar en la solicitud (para método POST).
- `family` (String): Familia de producto para los logs.

Retorno

- `System.HTTPResponse`: Respuesta HTTP del servicio externo.

Proceso

1. Almacena la carga útil para el registro.
2. Construye la URL del servicio combinando la ruta base con los parámetros.
3. Crea y configura una solicitud HTTP con credenciales nombradas.
4. Establece encabezados y cuerpo para solicitudes POST.
5. Configura el tiempo de espera según la configuración.
6. Crea un wrapper con toda la información necesaria para gestionar el envío.

7. Llama al método `send` para ejecutar la solicitud con lógica de reintento.

Código

```
public System.HTTPResponse emit(
  String methodType,
  SF_IntegrationSetting__mdt config,
  String serviceParameters,
  Object payload,
  String family
) {
  setBodyInput(String.valueOf(payload));
  String service = (config.Path__c ?? '') + serviceParameters;

  Core.debug('Using Named Credentials="' + NAMED_CREDENTIAL + '"');
  HttpRequest req = new HttpRequest();
  req.setEndpoint('callout:' + NAMED_CREDENTIAL + service);
  req.setMethod(methodType);
  if (methodType == 'POST') {
    Core.debug(
      'Emitting ' +
        config.MasterLabel +
        ' to external service="' +
        service +
        '" with payload=' +
        JSON.serialize(payload)
    );
    req.setHeader('Content-Type', 'application/json');
    req.setBody(JSON.serialize(payload));
    req.setTimeout(Integer.valueOf(config.TimeOutLimit__c) ?? 120000);
  }

  SendMethodResourceWrapper sendWrapper = new SendMethodResourceWrapper(
    req,
    (config.RetryOnResponseCode__c ?? '').split(';'),
    config.TimeoutType__c,
    Integer.valueOf(config.RetryQuantity__c)
  );
  String familyProduct = family != null ? family : '';
  System.HttpResponse res = send(
    sendWrapper,
    String.valueOf(config.MasterLabel),
    familyProduct
  );

  return res;
}
```

send(SendMethodResourceWrapper resourceWrapper, String process, String family)

Descripción

Método estático que realiza la solicitud HTTP con lógica de reintento basada en la configuración proporcionada.

Parámetros

- `resourceWrapper` (SendMethodResourceWrapper): Wrapper que contiene la solicitud y la configuración de reintentos.
- `process` (String): Nombre del proceso para incluir en los logs.

- `family` (String): Familia de producto para los logs.

Retorno

- `System.HTTPResponse`: Respuesta HTTP del servicio externo.

Proceso

1. Realiza la solicitud HTTP y mide el tiempo de ejecución.
2. Acumula el tiempo de reintento para control de límites.
3. Verifica si el tiempo de CPU consumido excede el límite máximo.
4. Determina si se debe realizar un reintento basado en el código de respuesta y el número de reintentos restantes.
5. Registra eventos de éxito o error según corresponda.
6. Si es necesario un reintento, aplica un tiempo de espera según la estrategia configurada y llama recursivamente al método.

Código

```
public static System.HttpResponse send(
  SendMethodResourceWrapper resourceWrapper,
  String process,
  String family
) {
  Http http = new Http();
  Long startTime = System.currentTimeMillis();
  Core.debug('Waiting response...');
  System.HttpResponse res = http.send(resourceWrapper.request);

  Long endTime = System.currentTimeMillis();
  Long retryTime = endTime - startTime;
  accumulatedRetryTime += retryTime;

  Integer activeTime = Limits.getCpuTime();
  if (activeTime > MAX_RETRY_TIME) {
    ExternalServiceValidationUtil.publishServiceEventLog(
      'Error',
      'Error trying connect to remote ' +
        resourceWrapper.request.getEndpoint() +
        ' failed with response= tiempo de cpu consumido supera el limite' +
        ' reintento numero = ' +
        getretryQuantity() +
        ' | Tiempo acumulado de CPU= ' +
        activeTime,
      process,
      family,
      getBodyInput(),
      JSON.serialize(res.getBody())
    );
    return res;
  }

  if (
    resourceWrapper.retriesLeft == 0 ||
    !resourceWrapper.retriesCodes.contains(
      String.valueOf(res.getStatusCode())
```

```
    )
  ) {
    if (!skipLogDefault) {
      switch on res.getStatusCode() {
        when 200 {
          ExternalServiceValidationUtil.publishServiceEventLog(
            'Success',
            'Success connecting to remote ' +
              resourceWrapper.request.getEndpoint() +
              ' reintento numero = ' +
              getretryQuantity() +
              ' | Tiempo acumulado = ' +
              accumulatedRetryTime +
              ' milisegundos',
            process,
            family,
            getBodyInput(),
            JSON.serialize(res.getBody())
          );
        }
        when else {
          ExternalServiceValidationUtil.publishServiceEventLog(
            'Error',
            'Error trying connect to remote ' +
              resourceWrapper.request.getEndpoint() +
              ' reintento numero = ' +
              getretryQuantity() +
              ' | Tiempo acumulado = ' +
              accumulatedRetryTime +
              ' milisegundos',
            process,
            family,
            getBodyInput(),
            JSON.serialize(res.getBody())
          );
        }
      }
    }
    return res;
  }

  Core.debug('Retry Code found, retrying...');
  applyTimeOut(
    resourceWrapper.timeOutType,
    resourceWrapper.retriesCurrentNumber
  );
  resourceWrapper.substractRetryLeft();
  resourceWrapper.addRetry();
  return send(resourceWrapper, process, family);
}
```

controlException(String integration, Exception e, String payload)

Descripción

Maneja excepciones relacionadas con llamadas a servicios externos, registrando eventos de plataforma y relanzando excepciones apropiadas.

Parámetros

- `integration` (String): Nombre de la integración donde ocurrió la excepción.
- `e` (Exception): La excepción capturada.
- `payload` (String): Carga útil que se estaba enviando cuando ocurrió la excepción.

Proceso

1. Determina el tipo de excepción (CalloutException, ExternalServiceValidationException, u otra).
2. Publica un evento de plataforma con código 400 o 500 según el tipo de excepción.
3. Lanza una ExternalServiceException o relanza la excepción original según corresponda.

Código

```
public void controlException(
  String integration,
  Exception e,
  String payload
) {
  if (
    e instanceof CalloutException ||
    e instanceof ExternalServiceValidationException
  ) {
    if (
      e.getMessage().contains('400') ||
      e instanceof ExternalServiceValidationException
    )
      this.publishPlatformEvent(
        400,
        JSON.serialize(payload),
        'A  has ocurred during ' +
          integration +
          ' Execution: ' +
          e.getMessage()
      );
    else
      this.publishPlatformEvent(
        500,
        JSON.serialize(payload),
        'A callout Exception has ocurred during ' +
          integration +
          ' Execution: ' +
          e.getMessage()
      );
    throw new ExternalServiceException(
      integration + ' has failed. Exception: ' + e.getMessage()
    );
  } else {
    this.publishPlatformEvent(
      500,
      JSON.serialize(payload),
      'An Exception has ocurred during ' +
        integration +
        ' Execution: ' +
```

```
        e.getMessage()
    );
    throw e;
  }
}
```

controlExceptionSavingLog(String integration, String messaje, String process)

Descripción

Registra información de excepciones en logs de eventos si el registro está habilitado.

Parámetros

- `integration` (String): Nombre de la integración donde ocurrió la excepción.
- `messaje` (String): Mensaje de error.
- `process` (String): Nombre del proceso para incluir en los logs.

Proceso

1. Verifica si el registro de logs está habilitado.
2. Calcula el tiempo transcurrido desde el inicio global.
3. Formatea el mensaje según el tipo de error.
4. Publica un evento de log con la información recopilada.

Código

```
public void controlExceptionSavingLog(
  String integration,
  String messaje,
  String process
) {
  if (!skipLogDefault) {
    System.debug('Generating eventlog');
    Long endTime = System.currentTimeMillis();
    Long retryTime = endTime - startTimeGlobal;

    String setMensaje = messaje.equals('Read timed out')
      ? 'failed with response= tiempo de espera del callout a superado el limite' +
        ' reintento numero = ' +
        getretryQuantity() +
        ' | Tiempo acumulado de cpu= ' +
        Limits.getCpuTime() +
        ' y tiempo acumulado de logica: ' +
        retryTime
      : 'Error trying connect to ' + integration + '.... ' + messaje;
    ExternalServiceValidationUtil.publishServiceEventLog(
      'ERROR',
      setMensaje,
      process,
      familyProductControl,
      getBodyInput(),
      ''
    );
```

```
  }
}
```

Métodos de Gestión de Tiempo de Espera

applyTimeOut(String timeOutType, Integer retriesCurrentNumber)

Descripción

Aplica una estrategia de tiempo de espera según el tipo configurado.

Parámetros

- `timeOutType` (String): Tipo de estrategia de tiempo de espera ('FIXED', 'INCREMENTAL', 'FIBONACCI').
- `retriesCurrentNumber` (Integer): Número actual de reintentos.

Proceso

Aplica la estrategia de tiempo de espera correspondiente según el tipo especificado.

Código

```
private static void applyTimeOut(
  String timeOutType,
  Integer retriesCurrentNumber
) {
  Integer miliseconds = 100;
  switch on timeOutType.toUpperCase() {
    when 'FIXED' {
      fixedTimeOut(miliseconds);
    }
    when 'INCREMENTAL' {
      incrementalTimeOut(retriesCurrentNumber, miliseconds);
    }
    when 'FIBONACCI' {
      waitMethod(miliseconds * fibonacciTimeOut(retriesCurrentNumber));
    }
  }
}
```

Métodos de Estrategia de Tiempo de Espera

- **`fixedTimeOut(Integer miliseconds)`**: Aplica un tiempo de espera fijo.
- **`incrementalTimeOut(Integer numberOfRetry, Integer miliseconds)`**: Aplica un tiempo de espera que incrementa linealmente con el número de reintentos.
- **`fibonacciTimeOut(Integer numberOfRetry)`**: Calcula el término de Fibonacci correspondiente al número de reintento.
- **`waitMethod(Integer miliseconds)`**: Implementa la espera activa durante el número de milisegundos especificado.

Clase Interna

SendMethodResourceWrapper

Descripción

Clase interna privada que encapsula la información necesaria para gestionar solicitudes HTTP con lógica de reintento.

Atributos

- `request` (HttpRequest): La solicitud HTTP a enviar.
- `retriesCodes` (List
  ): Lista de códigos de respuesta que deberían desencadenar un reintento.
- `timeOutType` (String): Tipo de estrategia de tiempo de espera.
- `retriesCurrentNumber` (Integer): Número actual de reintento.
- `retriesLeft` (Integer): Número de reintentos restantes.

Métodos

- **Constructor**: Inicializa los atributos con los valores proporcionados.
- **`substractRetryLeft()`**: Decrementa el contador de reintentos restantes.
- **`addRetry()`**: Incrementa el contador de reintentos actuales y actualiza el contador global.

Dependencias

- `PlatformEventIntegrationObserver`: Clase base que proporciona métodos para publicar eventos de plataforma.
- `SF_IntegrationSetting__mdt`: Tipo de metadatos personalizado que contiene la configuración para la integración.
- `SF_ConfigTimeRetryIntegrations__mdt`: Tipo de metadatos personalizado que contiene la configuración de tiempo máximo de reintento.
- `Core`: Clase de utilidad que proporciona el método `debug` para registro.
- `ExternalServiceValidationUtil`: Clase que proporciona métodos para validar y registrar eventos de servicio.
- `ExternalServiceException`: Excepción personalizada lanzada cuando hay problemas con el servicio externo.
- `ExternalServiceValidationException`: Excepción personalizada para errores de validación.

Estrategias de Reintento

La clase implementa tres estrategias diferentes de tiempo de espera para reintentos:

1. FIXED

Espera un tiempo fijo (100 ms) entre cada reintento.

2. INCREMENTAL

Espera un tiempo que aumenta linealmente con el número de reintentos (n * 100 ms).

3. FIBONACCI

Espera un tiempo basado en la secuencia de Fibonacci, multiplicado por un factor base (100 ms).

Ejemplo de Uso

```
try {
    // Obtener la configuración de integración
    SF_IntegrationSetting__mdt config = SF_IntegrationSetting__mdt.getInstance('mi_servicio');

    // Crear el manejador de notificaciones
    ExternalNotificationHandler handler = new ExternalNotificationHandler();

    // Establecer configuraciones específicas (opcional)
    handler.setFamilyProduct('MiProducto');
    handler.setskipLogDefault(false);
```

```
    // Preparar la carga útil
    Map<String, Object> payload = new Map<String, Object>{
        'id' => '12345',
        'name' => 'Test Request',
        'date' => System.now().format()
    };

    // Emitir la solicitud
    System.HTTPResponse response = handler.emit(
        'POST',
        config,
        '/resource/endpoint',
        payload,
        'MiProducto'
    );

    // Procesar la respuesta
    if (response.getStatusCode() == 200) {
        Map<String, Object> responseBody = (Map<String, Object>)JSON.deserializeUntyped(response.getBody());
        System.debug('Respuesta exitosa: ' + responseBody);
    } else {
        System.debug('Error en la respuesta: ' + response.getStatus());
    }
} catch (Exception e) {
    // Manejar la excepción
    System.debug('Error: ' + e.getMessage());
}
```

Notas Adicionales

1. La clase utiliza un mecanismo sofisticado de reintentos con diferentes estrategias de tiempo de espera.
2. Implementa un control de tiempo máximo para evitar que los reintentos consuman demasiados recursos.
3. Registra detalladamente los eventos de integración para facilitar la depuración y el monitoreo.
4. Está diseñada para ser extendida, permitiendo personalizar comportamientos específicos a través de herencia.
5. Utiliza metadatos personalizados para configurar aspectos como tiempos de espera, estrategias de reintento y códigos de respuesta que deben desencadenar reintentos.
6. Implementa un método de espera activa (`waitMethod`) que podría consumir significativamente recursos de CPU.
7. La clase contiene código comentado que podría representar versiones anteriores o alternativas de implementación.

CCMServiceHandler

Descripción General

`CCMServiceHandler` es una clase virtual global que extiende `PlatformEventIntegrationObserver` y está diseñada para manejar la integración con servicios CCM (Customer Communication Management). Esta clase implementa un patrón de procesamiento en tres fases y proporciona capacidades especializadas para el envío de documentos a través de un sistema de mensajería con enrutamiento específico.

Autor

Felipe Correa \<felipe.correa@nespon.com\>

Proyecto

Foundation Salesforce - Suratech - UH-17589

Última Modificación

20 de febrero de 2025 por "ChangeMeIn@UserSettingsUnder.SFDoc"

Estructura de la Clase

```
global virtual class CCMServiceHandler extends PlatformEventIntegrationObserver {
  // Constantes de configuración
  private static final String SERVICE_METHOD = 'POST';
  private static final String SURATECH_MDT_CONFIG_NAME = 'ccm_endpoint';

  // Métodos de procesamiento
  global virtual Map<String, Object> preProcess(Map<String, Object> input, Map<String, Object> options);
  global Map<String, Object> invokeProcess(Map<String, Object> input, Map<String, Object> output, Map<String,
Object> options);
  global virtual Map<String, Object> postProcess(Map<String, Object> response, Map<String, Object> options);
  global void publishPlatformEventIntegration(String moment, String data);
}
```

Herencia

- **Clase Base**: `PlatformEventIntegrationObserver`
- **Modificadores**: `global virtual`

Constantes

Nombre	Valor	Descripción
`SERVICE_METHOD`	`'POST'`	Método HTTP utilizado para las llamadas al servicio CCM
`SURATECH_MDT_CONFIG_NAME`	`'ccm_endpoint'`	Nombre del registro de metadatos personalizados que contiene la configuración del servicio CCM

Métodos

preProcess(Map<String, Object> input, Map<String, Object> options)

Descripción

Método virtual que se ejecuta antes del procesamiento principal. Realiza la preparación inicial de los datos y opcionalmente publica eventos de plataforma para el monitoreo del proceso CCM.

Parámetros

- `input` (Map<String, Object>): Datos de entrada para el procesamiento CCM.
- `options` (Map<String, Object>): Opciones de configuración para el procesamiento.

Retorno

- `Map<String, Object>`: Los datos de entrada, potencialmente modificados durante el pre-procesamiento.

Proceso

1. Registra el inicio del pre-procesamiento CCM.
2. Verifica si los eventos de plataforma están habilitados en las opciones.
3. Si están habilitados, publica un evento de plataforma con los datos de entrada.
4. Devuelve los datos de entrada sin modificaciones.

Código

```
global virtual Map<String, Object> preProcess(
  Map<String, Object> input,
  Map<String, Object> options
) {
  Core.debug('Starting CCM Integration Service Pre Process');
  if (
    options.containsKey('PlatformEventEnabled') &&
    ((Boolean) options.get('PlatformEventEnabled')) == true
  )
    publishPlatformEventIntegration('pre', JSON.serialize(input));
  return input;
}
```

invokeProcess(Map<String, Object> input, Map<String, Object> output, Map<String, Object> options)

Descripción

Método principal que ejecuta la lógica de integración con el servicio CCM. Maneja la configuración, codificación especial de datos, comunicación con el sistema de mensajería y procesamiento de respuestas.

Parámetros

- `input` (Map<String, Object>): Datos de entrada para el procesamiento CCM.
- `output` (Map<String, Object>): Mapa de salida (no utilizado en la implementación actual).
- `options` (Map<String, Object>): Opciones de configuración para el procesamiento.

Retorno

- `Map<String, Object>`: Respuesta procesada del servicio CCM, o `null` en caso de error.

Proceso

1. Obtiene la configuración desde metadatos personalizados.

2. Configura las opciones de eventos de plataforma.

3. Ejecuta el pre-procesamiento.

4. Codifica el payload en Base64 para transmisión segura.

5. Estructura el mensaje con propiedades específicas para el sistema de mensajería.

6. Realiza la llamada HTTP al servicio CCM.

7. Procesa la respuesta y la estructura en un formato estándar.

8. Publica eventos de plataforma con los resultados.

9. Ejecuta el post-procesamiento.

Características Específicas de CCM

- **Codificación Base64**: Los datos se codifican en Base64 para transmisión segura.
- **Sistema de Mensajería**: Utiliza un routing key específico ('CDD.ccm.document').
- **Metadata de Mensaje**: Incluye propiedades como content-type y payload_encoding.

Código

```
global Map<String, Object> invokeProcess(
  Map<String, Object> input,
  Map<String, Object> output,
  Map<String, Object> options
) {
  // Obtener configuración
  SF_IntegrationSetting__mdt config = Test.isRunningTest()
    ? ExternalServiceMetadataIntegrationMock.getIntegrationSettings(SURATECH_MDT_CONFIG_NAME)
    : SF_IntegrationSetting__mdt.getInstance(SURATECH_MDT_CONFIG_NAME);

  if (config == null) {
    throw new ExternalServiceException(
      'No existing "' + SURATECH_MDT_CONFIG_NAME +
      '" custom metadata type record at Integration Setting'
    );
  }

  options.put('PlatformEventEnabled', config.Platform_Events_Enabled__c);

  ExternalNotificationHandler enh = new ExternalNotificationHandler();
  Map<String, Object> payload = new Map<String, Object>();

  try {
    input = this.preProcess(input, options);
    Core.debug('Starting CCM Integration Service In Process');
    publishPlatformEventIntegration('in', JSON.serialize(input));

    // Codificación especial para CCM
    String payloadString = JSON.serialize(input);
    String encoded = EncodingUtil.base64Encode(Blob.valueOf(JSON.serialize(payloadString)));

    // Estructura del mensaje para CCM
    payload.put('payload_encoding', 'base64');
    payload.put('properties', new Map<String, Object>{ 'content-type' => 'application/json' });
    payload.put('routing_key', 'CDD.ccm.document');
```

```
    payload.put('payload', '{"Content":"' + encoded + '"}');

    // Enviar al servicio
    System.HTTPResponse response = enh.emit(SERVICE_METHOD, config, '', (Object) payload);

    // Procesar respuesta
    Map<String, Object> responseMap = new Map<String, Object>{
      'body' => (Map<String, Object>) JSON.deserializeUntyped(response.getBody()),
      'status' => response.getStatus(),
      'statusCode' => response.getStatusCode()
    };

    this.publishPlatFormEvent(
      response.getStatusCode(),
      JSON.serialize(payload),
      JSON.serialize(response.getBody())
    );

    System.debug('Response Wrapper: ' + responseMap);
    return this.postProcess(responseMap, options);
  } catch (Exception e) {
    enh.controlException('CCM remote service', e, JSON.serialize(payload));
    return null;
  }
}
```

postProcess(Map<String, Object> response, Map<String, Object> options)

Descripción

Método virtual que se ejecuta después del procesamiento principal. Realiza tareas de finalización específicas de CCM y opcionalmente publica eventos de plataforma con la respuesta final.

Parámetros

- `response` (Map<String, Object>): Respuesta del procesamiento principal CCM.
- `options` (Map<String, Object>): Opciones de configuración.

Retorno

- `Map<String, Object>`: La respuesta, potencialmente modificada durante el post-procesamiento.

Código

```
global virtual Map<String, Object> postProcess(
  Map<String, Object> response,
  Map<String, Object> options
) {
  Core.debug('Starting CCM Integration Service Post Process');
  if (
    options.containsKey('PlatformEventEnabled') &&
    ((Boolean) options.get('PlatformEventEnabled')) == true
  )
    publishPlatformEventIntegration('post', JSON.serialize(response));
  return response;
}
```

publishPlatformEventIntegration(String moment, String data)

Descripción

Método que publica eventos de plataforma específicos para CCM en diferentes momentos del procesamiento para proporcionar visibilidad y auditoría del proceso de integración CCM.

Parámetros

- `moment` (String): Momento del procesamiento ('pre', 'in', 'post').
- `data` (String): Datos a incluir en el evento de plataforma.

Retorno

- `void`: Este método no devuelve ningún valor.

Proceso

1. Evalúa el momento especificado usando un switch.
2. Llama al método de publicación de evento apropiado según el momento.
3. Incluye información contextual específica para CCM en cada momento.

Código

```
global void publishPlatformEventIntegration(String moment, String data) {
  switch on moment.toLowerCase() {
    when 'pre' {
      publishInitPrePlatformEvent(data, 'CCM Integration Service');
    }
    when 'post' {
      publishInitPosPlatformEvent(data, 'CCM Integration Service');
    }
    when 'in' {
      publishPlatFormEvent(200, data, 'Starting CCM In process');
    }
  }
}
```

Dependencias

- **`PlatformEventIntegrationObserver`**: Clase base que proporciona funcionalidad de eventos de plataforma.
- **`SF_IntegrationSetting__mdt`**: Tipo de metadatos personalizados que almacena la configuración de integración CCM.
- **`ExternalNotificationHandler`**: Clase que maneja las comunicaciones HTTP con servicios externos.
- **`ExternalServiceException`**: Excepción personalizada para errores de servicios externos.
- **`ExternalServiceMetadataIntegrationMock`**: Clase mock para proporcionar configuración durante las pruebas.
- **`Core`**: Clase utilitaria que proporciona funcionalidad de depuración.

Características Específicas de CCM

Sistema de Mensajería

La clase está diseñada para trabajar con un sistema de mensajería que utiliza:

- **Routing Key**: `'CDD.ccm.document'` para enrutar mensajes CCM
- **Codificación Base64**: Para transmisión segura de documentos
- **Propiedades de Mensaje**: Metadatos específicos del mensaje

Estructura del Payload

```
{
  "payload_encoding": "base64",
  "properties": {
    "content-type": "application/json"
  },
  "routing_key": "CDD.ccm.document",
  "payload": "{\"Content\":\"<base64_encoded_data>\"}"
}
```

Ejemplos de Uso

Ejemplo 1: Envío de Documento CCM Básico

```
public class CCMDocumentService {

    public static Map<String, Object> sendDocument(Map<String, Object> documentData) {
        try {
            // Crear instancia del handler CCM
            CCMServiceHandler handler = new CCMServiceHandler();

            // Preparar opciones
            Map<String, Object> options = new Map<String, Object>{
                'PlatformEventEnabled' => true
            };

            // Preparar datos del documento
            Map<String, Object> input = new Map<String, Object>{
                'documentType' => documentData.get('documentType'),
                'recipientEmail' => documentData.get('recipientEmail'),
                'templateId' => documentData.get('templateId'),
                'documentData' => documentData.get('documentData'),
                'metadata' => new Map<String, Object>{
                    'senderId' => UserInfo.getUserId(),
                    'organizationId' => UserInfo.getOrganizationId(),
                    'timestamp' => System.now()
                }
            };

            // Ejecutar el procesamiento CCM
            Map<String, Object> result = handler.invokeProcess(input, new Map<String, Object>(), options);

            if (result != null && result.containsKey('body')) {
                Map<String, Object> responseBody = (Map<String, Object>) result.get('body');
                return new Map<String, Object>{
                    'success' => true,
                    'messageId' => responseBody.get('messageId'),
                    'status' => result.get('status'),
                    'statusCode' => result.get('statusCode')
                };
            } else {
```

```
                return new Map<String, Object>{
                    'success' => false,
                    'error' => 'No se pudo enviar el documento CCM'
                };
            }
        } catch (Exception e) {
            System.debug('Error enviando documento CCM: ' + e.getMessage());
            return new Map<String, Object>{
                'success' => false,
                'error' => e.getMessage()
            };
        }
    }
}

// Uso del servicio
Map<String, Object> documentData = new Map<String, Object>{
    'documentType' => 'POLICY_CERTIFICATE',
    'recipientEmail' => 'customer@example.com',
    'templateId' => 'TEMPLATE_001',
    'documentData' => new Map<String, Object>{
        'policyNumber' => 'POL-123456',
        'customerName' => 'Juan Pérez',
        'effectiveDate' => Date.today(),
        'premium' => 150000
    }
};

Map<String, Object> result = CCMDocumentService.sendDocument(documentData);

if ((Boolean) result.get('success')) {
    String messageId = (String) result.get('messageId');
    System.debug('Documento CCM enviado exitosamente. Message ID: ' + messageId);
} else {
    System.debug('Error: ' + result.get('error'));
}
```

Ejemplo 2: Extensión Especializada para Comunicaciones de Seguros

```
public class InsuranceCCMHandler extends CCMServiceHandler {

    // Sobrescribir pre-procesamiento para validaciones específicas de seguros
    global override Map<String, Object> preProcess(
        Map<String, Object> input,
        Map<String, Object> options
    ) {
        Core.debug('Starting Insurance CCM Service Pre Process');

        // Validar datos específicos de seguros
        validateInsuranceDocument(input);

        // Enriquecer con datos de la póliza
        input = enrichWithPolicyData(input);

        // Llamar al método padre
        return super.preProcess(input, options);
    }
```

```
    // Sobrescribir post-procesamiento para acciones específicas
    global override Map<String, Object> postProcess(
        Map<String, Object> response,
        Map<String, Object> options
    ) {
        Core.debug('Starting Insurance CCM Service Post Process');

        // Crear registro de comunicación
        createCommunicationRecord(response, options);

        // Actualizar estado de la póliza si es necesario
        updatePolicyStatus(response);

        // Llamar al método padre
        return super.postProcess(response, options);
    }

    private void validateInsuranceDocument(Map<String, Object> input) {
        if (!input.containsKey('policyNumber')) {
            throw new CumuloMissingParameterException('El número de póliza es requerido para documentos de
seguros');
        }

        if (!input.containsKey('documentType')) {
            throw new CumuloMissingParameterException('El tipo de documento es requerido');
        }

        String documentType = (String) input.get('documentType');
        Set<String> validTypes = new Set<String>{'POLICY_CERTIFICATE', 'CLAIM_NOTICE', 'PAYMENT_RECEIPT',
'RENEWAL_NOTICE'};

        if (!validTypes.contains(documentType)) {
            throw new IllegalArgumentException('Tipo de documento no válido para seguros: ' + documentType);
        }
    }

    private Map<String, Object> enrichWithPolicyData(Map<String, Object> input) {
        String policyNumber = (String) input.get('policyNumber');

        // Consultar información adicional de la póliza
        List<Policy__c> policies = [
            SELECT Id, PolicyNumber, Status, Premium, EffectiveDate, ExpirationDate, Account__r.Name,
Account__r.Email
            FROM Policy__c
            WHERE PolicyNumber = :policyNumber
            LIMIT 1
        ];

        if (!policies.isEmpty()) {
            Policy__c policy = policies[0];
            input.put('policyId', policy.Id);
            input.put('policyStatus', policy.Status);
            input.put('customerName', policy.Account__r.Name);
            input.put('customerEmail', policy.Account__r.Email);
            input.put('effectiveDate', policy.EffectiveDate);
            input.put('expirationDate', policy.ExpirationDate);
```

```
        }

        return input;
    }

    private void createCommunicationRecord(Map<String, Object> response, Map<String, Object> options) {
        try {
            // Crear registro de comunicación en objeto personalizado
            Communication_Log__c commLog = new Communication_Log__c(
                Type__c = 'CCM_DOCUMENT',
                Status__c = (Integer) response.get('statusCode') == 200 ? 'Sent' : 'Failed',
                Response_Status_Code__c = (Integer) response.get('statusCode'),
                Sent_Date__c = System.now(),
                Platform_Events_Enabled__c = (Boolean) options.get('PlatformEventEnabled')
            );

            insert commLog;
        } catch (Exception e) {
            System.debug('Error creando registro de comunicación: ' + e.getMessage());
        }
    }

    private void updatePolicyStatus(Map<String, Object> response) {
        // Lógica para actualizar estado de póliza basada en la respuesta CCM
        // Por ejemplo, marcar como "Notificada" después de enviar certificado
    }
}
```

Ejemplo 3: Uso en Procedimiento de Integración

```
public class CCMIntegrationProcedure {

    public static Map<String, Object> sendCCMDocument(Map<String, Object> input) {
        try {
            // Extraer datos del input del procedimiento
            Map<String, Object> documentInfo = (Map<String, Object>) input.get('documentInfo');
            Map<String, Object> options = (Map<String, Object>) input.get('options');

            if (options == null) {
                options = new Map<String, Object>{
                    'PlatformEventEnabled' => true
                };
            }

            // Determinar el tipo de handler a usar
            CCMServiceHandler handler;
            String documentType = (String) documentInfo.get('documentType');

            if (documentType != null && documentType.startsWith('INSURANCE_')) {
                handler = new InsuranceCCMHandler();
            } else {
                handler = new CCMServiceHandler();
            }

            // Procesar con CCM
            Map<String, Object> result = handler.invokeProcess(
                documentInfo,
```

```
                new Map<String, Object>(),
                options
            );

            // Estructurar respuesta para el procedimiento de integración
            if (result != null) {
                return new Map<String, Object>{
                    'success' => true,
                    'data' => result,
                    'message' => 'Documento CCM enviado exitosamente'
                };
            } else {
                return new Map<String, Object>{
                    'success' => false,
                    'data' => null,
                    'message' => 'Error al enviar documento CCM'
                };
            }
        } catch (ExternalServiceException e) {
            return new Map<String, Object>{
                'success' => false,
                'data' => null,
                'message' => 'Error de servicio CCM: ' + e.getMessage(),
                'errorType' => 'CCM_SERVICE_ERROR'
            };
        } catch (Exception e) {
            return new Map<String, Object>{
                'success' => false,
                'data' => null,
                'message' => 'Error inesperado: ' + e.getMessage(),
                'errorType' => 'UNEXPECTED_ERROR'
            };
        }
    }
}
```

Ejemplo 4: Envío Masivo de Documentos CCM

```
public class BulkCCMService {

    @future(callout=true)
    public static void sendBulkDocuments(String serializedDocuments) {
        List<Map<String, Object>> documents =
            (List<Map<String, Object>>) JSON.deserializeUntyped(serializedDocuments);

        CCMServiceHandler handler = new CCMServiceHandler();
        List<Map<String, Object>> results = new List<Map<String, Object>>();

        Map<String, Object> options = new Map<String, Object>{
            'PlatformEventEnabled' => true
        };

        for (Map<String, Object> doc : documents) {
            try {
                Map<String, Object> result = handler.invokeProcess(
                    doc,
                    new Map<String, Object>(),
```

```
                    options
                );

                results.add(new Map<String, Object>{
                    'documentId' => doc.get('documentId'),
                    'success' => result != null,
                    'result' => result
                });

                // Pausa breve entre llamadas para evitar throttling
                if (!Test.isRunningTest()) {
                    System.debug('Documento procesado, continuando...');
                }
            } catch (Exception e) {
                results.add(new Map<String, Object>{
                    'documentId' => doc.get('documentId'),
                    'success' => false,
                    'error' => e.getMessage()
                });
            }
        }

        // Procesar resultados (guardar en base de datos, enviar notificaciones, etc.)
        processBulkResults(results);
    }

    private static void processBulkResults(List<Map<String, Object>> results) {
        Integer successCount = 0;
        Integer failureCount = 0;

        for (Map<String, Object> result : results) {
            if ((Boolean) result.get('success')) {
                successCount++;
            } else {
                failureCount++;
                System.debug('Error en documento ' + result.get('documentId') + ': ' + result.get('error'));
            }
        }

        System.debug('Procesamiento masivo CCM completado. Éxitos: ' + successCount + ', Fallos: ' +
failureCount);

        // Crear registro de resumen
        createBulkProcessingSummary(successCount, failureCount, results.size());
    }

    private static void createBulkProcessingSummary(Integer success, Integer failures, Integer total) {
        try {
            Bulk_Processing_Log__c summary = new Bulk_Processing_Log__c(
                Process_Type__c = 'CCM_BULK_SEND',
                Total_Records__c = total,
                Successful_Records__c = success,
                Failed_Records__c = failures,
                Processing_Date__c = System.now(),
                Success_Rate__c = total > 0 ? (Decimal.valueOf(success) / total * 100).setScale(2) : 0
            );
```

```
            insert summary;
        } catch (Exception e) {
            System.debug('Error creando resumen de procesamiento masivo: ' + e.getMessage());
        }
    }
}
```

Configuración de Metadatos Personalizados

La clase utiliza el registro de metadatos personalizados `SF_IntegrationSetting__mdt` con el nombre `ccm_endpoint`. Este registro debe contener:

Campo	Descripción
`Platform_Events_Enabled__c`	Boolean que indica si los eventos de plataforma están habilitados
Campos de endpoint CCM	URL del servicio CCM, credenciales, timeouts, etc.

Ejemplo de configuración:

```
Label: CCM Endpoint Configuration
Developer Name: ccm_endpoint
Platform_Events_Enabled__c: true
Endpoint__c: https://ccm.suratech.com/api/v1/messages
Timeout__c: 45000
API_Key__c: [encrypted]
```

Consideraciones de Seguridad CCM

1. **Codificación Base64**: Los datos se codifican en Base64 para transmisión segura.
2. **Datos Sensibles**: El contenido del documento puede contener información personal sensible.
3. **Enrutamiento**: El routing key específico asegura que los documentos lleguen al destino correcto.
4. **Configuración Segura**: Las credenciales deben estar protegidas en los metadatos personalizados.

Consideraciones de Rendimiento

1. **Codificación**: La doble serialización y codificación Base64 puede impactar el rendimiento con documentos grandes.
2. **Tamaño de Payload**: Los documentos grandes pueden exceder límites de tamaño de HTTP.
3. **Latencia de Red**: Las comunicaciones con CCM pueden tener latencia variable.
4. **Throttling**: El servicio CCM puede tener límites de velocidad que deben considerarse.

Notas Adicionales

1. **Doble Serialización**: El código realiza una doble serialización del payload, lo que puede ser optimizado.
2. **Routing Key Específico**: El routing key 'CDD.ccm.document' sugiere un sistema de mensajería empresarial específico.
3. **Métodos Virtuales**: Permite extensión para casos de uso específicos como seguros o otros dominios.
4. **Sistema de Colas**: La estructura del payload sugiere integración con un sistema de colas o mensaje broker.

5. **Validación Comentada**: Existe código comentado para validación de esquema CCM que podría implementarse.

MetaDataUtils

Descripción General

`MetaDataUtils` es una clase Apex que proporciona utilidades para trabajar con metadatos personalizados en Salesforce. Esta clase facilita la creación y actualización (upsert) de registros de metadatos personalizados mediante la API de Metadata.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

28 de octubre de 2024

Métodos Públicos

upsertMetadata(List<sObject> customMetadataList)

Descripción

Este método permite crear o actualizar (upsert) múltiples registros de metadatos personalizados en una sola operación.

Parámetros

- `customMetadataList` (List\<sObject\>): Lista de objetos SObject que representan los registros de metadatos personalizados a insertar o actualizar.

Retorno

- `String`: ID del despliegue de metadatos. Este ID puede utilizarse para rastrear el estado del despliegue.

Proceso

1. Crea un contenedor de despliegue (`Metadata.DeployContainer`) para los metadatos personalizados.
2. Para cada objeto en la lista de metadatos:
   - Obtiene el nombre del objeto y sus detalles.
   - Crea una instancia de metadatos personalizados.
   - Establece el nombre completo y la etiqueta del registro.
   - Obtiene todos los campos del objeto.
   - Define un conjunto de campos para omitir ('developername', 'masterlabel', etc.).
   - Itera sobre los campos populados en el objeto.
   - Para cada campo no omitido y con valor no nulo:
     - Crea una instancia de valor de metadatos personalizados.
     - Asigna el nombre del campo y su valor.
     - Añade el campo al objeto de metadatos.
   - Añade el objeto de metadatos al contenedor de despliegue.
3. Crea una instancia de la clase callback (`CustomMetadataCallback`).
4. Encola el despliegue y retorna el ID de despliegue o un ID fijo ('04tO9000000ScNJ') si está ejecutándose en un contexto de prueba.

Código

```
public static String upsertMetadata(List<sObject> customMetadataList) {
  //Create Deployment container for custom Metadata
  Metadata.DeployContainer mdContainer = new Metadata.DeployContainer();
  for (sobject sObMD : customMetadataList) {
    //Get metadata object name and details
    String sObjectname = sObMD.getSObjectType().getDescribe().getName();

    //Create custom Metadata instance
    Metadata.CustomMetadata customMetadata = new Metadata.CustomMetadata();
    String recordName = String.valueOf(sObMD.get('DeveloperName'))
      .replaceAll(' ', '_');
    customMetadata.fullName = sObjectname + '.' + recordName;
    customMetadata.label = (String) sObMD.get('MasterLabel');

    //Get all fields
    schema.SObjectType sobjType = Schema.getGlobalDescribe().get(sObjectname);

    Map<String, Schema.sObjectField> sObjectFields = sobjType.getDescribe()
      .fields.getMap();
    Set<String> skipFieldSet = new Set<String>{
      'developername',
      'masterlabel',
      'language',
      'namespaceprefix',
      'label',
      'qualifiedapiname',
      'id'
    };

    // Use getPopulatedFieldsAsMap to get the populate field and iterate over them
    for (String fieldName : sObMD.getPopulatedFieldsAsMap().keySet()) {
      if (
        skipFieldSet.contains(fieldName.toLowerCase()) ||
        sObMD.get(fieldName) == null
      ) {
        continue;
      }

      Object value = sObMD.get(fieldName);
      //create field instance and populate it with field API name and value
      Metadata.CustomMetadataValue customField = new Metadata.CustomMetadataValue();
      customField.field = fieldName;

      Schema.DisplayType valueType = sObjectFields.get(fieldName)
        .getDescribe()
        .getType();
      customField.value = value;
      //Add fields in the object, similar to creating sObject instance
      customMetadata.values.add(customField);
    }
    //Add metadata in container
    mdContainer.addMetadata(customMetadata);
  }

  // Callback class instance
```

```
  CustomMetadataCallback callback = new CustomMetadataCallback();
  return Test.isRunningTest()
    ? '04tO9000000ScNJ'
    : Metadata.Operations.enqueueDeployment(mdContainer, callback);
}
```

upsertMetadata(sObject customMetadata)

Descripción

Sobrecarga del método `upsertMetadata` que acepta un solo objeto SObject en lugar de una lista. Internamente, convierte el objeto en una lista y llama al método principal.

Parámetros

- `customMetadata` (sObject): Objeto SObject que representa el registro de metadatos personalizado a insertar o actualizar.

Retorno

- `String`: ID del despliegue de metadatos.

Código

```
public static String upsertMetadata(sObject customMetadata) {
  return upsertMetadata(new List<sObject>{ customMetadata });
}
```

Observaciones

1. La clase incluye código comentado relacionado con la conversión de tipos de datos, particularmente para fechas, horas, porcentajes, monedas, etc. Este código está actualmente inactivo pero podría ser útil para manejar conversiones de tipos especiales si es necesario.
2. El método utiliza `getPopulatedFieldsAsMap()` para obtener solo los campos que han sido explícitamente asignados en el objeto, lo que mejora la eficiencia.
3. Se utiliza una clase de callback (`CustomMetadataCallback`) para manejar las respuestas del proceso de despliegue, aunque esta clase no está definida en el código proporcionado.
4. En entornos de prueba (`Test.isRunningTest()`), el método retorna un ID fijo ('04tO9000000ScNJ') en lugar de realizar el despliegue real.

Dependencias

- `CustomMetadataCallback`: Una clase que implementa la interfaz `Metadata.DeployCallback` para manejar las respuestas del proceso de despliegue. Esta clase no está incluida en el código proporcionado.

Ejemplo de Uso

```
// Crear un registro de metadatos personalizado
My_Custom_Metadata__mdt customMD = new My_Custom_Metadata__mdt();
customMD.DeveloperName = 'Test_Record';
customMD.MasterLabel = 'Test Record';
customMD.Field_1__c = 'Value 1';
customMD.Field_2__c = 100;

// Realizar el upsert
```

```
String deploymentId = MetaDataUtils.upsertMetadata(customMD);
System.debug('Deployment ID: ' + deploymentId);
```

LocalEmisionNotificationHandlerMock

Descripción

Clase mock para simular respuestas HTTP durante pruebas del manejador de notificaciones de emisión local. Implementa la interfaz HttpCalloutMock para interceptar llamadas HTTP salientes durante pruebas.

Autor

ChangeMeIn@UserSettingsUnder.SFDoc

Última modificación

31 de enero de 2025 por ChangeMeIn@UserSettingsUnder.SFDoc

Detalles

Esta clase de prueba permite simular diferentes respuestas del servicio de notificaciones de emisión local, incluyendo respuestas exitosas y escenarios de error.

Anotaciones

- `@isTest`: Indica que esta clase es exclusivamente para pruebas.
- `global`: Visibilidad máxima para permitir su uso en cualquier contexto de prueba.

Implementación

Implementa la interfaz `HttpCalloutMock` para simular respuestas HTTP durante pruebas.

Propiedades

Nombre	Tipo	Visibilidad	Descripción
statusCode	Integer	private	Código de estado HTTP a devolver en la respuesta simulada
status	String	private	Mensaje de estado HTTP a devolver
cException	Boolean	private	Bandera para indicar si se debe lanzar una excepción

Constructor

```
global LocalEmisionNotificationHandlerMock(Integer statusCode, String status, boolean cException)
```

Configura la instancia mock con los parámetros especificados para controlar su comportamiento.

Parámetros

- `statusCode`: Código de estado HTTP que devolverá la respuesta simulada
- `status`: Mensaje de estado HTTP que devolverá la respuesta
- `cException`: Si es `true`, el mock lanzará una excepción de callout en lugar de devolver una respuesta

Métodos

respond

```
global HTTPResponse respond(HTTPRequest req)
```

Método principal que implementa la interfaz HttpCalloutMock. Genera y devuelve una respuesta HTTP simulada basada en la configuración del mock.

Parámetros

- `req`: La solicitud HTTP interceptada

Retorno

- Objeto `HTTPResponse` con la respuesta simulada según la configuración

Comportamiento

- Si `cException` es verdadero, lanza una excepción `CalloutException` con el mensaje "Error en el servicio"
- De lo contrario, construye una respuesta HTTP con el código y estado configurados
- El cuerpo de la respuesta depende del endpoint:
  - Para requests a "https://api.ipify.org", devuelve una IP simulada
  - Para otros endpoints, devuelve un objeto JSON vacío

localEmisionBodyResponse (private)

```
private static String localEmisionBodyResponse()
```

Genera el cuerpo de respuesta estándar para el servicio de emisión local.

Retorno

- Un objeto JSON vacío como string: `"{}"`

salesforceExternalIP (private)

```
private static String salesforceExternalIP()
```

Proporciona una dirección IP simulada para las solicitudes al servicio de identificación de IP externa.

Retorno

- Una dirección IP simulada como string: `"155.226.129.250"`

EventLog_Subset_CleanupScheduleClass

Descripción General

`EventLog_Subset_CleanupScheduleClass` es una clase que implementa la interfaz `Schedulable` de Salesforce. Esta clase está diseñada para programar y ejecutar automáticamente la limpieza de registros antiguos del objeto `EventLog_Subset__c`, eliminando entradas que tienen más de un día de antigüedad.

Última Modificación

7 de marzo de 2025

Implementación de Interfaz

- **Interfaz Implementada**: `Schedulable`
- **Propósito**: Permite que la clase sea programada para ejecución utilizando el sistema de trabajos programados de Salesforce.

Método Principal

execute(SchedulableContext sc)

Descripción

Este método es requerido por la interfaz `Schedulable` y se ejecuta automáticamente según la programación establecida. Configura y ejecuta un trabajo por lotes para eliminar registros antiguos de `EventLog_Subset__c`.

Parámetros

- `sc` (SchedulableContext): El contexto de programación proporcionado por el sistema de Salesforce.

Retorno

- `void`

Proceso

1. Obtiene la configuración desde el metadato personalizado `Suratech_Foundation_Schedule_Event__mdt` con el nombre desarrollador 'schedule'.
2. En caso de estar en un contexto de prueba, crea una configuración por defecto.
3. Verifica si la configuración existe, lanzando una excepción si no se encuentra.
4. Comprueba si la programación está activa según la configuración.
5. Si está activa:
   - Calcula la fecha de un día atrás.
   - Formatea la fecha para su uso en una consulta SOQL.
   - Construye una consulta para seleccionar registros de `EventLog_Subset__c` creados antes o en la fecha calculada.
   - Ejecuta un trabajo por lotes (`EventLog_Subset_CleanupBatchClass`) con la consulta construida y un tamaño de lote de 150 registros.

Código

```
public void execute(SchedulableContext sc) {
  Suratech_Foundation_Schedule_Event__mdt config = Suratech_Foundation_Schedule_Event__mdt.getInstance(
    'schedule'
  );

  if (Test.isRunningTest()) {
    config = new Suratech_Foundation_Schedule_Event__mdt(
      MasterLabel = 'DefaultConfig',
      DeveloperName = 'DefaultConfig',
      Frequency__c = '',
      Is_Active__c = true,
      Schedule_Job_Id__c = ''
    );
  }

  if (config == null) {
    throw new MetadataNotFoundException(
      'No Suratech_Foundation_Schedule_Event__mdt Custom Metadata found'
    );
  }

  if (config.Is_Active__c) {
    Datetime todaySubstracted = dateTime.now().addDays(-1);
    String formattedDatetime = todaySubstracted.format(
      'yyyy-MM-dd\'T\'HH:mm:ss\'Z\''
    );
    String query =
      'SELECT Id, CreatedDate FROM EventLog_Subset__c WHERE CreatedDate<=' +
      formattedDatetime;

    Id batchId = Database.executeBatch(
      new EventLog_Subset_CleanupBatchClass(query),
      150
    );
  }
}
```

Excepciones

MetadataNotFoundException

Esta excepción personalizada se lanza cuando no se encuentra la configuración de metadatos personalizada necesaria para la ejecución del trabajo programado.

Dependencias

- `Schedulable`: Interfaz estándar de Salesforce para trabajos programados.
- `Suratech_Foundation_Schedule_Event__mdt`: Tipo de metadatos personalizado que contiene la configuración para el trabajo programado.
- `EventLog_Subset_CleanupBatchClass`: Clase de lotes que realiza la eliminación efectiva de los registros.
- `EventLog_Subset__c`: Objeto personalizado cuyos registros antiguos se eliminarán.
- `MetadataNotFoundException`: Excepción personalizada utilizada cuando no se encuentra la configuración.

Configuración de Metadatos Personalizada

Suratech_Foundation_Schedule_Event__mdt

La clase depende de un registro de metadatos personalizado con el nombre desarrollador 'schedule' que tiene los siguientes campos:

- `MasterLabel`: Etiqueta principal del registro.
- `DeveloperName`: Nombre de desarrollador único.
- `Frequency__c`: Frecuencia de ejecución (no utilizado directamente en el código).
- `Is_Active__c`: Indicador booleano que determina si la programación está activa.
- `Schedule_Job_Id__c`: ID del trabajo programado (no utilizado directamente en el código).

Comportamiento en Contexto de Prueba

En un contexto de prueba (`Test.isRunningTest()`), la clase crea una configuración por defecto con valores predefinidos para evitar depender de la existencia de registros de metadatos personalizados durante las pruebas.

Ejemplo de Uso

Programación Manual

```
// Programar la ejecución diaria a la 1:00 AM
String jobId = System.schedule(
    'EventLog Subset Cleanup - Daily',
    '0 0 1 * * ?',
    new EventLog_Subset_CleanupScheduleClass()
);
```

Programación Dinámica

```
// Obtener la configuración actual
Suratech_Foundation_Schedule_Event__mdt config =
Suratech_Foundation_Schedule_Event__mdt.getInstance('schedule');

// Verificar si existe y está activa
if (config != null && config.Is_Active__c) {
    // Utilizar la frecuencia configurada, o usar un valor predeterminado
    String cronExpression = !String.isBlank(config.Frequency__c) ?
                            config.Frequency__c : '0 0 1 * * ?'; // 1 AM diariamente por defecto

    // Abortar cualquier trabajo existente
    if (!String.isBlank(config.Schedule_Job_Id__c)) {
        try {
            System.abortJob(config.Schedule_Job_Id__c);
        } catch (Exception e) {
            // Manejar el caso donde el trabajo ya no existe
        }
    }

    // Programar el nuevo trabajo
    String jobId = System.schedule(
        'EventLog Subset Cleanup - ' + System.now(),
        cronExpression,
        new EventLog_Subset_CleanupScheduleClass()
```

```
    );

    // Actualizar el ID del trabajo en la configuración
    // (Requiere una implementación personalizada para actualizar metadatos)
    updateScheduleJobId(jobId);
}
```

Clase de Lotes Relacionada (Referencia)

EventLog_Subset_CleanupBatchClass

Esta clase de lotes es la encargada de ejecutar la eliminación real de los registros seleccionados por la consulta.

```
public class EventLog_Subset_CleanupBatchClass implements Database.Batchable<sObject> {
    private String query;

    public EventLog_Subset_CleanupBatchClass(String query) {
        this.query = query;
    }

    public Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator(query);
    }

    public void execute(Database.BatchableContext bc, List<EventLog_Subset__c> scope) {
        if (!scope.isEmpty()) {
            delete scope;
        }
    }

    public void finish(Database.BatchableContext bc) {
        // Acciones posteriores a la eliminación, si son necesarias
    }
}
```

Nota: No incluida en este código, pero referenciada

Impacto y Consideraciones

Rendimiento y Gobernadores

- El tamaño del lote está establecido en 150 registros, lo que permite un equilibrio entre velocidad de procesamiento y riesgo de alcanzar límites de gobernadores.
- La eliminación de registros se realiza en lotes para evitar alcanzar el límite de registros DML en una sola transacción.

Mantenimiento de Datos

- La clase elimina registros que tienen más de un día de antigüedad, lo que ayuda a controlar el tamaño de la base de datos.
- Es importante asegurarse de que los registros de `EventLog_Subset__c` que deben conservarse por más tiempo estén adecuadamente protegidos o archivados antes de que se ejecute esta limpieza.

Programación

- La frecuencia real de ejecución depende de cómo se programe la clase a través del sistema de trabajos programados de Salesforce.

- Se recomienda programarla para ejecutarse durante horas de baja actividad para minimizar el impacto en el rendimiento del sistema.

Notas Adicionales

1. La clase utiliza un enfoque configurable a través de metadatos personalizados, lo que permite activar o desactivar la limpieza sin necesidad de modificar el código.
2. La construcción de la consulta SOQL concatena directamente la fecha formateada, lo que es seguro en este contexto ya que la fecha se genera internamente y no proviene de entrada de usuario.
3. La clase no maneja directamente errores durante la ejecución del lote, confiando en el comportamiento predeterminado de los lotes de Salesforce.
4. La configuración en metadatos personalizados permite una fácil adaptación a diferentes entornos (desarrollo, prueba, producción), ya que cada entorno puede tener su propia configuración.

EventLog_Subset_CleanupBatchClass

Descripción General

`EventLog_Subset_CleanupBatchClass` es una clase que implementa la interfaz `Database.Batchable<sObject>` de Salesforce. Esta clase está diseñada para ejecutar operaciones de eliminación en lotes sobre registros del objeto `EventLog_Subset__c` (inferido por el nombre de la clase), permitiendo procesar grandes volúmenes de datos de manera eficiente y dentro de los límites de gobernabilidad de la plataforma.

Autor

No especificado (aparece como "ChangeMeIn@UserSettingsUnder.SFDoc")

Última Modificación

25 de febrero de 2025

Estructura de la Clase

```
public class EventLog_Subset_CleanupBatchClass implements Database.Batchable<sObject> {
  private String query;

  public EventLog_Subset_CleanupBatchClass(String query) {
    this.query = query;
  }

  public Database.QueryLocator start(Database.BatchableContext bc) {
    return Database.getQueryLocator(this.query);
  }

  public void execute(Database.BatchableContext bc, List<sObject> scope) {
    delete scope;
  }

  public void finish(Database.BatchableContext bc) {
    System.debug('Finish Method');
  }
}
```

Interfaz Implementada

Database.Batchable<sObject>

La clase implementa la interfaz `Database.Batchable<sObject>` de Salesforce, que define tres métodos que deben ser implementados:

- `start`: Método que inicia el proceso por lotes y define el conjunto de registros a procesar.
- `execute`: Método que se ejecuta para cada lote de registros.
- `finish`: Método que se ejecuta después de que todos los lotes han sido procesados.

Propiedades

query

- **Tipo**: `String`

- **Descripción**: Almacena la consulta SOQL que determina qué registros serán procesados por el trabajo por lotes.
- **Visibilidad**: Privada

Constructor

EventLog_Subset_CleanupBatchClass(String query)

Descripción

Constructor que inicializa la clase con la consulta SOQL que determinará qué registros serán procesados.

Parámetros

- `query` (String): Consulta SOQL que especifica los registros a procesar.

Código

```
public EventLog_Subset_CleanupBatchClass(String query) {
  this.query = query;
}
```

Métodos

start(Database.BatchableContext bc)

Descripción

Método que se ejecuta al inicio del proceso por lotes. Define el conjunto de registros que serán procesados utilizando un QueryLocator basado en la consulta SOQL proporcionada.

Parámetros

- `bc` (Database.BatchableContext): Contexto del trabajo por lotes, proporcionado por la plataforma Salesforce.

Retorno

- `Database.QueryLocator`: Un QueryLocator que define el conjunto de registros a procesar.

Código

```
public Database.QueryLocator start(Database.BatchableContext bc) {
  return Database.getQueryLocator(this.query);
}
```

execute(Database.BatchableContext bc, List<sObject> scope)

Descripción

Método que se ejecuta para cada lote de registros definido por el QueryLocator. Elimina todos los registros en el lote actual.

Parámetros

- `bc` (Database.BatchableContext): Contexto del trabajo por lotes, proporcionado por la plataforma Salesforce.
- `scope` (List
  ): Lista de registros a procesar en el lote actual.

Retorno

- `void`: Este método no devuelve ningún valor.

Proceso

1. Elimina todos los registros en el lote actual utilizando la operación DML `delete`.

Código

```
public void execute(Database.BatchableContext bc, List<sObject> scope) {
  delete scope;
}
```

finish(Database.BatchableContext bc)

Descripción

Método que se ejecuta después de que todos los lotes han sido procesados. Actualmente solo registra un mensaje de depuración.

Parámetros

- `bc` (Database.BatchableContext): Contexto del trabajo por lotes, proporcionado por la plataforma Salesforce.

Retorno

- `void`: Este método no devuelve ningún valor.

Proceso

1. Registra un mensaje de depuración indicando que el método de finalización ha sido ejecutado.

Código

```
public void finish(Database.BatchableContext bc) {
  System.debug('Finish Method');
}
```

Relación con EventLog_Subset_CleanupScheduleClass

Esta clase está diseñada para ser utilizada en conjunto con `EventLog_Subset_CleanupScheduleClass`, que es una clase programable que configura y ejecuta este trabajo por lotes. La clase programable:

1. Verifica si la limpieza debe ejecutarse basándose en la configuración de metadatos personalizados.
2. Construye una consulta SOQL para seleccionar registros `EventLog_Subset__c` que sean más antiguos que un día.
3. Inicia este trabajo por lotes pasándole la consulta y un tamaño de lote de 150 registros.

Flujo de Trabajo

1. Una clase programable o algún otro proceso construye una consulta SOQL para seleccionar los registros de `EventLog_Subset__c` que deben ser eliminados.
2. Se crea una instancia de `EventLog_Subset_CleanupBatchClass` pasándole la consulta.
3. Se ejecuta el trabajo por lotes utilizando `Database.executeBatch()`, especificando opcionalmente un tamaño de lote.
4. El método `start` se ejecuta, definiendo el conjunto completo de registros a procesar.
5. El método `execute` se ejecuta para cada lote de registros, eliminándolos.
6. Después de que todos los lotes han sido procesados, se ejecuta el método `finish`, que actualmente solo registra un mensaje de depuración.

Mejores Prácticas Implementadas

1. **Procesamiento por Lotes**: La clase utiliza el framework de procesamiento por lotes de Salesforce para manejar grandes volúmenes de datos de manera eficiente.
2. **Configurabilidad**: La consulta SOQL que determina qué registros se procesan se proporciona externamente, lo que permite flexibilidad en la selección de registros.
3. **Simplicidad**: El código es conciso y se enfoca en una única responsabilidad: eliminar los registros especificados.

Mejoras Posibles

1. **Manejo de Errores**: La implementación actual no incluye manejo de errores durante la eliminación. Podría mejorar incluyendo bloques try-catch para manejar y registrar posibles errores.
2. **Registro de Actividad**: Podría añadir más registro de actividad para facilitar el seguimiento y la depuración.
3. **Notificaciones**: Podría implementar notificaciones para informar sobre el resultado del proceso de limpieza.
4. **Información de Resumen**: El método `finish` podría proporcionar información resumida sobre cuántos registros se procesaron y eliminaron.

Implementación Mejorada Propuesta

```
/**
 * @description       : Clase por lotes para eliminar registros antiguos de EventLog_Subset__c
 * @author            : [Nombre del Autor]
 * @group             : Mantenimiento
 * @last modified on  : 02-25-2025
 * @last modified by  : [Nombre del Modificador]
 **/
public class EventLog_Subset_CleanupBatchClass implements Database.Batchable<sObject>, Database.Stateful {
  private String query;
  private Integer totalRecordsProcessed = 0;
  private Integer totalRecordsDeleted = 0;
  private List<String> errors = new List<String>();

  /**
   * @description Constructor que inicializa la clase con la consulta SOQL para seleccionar los registros a
eliminar
   * @param query Consulta SOQL que define los registros a procesar
   */
  public EventLog_Subset_CleanupBatchClass(String query) {
    this.query = query;
    System.debug(LoggingLevel.INFO, 'Iniciando trabajo de limpieza con consulta: ' + query);
  }

  /**
   * @description Define el conjunto de registros que serán procesados
   * @param bc Contexto del trabajo por lotes
   * @return QueryLocator que define el conjunto de registros a procesar
   */
  public Database.QueryLocator start(Database.BatchableContext bc) {
    try {
      return Database.getQueryLocator(this.query);
    } catch (Exception e) {
```

```
      System.debug(LoggingLevel.ERROR, 'Error en método start: ' + e.getMessage());
      errors.add('Error en método start: ' + e.getMessage());
      // Retornar un QueryLocator vacío en caso de error
      return Database.getQueryLocator('SELECT Id FROM EventLog_Subset__c WHERE Id = null');
    }
  }

  /**
   * @description Procesa cada lote de registros, eliminándolos
   * @param bc Contexto del trabajo por lotes
   * @param scope Lista de registros a procesar en el lote actual
   */
  public void execute(Database.BatchableContext bc, List<sObject> scope) {
    if (scope == null || scope.isEmpty()) {
      return;
    }

    totalRecordsProcessed += scope.size();
    System.debug(LoggingLevel.INFO, 'Procesando lote de ' + scope.size() + ' registros');

    try {
      // Opcionalmente, registrar algunos detalles de los registros antes de eliminarlos
      if (System.isBatch()) {
        for (SObject record : scope) {
          System.debug(LoggingLevel.FINE, 'Eliminando registro: ' + record.Id);
        }
      }

      // Eliminar los registros
      Database.DeleteResult[] results = Database.delete(scope, false); // Allownos some partial success

      // Procesar resultados
      for (Database.DeleteResult result : results) {
        if (result.isSuccess()) {
          totalRecordsDeleted++;
        } else {
          for (Database.Error error : result.getErrors()) {
            String errorMsg = 'Error al eliminar registro: ' + error.getStatusCode() + ' - ' +
error.getMessage();
            System.debug(LoggingLevel.ERROR, errorMsg);
            errors.add(errorMsg);
          }
        }
      }
    } catch (Exception e) {
      String errorMsg = 'Error en ejecución de lote: ' + e.getMessage() + ' ' + e.getStackTraceString();
      System.debug(LoggingLevel.ERROR, errorMsg);
      errors.add(errorMsg);
    }
  }

  /**
   * @description Se ejecuta después de que todos los lotes han sido procesados
   * @param bc Contexto del trabajo por lotes
   */
  public void finish(Database.BatchableContext bc) {
    // Generar mensaje de resumen
```

```
    String summary = 'Proceso de limpieza de EventLog_Subset__c completado. ' +
                     'Registros procesados: ' + totalRecordsProcessed + '. ' +
                     'Registros eliminados: ' + totalRecordsDeleted + '. ' +
                     'Errores: ' + errors.size();

    System.debug(LoggingLevel.INFO, summary);

    // Registrar errores si los hay
    if (!errors.isEmpty()) {
      System.debug(LoggingLevel.ERROR, 'Errores encontrados durante el proceso:');
      for (String error : errors) {
        System.debug(LoggingLevel.ERROR, error);
      }
    }

    // Aquí se podría añadir código para enviar notificaciones, crear registros de auditoría, etc.
    try {
      sendCompletionNotification(summary, errors);
    } catch (Exception e) {
      System.debug(LoggingLevel.ERROR, 'Error al enviar notificación: ' + e.getMessage());
    }
  }

  /**
   * @description Envía una notificación con el resumen de la ejecución del trabajo
   * @param summary Resumen de la ejecución
   * @param errors Lista de errores encontrados
   */
  private void sendCompletionNotification(String summary, List<String> errors) {
    // Implementación del envío de notificación
    // Esto podría incluir envío de email, creación de registro de log, etc.

    // Ejemplo: Crear un registro de log
    /*
    Event_Log__c log = new Event_Log__c(
      Type__c = 'Batch Execution',
      Level__c = errors.isEmpty() ? 'Info' : 'Error',
      Message__c = summary,
      Details__c = String.join(errors, '\n').left(32000) // Límite de caracteres para campo de texto largo
    );

    insert log;
    */
  }
}
```

Ejemplos de Uso

Uso Básico Directamente

```
// Construir la consulta para seleccionar registros de más de un día de antigüedad
Datetime cutoffDate = Datetime.now().addDays(-1);
String formattedCutoffDate = cutoffDate.format('yyyy-MM-dd\'T\'HH:mm:ss\'Z\'');
String query = 'SELECT Id, CreatedDate FROM EventLog_Subset__c WHERE CreatedDate <= ' + formattedCutoffDate;

// Ejecutar el trabajo por lotes con un tamaño de lote de 150
```

```
Id batchId = Database.executeBatch(new EventLog_Subset_CleanupBatchClass(query), 150);

System.debug('Trabajo por lotes iniciado con ID: ' + batchId);
```

Uso desde una Clase Programable

```
public class EventLog_Subset_CleanupScheduleClass implements Schedulable {
  public void execute(SchedulableContext sc) {
    // Verificar configuración (ejemplo simplificado)
    Boolean shouldRun = true;

    if (shouldRun) {
      // Construir consulta para registros de más de un día
      Datetime cutoffDate = Datetime.now().addDays(-1);
      String formattedCutoffDate = cutoffDate.format('yyyy-MM-dd\'T\'HH:mm:ss\'Z\'');
      String query = 'SELECT Id, CreatedDate FROM EventLog_Subset__c WHERE CreatedDate <= ' +
formattedCutoffDate;

      // Ejecutar el trabajo por lotes
      Id batchId = Database.executeBatch(new EventLog_Subset_CleanupBatchClass(query), 150);

      System.debug('Trabajo de limpieza programado iniciado con ID: ' + batchId);
    }
  }
}
```

Programación del Trabajo

```
// Programar el trabajo para ejecutarse todos los días a la 1 AM
String jobId = System.schedule(
    'EventLog Subset Cleanup - Daily',
    '0 0 1 * * ?',
    new EventLog_Subset_CleanupScheduleClass()
);

System.debug('Trabajo programado con ID: ' + jobId);
```

Consideraciones de Rendimiento

1. **Tamaño del Lote**: La clase está diseñada para ser ejecutada con un tamaño de lote específico (150 registros según se infiere de la clase programable relacionada). Este valor puede ajustarse según las necesidades específicas y las características de los registros.
2. **Consulta Eficiente**: La eficiencia de la consulta SOQL proporcionada afectará el rendimiento del trabajo. Es importante asegurarse de que la consulta esté optimizada y utilice índices cuando sea posible.
3. **Operaciones DML**: La clase realiza operaciones de eliminación directamente, lo que consume límites de DML. Asegúrese de que el tamaño de lote sea apropiado para no exceder estos límites.
4. **Datos en Cascada**: Si los registros que se eliminan tienen relaciones y están configurados para eliminación en cascada, esto podría aumentar el consumo de recursos y afectar el rendimiento.

Pruebas Unitarias

A continuación se presenta un ejemplo de cómo podrían ser las pruebas unitarias para esta clase:

```
@IsTest
private class EventLog_Subset_CleanupBatchClassTest {
```

```
    @TestSetup
    static void setupTestData() {
        // Crear registros de prueba
        List<EventLog_Subset__c> testLogs = new List<EventLog_Subset__c>();
        for (Integer i = 0; i < 200; i++) {
            testLogs.add(new EventLog_Subset__c());
        }

        insert testLogs;
    }

    @IsTest
    static void testBatchExecution() {
        // Verificar que existen registros antes de la prueba
        System.assertEquals(200, [SELECT COUNT() FROM EventLog_Subset__c], 'Deberían existir 200 registros
antes de la prueba');

        // Construir consulta para seleccionar todos los registros
        String query = 'SELECT Id FROM EventLog_Subset__c';

        Test.startTest();
        // Ejecutar el trabajo por lotes
        Id batchId = Database.executeBatch(new EventLog_Subset_CleanupBatchClass(query), 50);
        Test.stopTest();

        // Verificar que todos los registros han sido eliminados
        System.assertEquals(0, [SELECT COUNT() FROM EventLog_Subset__c], 'Todos los registros deberían haber
sido eliminados');
    }

    @IsTest
    static void testBatchWithNoRecords() {
        // Eliminar todos los registros creados en setup
        delete [SELECT Id FROM EventLog_Subset__c];

        // Verificar que no hay registros antes de la prueba
        System.assertEquals(0, [SELECT COUNT() FROM EventLog_Subset__c], 'No deberían existir registros antes
de la prueba');

        // Construir consulta para seleccionar todos los registros (que serán ninguno)
        String query = 'SELECT Id FROM EventLog_Subset__c';

        Test.startTest();
        // Ejecutar el trabajo por lotes
        Id batchId = Database.executeBatch(new EventLog_Subset_CleanupBatchClass(query), 50);
        Test.stopTest();

        // Verificar que no hay errores (el trabajo debería completarse sin problemas)
        // No hay aserciones específicas aquí ya que solo estamos verificando que no hay excepciones
    }

    @IsTest
    static void testBatchWithInvalidQuery() {
        // Construir una consulta inválida
        String invalidQuery = 'SELECT InvalidField FROM EventLog_Subset__c';
```

```
        Test.startTest();

        // Capturar la excepción que debería ocurrir
        try {
            Id batchId = Database.executeBatch(new EventLog_Subset_CleanupBatchClass(invalidQuery), 50);
            System.assert(false, 'Debería haber lanzado una excepción');
        } catch (Exception e) {
            // Verificar que se lanzó la excepción esperada
            System.assert(e.getMessage().contains('InvalidField'), 'La excepción debería estar relacionada con
el campo inválido');
        }

        Test.stopTest();
    }
}
```

Notas Adicionales

1. La clase está diseñada con un propósito específico: eliminar registros de `EventLog_Subset__c` según una consulta proporcionada.
2. La implementación actual es muy simple y directa, sin manejo de errores o registro detallado de actividad.
3. El nombre de la clase sugiere que es parte de un sistema de mantenimiento para limpiar registros antiguos del objeto `EventLog_Subset__c`.
4. Esta clase trabaja en conjunto con `EventLog_Subset_CleanupScheduleClass`, que es la clase programable que la configura y ejecuta.
5. La simplicidad de la implementación actual puede ser adecuada si los volúmenes de datos son pequeños y el riesgo de errores es bajo, pero para entornos de producción con grandes volúmenes de datos, se recomendaría una implementación más robusta como la propuesta en la sección "Implementación Mejorada".

EventLogManagerProvider

Descripción General

`EventLogManagerProvider` es una clase que proporciona funcionalidad para generar registros de eventos de prueba o datos de muestra en el objeto Big Object `EventLog__b`. Esta clase está diseñada para facilitar la carga masiva de datos de eventos simulados con valores aleatorios dentro de un conjunto predefinido de opciones.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

30 de octubre de 2024

Estructura de la Clase

```
public with sharing class EventLogManagerProvider {
  public static void generate() {
    // Implementación del método para generar registros de eventos
  }
}
```

Métodos

generate()

Descripción

Método estático que genera y guarda un conjunto de registros de eventos simulados (hasta 5000) en el objeto Big Object `EventLog__b`. Los registros se generan con valores aleatorios para diversos campos, seleccionados de listas predefinidas.

Parámetros

Ninguno.

Retorno

- `void`: Este método no devuelve ningún valor.

Proceso

1. Inicializa un array para almacenar los resultados de las operaciones de guardado y una lista para acumular los registros de eventos.
2. Define listas constantes con valores predefinidos para diferentes campos:
   - `PRODUCTS`: Lista de productos ('Moto', 'Auto', 'Arrendmiento', 'Viaje').
   - `LEVELS`: Lista de niveles de registro ('Info', 'Warning', 'Error', 'Alert').
   - `PLATFORMS`: Lista de plataformas ('Salesforce', 'Mulesoft').
   - `TYPES`: Lista de tipos de evento ('INFO', 'ERROR').
   - `PROCESSES`: Lista de procesos ('Knowing', 'Rating', 'Quoting', 'Issuing', 'Notifying').
3. Itera hasta 5000 veces, generando en cada iteración:
   - Selecciona valores aleatorios de cada lista para crear un registro de evento único.
   - Crea un nuevo registro `EventLog__b` con los valores seleccionados y la fecha actual.

- Añade el registro a la lista de eventos.
- Cuando la lista alcanza 200 registros, los inserta en la base de datos en modo inmediato y limpia la lista.

4. Después del bucle, inserta cualquier registro restante (menos de 200) en la base de datos.

Código

```
public static void generate() {
  Database.SaveResult[] results;
  List<EventLog__b> eventLogs = new List<EventLog__b>();

  final List<String> PRODUCTS = new List<String>{
    'Moto',
    'Auto',
    'Arrendmiento',
    'Viaje'
  };
  final List<String> LEVELS = new List<String>{
    'Info',
    'Warning',
    'Error',
    'Alert'
  };
  final List<String> PLATFORMS = new List<String>{ 'Salesforce', 'Mulesoft' };
  final List<String> TYPES = new List<String>{ 'INFO', 'ERROR' };
  final List<String> PROCESSES = new List<String>{
    'Knowing',
    'Rating',
    'Quoting',
    'Issuing',
    'Notifying'
  };

  for (Integer i = 0; i < 5000; i++) {
    String product = PRODUCTS.get(
      (Math.random() * PRODUCTS.size()).intValue()
    );
    String platform = PLATFORMS.get(
      (Math.random() * PLATFORMS.size()).intValue()
    );
    String level = LEVELS.get((Math.random() * LEVELS.size()).intValue());
    String type = TYPES.get((Math.random() * TYPES.size()).intValue());
    String process = PROCESSES.get(
      (Math.random() * PROCESSES.size()).intValue()
    );

    EventLog__b eventLog = new EventLog__b(
      Date__c = System.now(),
      Product__c = product,
      Level__c = level,
      Platform__c = Platform,
      Process__c = process,
      Type__c = type
    );
    eventLogs.add(eventLog);

    // Insert records in batches
```

```
    if (eventLogs.size() == 200) {
      Database.insertImmediate(eventLogs);
      eventLogs.clear();
    }
  }

  // Insert any remaining records
  if (!eventLogs.isEmpty()) {
    results = Database.insertImmediate(eventLogs);
  }
}
```

Dependencias

- `EventLog__b`: Objeto Big Object que almacena los registros de eventos. Este objeto debe estar definido en la organización con los siguientes campos:
  - `Date__c`: Campo de fecha y hora para el momento del evento.
  - `Product__c`: Campo de texto para el producto relacionado.
  - `Level__c`: Campo de texto para el nivel de importancia del evento.
  - `Platform__c`: Campo de texto para la plataforma donde ocurrió el evento.
  - `Process__c`: Campo de texto para el proceso relacionado.
  - `Type__c`: Campo de texto para el tipo de evento.

Observaciones sobre el Código

Puntos Fuertes

1. **Eficiencia en inserción masiva**: Utiliza un enfoque por lotes (batch) para insertar registros, lo que es más eficiente que insertar registros individualmente.
2. **Versatilidad en datos generados**: Crea una variedad de combinaciones de datos utilizando selección aleatoria de valores predefinidos.
3. **Uso de Big Objects**: Aprovecha la capacidad de Big Objects para almacenar grandes volúmenes de datos de eventos.
4. **Método inmediato de inserción**: Utiliza `Database.insertImmediate()`, que es el método adecuado para insertar registros en Big Objects.

Posibles Errores y Mejoras

1. **Error de variable**: En la asignación `Platform__c = Platform`, se usa `Platform` (mayúscula) como valor, pero la variable es `platform` (minúscula). Esto provocará un error en tiempo de ejecución.
2. **Manejo de errores**: No hay manejo de errores para las operaciones de inserción. Sería recomendable añadir código para verificar los resultados de `Database.insertImmediate()`.
3. **Número fijo de registros**: La generación de 5000 registros está codificada de forma fija. Podría ser beneficioso hacer este número configurable.
4. **Distribución aleatoria simple**: Utiliza una distribución uniforme simple para seleccionar valores, lo que podría no reflejar patrones reales de eventos en un sistema de producción.
5. **No verifica límites del sistema**: No verifica los límites del sistema (como límites de API, DML, etc.) antes de intentar insertar

registros.

Corrección del Error de Variable

El error en `Platform__c = Platform` debería corregirse a `Platform__c = platform` para utilizar la variable correctamente:

```
EventLog__b eventLog = new EventLog__b(
  Date__c = System.now(),
  Product__c = product,
  Level__c = level,
  Platform__c = platform,  // Corregido: platform en minúscula
  Process__c = process,
  Type__c = type
);
```

Mejoras Propuestas

Versión Mejorada con Manejo de Errores y Parámetros Configurables

```
public with sharing class EventLogManagerProvider {
  /**
   * @description Genera registros de eventos simulados en el objeto Big Object EventLog__b
   * @param count Número de registros a generar (predeterminado: 5000, máximo: 10000)
   * @param batchSize Tamaño del lote para inserción (predeterminado: 200, máximo: 200)
   * @return ResultInfo Información sobre el resultado de la operación
   */
  public static ResultInfo generate(Integer count, Integer batchSize) {
    // Validar y establecer valores predeterminados para los parámetros
    count = (count != null && count > 0) ? Math.min(count, 10000) : 5000;
    batchSize = (batchSize != null && batchSize > 0) ? Math.min(batchSize, 200) : 200;

    ResultInfo result = new ResultInfo();
    List<EventLog__b> eventLogs = new List<EventLog__b>();

    // Listas predefinidas de valores
    final List<String> PRODUCTS = new List<String>{
      'Moto', 'Auto', 'Arrendmiento', 'Viaje'
    };
    final List<String> LEVELS = new List<String>{
      'Info', 'Warning', 'Error', 'Alert'
    };
    final List<String> PLATFORMS = new List<String>{
      'Salesforce', 'Mulesoft'
    };
    final List<String> TYPES = new List<String>{
      'INFO', 'ERROR'
    };
    final List<String> PROCESSES = new List<String>{
      'Knowing', 'Rating', 'Quoting', 'Issuing', 'Notifying'
    };

    try {
      for (Integer i = 0; i < count; i++) {
        // Generar valores aleatorios
        String product = getRandomValue(PRODUCTS);
        String platform = getRandomValue(PLATFORMS);
        String level = getRandomValue(LEVELS);
        String type = getRandomValue(TYPES);
```

```
        String process = getRandomValue(PROCESSES);

        // Crear registro de evento
        EventLog__b eventLog = new EventLog__b(
          Date__c = System.now(),
          Product__c = product,
          Level__c = level,
          Platform__c = platform,
          Process__c = process,
          Type__c = type
        );
        eventLogs.add(eventLog);

        // Insertar registros en lotes
        if (eventLogs.size() == batchSize) {
          insertAndProcessResults(eventLogs, result);
          eventLogs.clear();
        }
      }

      // Insertar registros restantes
      if (!eventLogs.isEmpty()) {
        insertAndProcessResults(eventLogs, result);
      }

      result.success = true;
      result.message = 'Generación completada: ' + result.totalRecords + ' registros insertados con ' +
                       result.failedRecords + ' fallos.';
    } catch (Exception e) {
      result.success = false;
      result.message = 'Error en la generación: ' + e.getMessage() + ' en la línea ' + e.getLineNumber();
      result.exception = e;
    }

    return result;
  }

  // Método sobrecargado para usar valores predeterminados
  public static ResultInfo generate() {
    return generate(5000, 200);
  }

  // Método ayudante para seleccionar un valor aleatorio de una lista
  private static String getRandomValue(List<String> values) {
    return values.get((Math.random() * values.size()).intValue());
  }

  // Método auxiliar para insertar registros y procesar resultados
  private static void insertAndProcessResults(List<EventLog__b> eventLogs, ResultInfo result) {
    Database.SaveResult[] saveResults = Database.insertImmediate(eventLogs);

    for (Database.SaveResult sr : saveResults) {
      result.totalRecords++;
      if (!sr.isSuccess()) {
        result.failedRecords++;
        for (Database.Error err : sr.getErrors()) {
          result.errors.add(err.getMessage());
```

```
        }
      }
    }
  }

  // Clase interna para devolver información sobre el resultado
  public class ResultInfo {
    public Boolean success = false;
    public String message = '';
    public Integer totalRecords = 0;
    public Integer failedRecords = 0;
    public List<String> errors = new List<String>();
    public Exception exception;
  }
}
```

Ejemplo de Uso

Uso Básico

```
// Generar 5000 registros usando el método predeterminado
EventLogManagerProvider.generate();
```

Uso Avanzado

```
// Generar 1000 registros con un tamaño de lote de 150
EventLogManagerProvider.ResultInfo result = EventLogManagerProvider.generate(1000, 150);

// Verificar el resultado
if (result.success) {
    System.debug('Generación exitosa: ' + result.message);
} else {
    System.debug('Error en la generación: ' + result.message);
    for (String error : result.errors) {
        System.debug('Error: ' + error);
    }
}
```

Uso en un Trabajo por Lotes

```
public class EventLogGeneratorBatch implements Database.Batchable<Integer>, Database.Stateful {
    private Integer totalRecords;
    private Integer batchSize;
    private List<Integer> iterations;
    private EventLogManagerProvider.ResultInfo finalResult;

    public EventLogGeneratorBatch(Integer totalRecords, Integer batchSize) {
        this.totalRecords = totalRecords;
        this.batchSize = batchSize;
        this.finalResult = new EventLogManagerProvider.ResultInfo();

        // Crear una lista de iteraciones para procesar
        this.iterations = new List<Integer>();
        Integer remainingRecords = totalRecords;
        Integer maxPerBatch = 5000; // Máximo de registros por ejecución de generate()
```

```
        while (remainingRecords > 0) {
            Integer recordsThisBatch = Math.min(remainingRecords, maxPerBatch);
            iterations.add(recordsThisBatch);
            remainingRecords -= recordsThisBatch;
        }
    }

    public List<Integer> start(Database.BatchableContext bc) {
        return iterations;
    }

    public void execute(Database.BatchableContext bc, List<Integer> scope) {
        for (Integer recordCount : scope) {
            EventLogManagerProvider.ResultInfo result = EventLogManagerProvider.generate(recordCount,
batchSize);

            // Acumular resultados
            finalResult.totalRecords += result.totalRecords;
            finalResult.failedRecords += result.failedRecords;
            finalResult.errors.addAll(result.errors);

            if (!result.success) {
                finalResult.success = false;
                finalResult.message += ' ' + result.message;
            }
        }
    }

    public void finish(Database.BatchableContext bc) {
        finalResult.message = 'Generación de registros completada. Total: ' +
                              finalResult.totalRecords + ', Fallidos: ' +
                              finalResult.failedRecords;

        // Registrar resultado final
        System.debug('Resultado final: ' + finalResult.message);
    }
}

// Uso del trabajo por lotes para generar 100,000 registros
Database.executeBatch(new EventLogGeneratorBatch(100000, 200), 1);
```

Consideraciones de Rendimiento

1. **Volumen de Datos**: Generar 5000 registros puede ser intensivo en recursos. Para volúmenes más grandes, considerar un enfoque por lotes o asíncrono.
2. **Límites de Big Objects**: BigObjects tienen límites diferentes a los objetos estándar. Verificar la documentación de Salesforce para conocer los límites actuales.
3. **Límites de Transacción**: El método `Database.insertImmediate()` cuenta para los límites de la transacción actual.
4. **Aleatoriedad**: El método `Math.random()` proporciona una distribución uniforme simple, que puede no ser óptima para todos los casos de prueba.

Uso Típico

Esta clase es útil en varios escenarios:

1. **Configuración de Entornos**: Cargar datos de muestra en entornos de desarrollo o sandbox.
2. **Pruebas de Rendimiento**: Generar volúmenes grandes de datos para probar el rendimiento de componentes que procesan eventos.
3. **Demos y Presentaciones**: Crear datos realistas para demostraciones o presentaciones.
4. **Pruebas de Integración**: Simular eventos para probar integración con sistemas externos.

Notas Adicionales

1. El uso de Big Objects (`EventLog__b`) es una buena elección para almacenar grandes volúmenes de datos de registro, ya que están optimizados para este propósito.
2. La clase utiliza una distribución de valores aleatoria uniforme, lo que significa que cada opción tiene la misma probabilidad de ser seleccionada. En un sistema real, algunos tipos de eventos podrían ser más comunes que otros.
3. La inserción por lotes de 200 registros es una buena práctica para evitar alcanzar los límites de Salesforce para operaciones DML.
4. La clase no incluye la generación de datos relacionados específicos del negocio o contenido detallado de mensajes de error, que podrían hacer los datos más realistas.
5. La fecha utilizada para todos los registros es la misma (`System.now()`). Para simular datos históricos, podría ser útil generar fechas variadas dentro de un rango determinado.

EventLogManagerFoundationAdapter

Descripción General

`EventLogManagerFoundationAdapter` es una clase abstracta global que proporciona una estructura base para la implementación de adaptadores de gestión de registros de eventos. Esta clase define una interfaz común para aplicar la lógica de registro de eventos, permitiendo diferentes implementaciones específicas a través de herencia.

Estructura

```
global abstract class EventLogManagerFoundationAdapter {

    global abstract Boolean applyEventLog(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap,
        Map<String, Object> optionMap
    );

    public Boolean invokeMethod(
        String methodName,
        Map<String, Object> inputMap,
        Map<String, Object> outputMap,
        Map<String, Object> optionMap
    ) {
        switch on methodName.toUpperCase() {
            when 'APPLYEVENTLOG' {
                return applyEventLog(inputMap, outputMap, optionMap);
            }
            when else {
                Core.debug('Unsupported method: ' + methodName);
                return false;
            }
        }
    }
}
```

Métodos

applyEventLog(Map<String, Object> inputMap, Map<String, Object> outputMap, Map<String, Object> optionMap)

Descripción

Método abstracto que debe ser implementado por las clases derivadas. Este método aplica la lógica de registro de eventos.

Parámetros

- `inputMap` (Map<String, Object>): Mapa con los datos de entrada para el registro de eventos.
- `outputMap` (Map<String, Object>): Mapa donde se pueden guardar resultados o datos de salida.
- `optionMap` (Map<String, Object>): Mapa con opciones adicionales para configurar el comportamiento del registro.

Retorno

- `Boolean`: Indica si el registro de eventos se aplicó con éxito (`true`) o no (`false`).

Implementación

Por ser un método abstracto, no tiene implementación en esta clase. Las clases que extiendan `EventLogManagerFoundationAdapter` deben proporcionar una implementación concreta.

invokeMethod(String methodName, Map<String, Object> inputMap, Map<String, Object> outputMap, Map<String, Object> optionMap)

Descripción

Método público que sirve como punto de entrada para invocar métodos específicos del adaptador. Actualmente, soporta el método 'APPLYEVENTLOG'.

Parámetros

- `methodName` (String): Nombre del método a invocar.
- `inputMap` (Map<String, Object>): Mapa con los datos de entrada para el método.
- `outputMap` (Map<String, Object>): Mapa donde se pueden guardar resultados o datos de salida.
- `optionMap` (Map<String, Object>): Mapa con opciones adicionales para configurar el comportamiento del método.

Retorno

- `Boolean`: Resultado de la operación, que depende del método invocado:
  - Para 'APPLYEVENTLOG': Devuelve el resultado del método `applyEventLog`.
  - Para cualquier otro método: Devuelve `false` e imprime un mensaje de depuración indicando que el método no está soportado.

Proceso

1. Convierte el nombre del método a mayúsculas para una comparación insensible a mayúsculas/minúsculas.
2. Utiliza una declaración `switch` para determinar qué método invocar:
   - Si el método es 'APPLYEVENTLOG', invoca el método abstracto `applyEventLog`.
   - Para cualquier otro método, registra un mensaje de depuración y devuelve `false`.

Código

```
public Boolean invokeMethod(
    String methodName,
    Map<String, Object> inputMap,
    Map<String, Object> outputMap,
    Map<String, Object> optionMap
) {
    switch on methodName.toUpperCase() {
        when 'APPLYEVENTLOG' {
            return applyEventLog(inputMap, outputMap, optionMap);
        }
        when else {
            Core.debug('Unsupported method: ' + methodName);
            return false;
        }
    }
}
```

Patrones de Diseño Utilizados

Adaptador (Adapter)

La clase implementa el patrón de diseño Adaptador, proporcionando una interfaz común (a través de `applyEventLog`) que puede ser implementada por diferentes adaptadores concretos para trabajar con distintos sistemas o mecanismos de registro de eventos.

Plantilla (Template Method)

El método `invokeMethod` sigue el patrón Plantilla, definiendo la estructura de la operación, pero delegando pasos específicos (como `applyEventLog`) a las subclases.

Comando (Command)

El método `invokeMethod` implementa un patrón similar al Comando, donde la acción a ejecutar se determina en función del nombre del método pasado como parámetro.

Dependencias

- `Core`: Clase utilitaria que proporciona el método `debug` para registrar mensajes de depuración.

Extensibilidad

La clase está diseñada para ser extendida. Las clases derivadas deben implementar el método abstracto `applyEventLog` para proporcionar la lógica específica de registro de eventos.

Ejemplos de Implementación

Implementación Básica

```
public class SimpleEventLogAdapter extends EventLogManagerFoundationAdapter {

    global override Boolean applyEventLog(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap,
        Map<String, Object> optionMap
    ) {
        try {
            // Extraer datos necesarios del inputMap
            String eventType = (String)inputMap.get('eventType');
            String message = (String)inputMap.get('message');
            String userId = UserInfo.getUserId();
            Datetime timestamp = Datetime.now();

            // Crear un registro de evento
            Event_Log__c eventLog = new Event_Log__c(
                Event_Type__c = eventType,
                Message__c = message,
                User__c = userId,
                Timestamp__c = timestamp
            );

            // Opcionalmente, añadir detalles adicionales si están presentes
            if (inputMap.containsKey('details')) {
                eventLog.Details__c = JSON.serialize(inputMap.get('details'));
            }

            // Insertar el registro
            insert eventLog;

            // Opcionalmente, guardar información en el outputMap
            outputMap.put('success', true);
```

```
            outputMap.put('eventLogId', eventLog.Id);

            return true;
        } catch (Exception e) {
            // Registrar el error
            System.debug('Error al aplicar registro de evento: ' + e.getMessage());

            // Opcionalmente, guardar información de error en el outputMap
            outputMap.put('success', false);
            outputMap.put('errorMessage', e.getMessage());

            return false;
        }
    }
}
```

Implementación con Plataforma de Eventos

```
public class PlatformEventLogAdapter extends EventLogManagerFoundationAdapter {

    global override Boolean applyEventLog(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap,
        Map<String, Object> optionMap
    ) {
        try {
            // Extraer datos necesarios del inputMap
            String eventName = (String)inputMap.get('eventName');

            // Determinar dinámicamente qué tipo de evento de plataforma publicar
            switch on eventName {
                when 'SystemAudit' {
                    return publishSystemAuditEvent(inputMap, outputMap);
                }
                when 'SecurityAlert' {
                    return publishSecurityAlertEvent(inputMap, outputMap);
                }
                when 'IntegrationLog' {
                    return publishIntegrationLogEvent(inputMap, outputMap);
                }
                when else {
                    outputMap.put('success', false);
                    outputMap.put('errorMessage', 'Tipo de evento no soportado: ' + eventName);
                    return false;
                }
            }
        } catch (Exception e) {
            // Registrar el error
            System.debug('Error al publicar evento de plataforma: ' + e.getMessage());

            // Guardar información de error en el outputMap
            outputMap.put('success', false);
            outputMap.put('errorMessage', e.getMessage());

            return false;
        }
    }
```

```
    private Boolean publishSystemAuditEvent(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap
    ) {
        // Crear el evento de plataforma
        System_Audit_Event__e auditEvent = new System_Audit_Event__e(
            User_Id__c = UserInfo.getUserId(),
            Action__c = (String)inputMap.get('action'),
            Record_Id__c = (String)inputMap.get('recordId'),
            Details__c = (String)inputMap.get('details')
        );

        // Publicar el evento
        Database.SaveResult result = EventBus.publish(auditEvent);

        // Verificar el resultado
        if (result.isSuccess()) {
            outputMap.put('success', true);
            outputMap.put('eventId', result.getId());
            return true;
        } else {
            outputMap.put('success', false);
            outputMap.put('errors', result.getErrors());
            return false;
        }
    }

    private Boolean publishSecurityAlertEvent(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap
    ) {
        // Implementación similar para eventos de alerta de seguridad
        // ...
        return true;
    }

    private Boolean publishIntegrationLogEvent(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap
    ) {
        // Implementación similar para eventos de log de integración
        // ...
        return true;
    }
}
```

Implementación con Múltiples Destinos

```
public class MultiTargetEventLogAdapter extends EventLogManagerFoundationAdapter {

    global override Boolean applyEventLog(
        Map<String, Object> inputMap,
        Map<String, Object> outputMap,
        Map<String, Object> optionMap
    ) {
        // Determinar los destinos de registro basados en opciones
```

```
        Set<String> targetSystems = new Set<String>();

        if (optionMap.containsKey('targets')) {
            targetSystems = (Set<String>)optionMap.get('targets');
        } else {
            // Destinos predeterminados si no se especifican
            targetSystems.add('DATABASE');
            targetSystems.add('PLATFORM_EVENTS');
        }

        Boolean overallSuccess = true;
        Map<String, Object> results = new Map<String, Object>();

        // Aplicar el registro a cada destino
        for (String target : targetSystems) {
            Boolean targetSuccess = false;

            switch on target.toUpperCase() {
                when 'DATABASE' {
                    targetSuccess = logToDatabase(inputMap);
                }
                when 'PLATFORM_EVENTS' {
                    targetSuccess = logToPlatformEvents(inputMap);
                }
                when 'EXTERNAL_SYSTEM' {
                    targetSuccess = logToExternalSystem(inputMap);
                }
                when else {
                    System.debug('Destino de registro no soportado: ' + target);
                    targetSuccess = false;
                }
            }

            results.put(target, targetSuccess);
            overallSuccess = overallSuccess && targetSuccess;
        }

        // Guardar resultados detallados
        outputMap.put('targetResults', results);
        outputMap.put('overallSuccess', overallSuccess);

        return overallSuccess;
    }

    private Boolean logToDatabase(Map<String, Object> inputMap) {
        // Implementación para registrar en la base de datos
        // ...
        return true;
    }

    private Boolean logToPlatformEvents(Map<String, Object> inputMap) {
        // Implementación para publicar eventos de plataforma
        // ...
        return true;
    }

    private Boolean logToExternalSystem(Map<String, Object> inputMap) {
```

```
        // Implementación para enviar a un sistema externo
        // ...
        return true;
    }
}
```

Ejemplo de Uso

```
// Crear una instancia del adaptador concreto
EventLogManagerFoundationAdapter logAdapter = new SimpleEventLogAdapter();

// Preparar los mapas de entrada, salida y opciones
Map<String, Object> inputMap = new Map<String, Object>{
    'eventType' => 'User Activity',
    'message' => 'Usuario actualizó registro',
    'details' => new Map<String, Object>{
        'recordId' => '001xx000003G9abCDE',
        'action' => 'UPDATE',
        'fields' => new List<String>{'Name', 'Phone', 'Email'}
    }
};

Map<String, Object> outputMap = new Map<String, Object>();
Map<String, Object> optionMap = new Map<String, Object>{
    'storeDetails' => true,
    'notifyAdmin' => false
};

// Invocar el método para aplicar el registro de eventos
Boolean success = logAdapter.invokeMethod('applyEventLog', inputMap, outputMap, optionMap);

// Verificar el resultado
if (success) {
    System.debug('Registro de evento aplicado con éxito. ID: ' + outputMap.get('eventLogId'));
} else {
    System.debug('Error al aplicar registro de evento: ' + outputMap.get('errorMessage'));
}
```

Ejemplo de Uso en Sistema Más Amplio

```
public class TransactionManager {

    private EventLogManagerFoundationAdapter eventLogAdapter;

    public TransactionManager(EventLogManagerFoundationAdapter logAdapter) {
        this.eventLogAdapter = logAdapter;
    }

    public Boolean processTransaction(Map<String, Object> transactionData) {
        try {
            // 1. Validar datos de transacción
            validateTransaction(transactionData);

            // 2. Procesar la transacción
            String transactionId = executeTransaction(transactionData);
```

```
        // 3. Registrar evento de éxito
        Map<String, Object> logInput = new Map<String, Object>{
            'eventType' => 'Transaction',
            'message' => 'Transacción completada con éxito',
            'details' => new Map<String, Object>{
                'transactionId' => transactionId,
                'amount' => transactionData.get('amount'),
                'accountId' => transactionData.get('accountId'),
                'timestamp' => Datetime.now()
            }
        };

        Map<String, Object> logOutput = new Map<String, Object>();
        Map<String, Object> logOptions = new Map<String, Object>{
            'priority' => 'High',
            'retainDays' => 90
        };

        eventLogAdapter.invokeMethod('applyEventLog', logInput, logOutput, logOptions);

        return true;
    } catch (Exception e) {
        // Registrar evento de error
        Map<String, Object> errorLogInput = new Map<String, Object>{
            'eventType' => 'Error',
            'message' => 'Error en transacción: ' + e.getMessage(),
            'details' => new Map<String, Object>{
                'errorType' => e.getTypeName(),
                'stackTrace' => e.getStackTraceString(),
                'transactionData' => transactionData
            }
        };

        Map<String, Object> errorLogOutput = new Map<String, Object>();
        Map<String, Object> errorLogOptions = new Map<String, Object>{
            'priority' => 'Critical',
            'notifyAdmin' => true
        };

        eventLogAdapter.invokeMethod('applyEventLog', errorLogInput, errorLogOutput, errorLogOptions);

        return false;
    }
}

private void validateTransaction(Map<String, Object> transactionData) {
    // Lógica de validación
    // ...
}

private String executeTransaction(Map<String, Object> transactionData) {
    // Lógica de ejecución de transacción
    // ...
    return 'TX-' + String.valueOf(Math.random()).substring(2, 10);
```

```
    }
}
```

Consideraciones de Diseño

Ventajas

1. **Flexibilidad**: Permite diferentes implementaciones de registro de eventos sin cambiar la interfaz.
2. **Extensibilidad**: Facilita la adición de nuevos tipos de adaptadores de registro.
3. **Separación de Responsabilidades**: Separa la lógica de invocación de métodos de la implementación específica de registro.
4. **Reutilización**: La estructura común puede ser reutilizada por múltiples adaptadores.

Desafíos

1. **Complejidad**: Introduce una capa adicional de abstracción.
2. **Mantenimiento**: Requiere mantener la coherencia entre diferentes implementaciones.
3. **Rendimiento**: Puede tener un impacto en el rendimiento debido a la abstracción adicional.

Mejores Prácticas

1. **Documentación Clara**: Documentar claramente la estructura esperada de los mapas de entrada, salida y opciones para cada implementación.
2. **Manejo de Errores Robusto**: Implementar un manejo de errores coherente en todas las implementaciones.
3. **Pruebas Unitarias**: Crear pruebas unitarias exhaustivas para cada implementación concreta.
4. **Uso Consistente**: Mantener una estructura consistente para los datos de entrada y salida entre diferentes implementaciones.
5. **Extender según Necesidades**: Considerar extender la funcionalidad añadiendo más métodos soportados en `invokeMethod` si es necesario.

Notas Adicionales

1. La clase utiliza mapas genéricos (`Map<String, Object>`) para los datos de entrada, salida y opciones, proporcionando flexibilidad pero requiriendo un cuidado adicional para garantizar la consistencia de los datos.
2. El uso de `Core.debug` para los mensajes de depuración sugiere que hay una clase utilitaria `Core` disponible en el sistema.
3. La naturaleza abstracta de la clase asegura que solo se pueden crear instancias de subclases concretas.
4. El modificador `global` indica que la clase está diseñada para ser accesible desde diferentes paquetes o namespaces.
5. Actualmente, el método `invokeMethod` solo soporta 'APPLYEVENTLOG', pero el diseño permite añadir fácilmente soporte para más métodos en el futuro.

EventLogManager

Descripción General

`EventLogManager` es una clase global con sharing que proporciona funcionalidad completa para la gestión de registros de eventos en el sistema. Esta clase maneja tanto Big Objects (`EventLog__b`) como objetos personalizados estándar (`EventLog_Subset__c`), ofreciendo capacidades de consulta, guardado, análisis y limpieza de logs de eventos.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

17 de marzo de 2025 por Esneyder Zabala

Estructura de la Clase

```
global with sharing class EventLogManager {
  public static List<EventLog__b> fetchAll(Datetime startAt, Datetime endAt);
  public static Boolean save(List<Object> inputMapList);
  @future(callout=true) global static void saveAsync(String serializeInputMapList);
  public static Boolean refreshAnalytics(Datetime startAt, Datetime endAt, Boolean reset);
  @future public static void refreshAnalyticsAsync(Datetime startAt, Datetime endAt, Boolean reset);
  public static Boolean clearAnalytics();
  @future public static void clearLogsAsync();
}
```

Métodos

fetchAll(Datetime startAt, Datetime endAt)

Descripción

Recupera registros de eventos del Big Object `EventLog__b` dentro de un rango de fechas específico. Incluye soporte para testing a través de mocks.

Parámetros

- `startAt` (Datetime): Fecha y hora de inicio del rango de consulta.
- `endAt` (Datetime): Fecha y hora de fin del rango de consulta.

Retorno

- `List<EventLog__b>`: Lista de registros de eventos encontrados en el rango especificado.

Proceso

1. Registra información de depuración sobre el rango de consulta.
2. Ejecuta una consulta SOQL al Big Object `EventLog__b` con límite de 50,000 registros.
3. Registra el número de registros encontrados.
4. Si se ejecuta en contexto de prueba, devuelve datos mock.
5. Retorna los registros encontrados.

Código

```
public static List<EventLog__b> fetchAll(Datetime startAt, Datetime endAt) {
  Core.debug('Querying Event Logs from ' + startAt + ' to ' + endAt);
  List<EventLog__b> records = [
    SELECT
      Id,
      Date__c,
      Product__c,
      Level__c,
      Platform__c,
      Process__c,
      Details__c,
      Data__c,
      Type__c,
      UserId__c
    FROM EventLog__b
    WHERE Date__c >= :startAt AND Date__c <= :endAt
    LIMIT 50000
  ];
  Core.debug('Found ' + records.size() + ' records on the specified range...');

  if (Test.isRunningTest()) {
    return EventLogManagerMock.getMockLogs();
  }

  return records;
}
```

save(List<Object> inputMapList)

Descripción

Guarda una lista de registros de eventos tanto en el Big Object `EventLog__b` como en el objeto personalizado `EventLog_Subset__c`. Incluye validación de campos obligatorios y procesamiento por lotes.

Parámetros

- `inputMapList` (List

  ): Lista de mapas que contienen los datos de eventos a guardar.

Retorno

- `Boolean`: `true` si el guardado fue exitoso, `false` en caso de error.

Proceso

1. Verifica si la feature flag 'sura_foundation_event_logs' está habilitada.
2. Itera sobre la lista de mapas de entrada.
3. Valida que cada mapa contenga los campos obligatorios (Level, Type, Product).
4. Crea registros para ambos objetos: `EventLog__b` y `EventLog_Subset__c`.
5. Procesa en lotes de 200 registros para optimizar el rendimiento.
6. Ejecuta las operaciones de inserción correspondientes.
7. Maneja excepciones y retorna el resultado.

Campos Procesados

- **Date__c**: Fecha y hora actual
- **Product__c**: Producto (obligatorio)
- **Level__c**: Nivel del evento (obligatorio)
- **Platform__c**: Plataforma de origen
- **Process__c**: Proceso relacionado
- **Details__c**: Detalles del evento
- **Data__c**: Datos adicionales
- **Type__c**: Tipo de evento (obligatorio)
- **UserId__c**: ID del usuario
- **Response__c**: Respuesta del evento

saveAsync(String serializeInputMapList)

Descripción

Versión asíncrona del método `save` que utiliza el decorador `@future(callout=true)` para procesar el guardado de eventos de forma asíncrona.

Parámetros

- `serializeInputMapList` (String): Lista de mapas serializada en JSON que contiene los datos de eventos a guardar.

Retorno

- `void`: Al ser un método future, no devuelve valor.

Proceso

1. Deserializa la cadena JSON a una lista de objetos.
2. Llama al método `save` con la lista deserializada.

Código

```
@future(callout=true)
global static void saveAsync(String serializeInputMapList) {
  List<Object> inputMapList = (List<Object>) JSON.deserializeUntyped(
    serializeInputMapList
  );
  EventLogManager.save(inputMapList);
}
```

refreshAnalytics(Datetime startAt, Datetime endAt, Boolean reset)

Descripción

Actualiza los datos analíticos poblando el objeto `EventLog_Subset__c` con datos del Big Object `EventLog__b` dentro de un rango de fechas específico.

Parámetros

- `startAt` (Datetime): Fecha de inicio del rango para el análisis.
- `endAt` (Datetime): Fecha de fin del rango para el análisis.

- `reset` (Boolean): Indica si se deben limpiar los datos analíticos existentes antes de la actualización.

Retorno

- `Boolean`: `true` si la actualización fue exitosa.

Proceso

1. Si `reset` es true, limpia los datos analíticos existentes.
2. Consulta eventos del Big Object usando `fetchAll`.
3. Crea registros correspondientes en `EventLog_Subset__c`.
4. Inserta los registros del subset si no se está ejecutando en pruebas.

refreshAnalyticsAsync(Datetime startAt, Datetime endAt, Boolean reset)

Descripción

Versión asíncrona del método `refreshAnalytics` que utiliza el decorador `@future`.

Parámetros

- `startAt` (Datetime): Fecha de inicio del rango.
- `endAt` (Datetime): Fecha de fin del rango.
- `reset` (Boolean): Indica si se deben limpiar los datos existentes.

Retorno

- `void`: Al ser un método future, no devuelve valor.

clearAnalytics()

Descripción

Elimina todos los registros del objeto `EventLog_Subset__c` para limpiar los datos analíticos.

Parámetros

Ninguno.

Retorno

- `Boolean`: `true` si la limpieza fue exitosa, `false` en caso de error.

Proceso

1. Consulta hasta 50,000 registros de `EventLog_Subset__c`.
2. Si existen registros, los elimina.
3. Maneja excepciones y retorna el resultado.

clearLogsAsync()

Descripción

Método asíncrono que elimina registros del Big Object `EventLog__b` para limpiar logs antiguos.

Parámetros

Ninguno.

Retorno

- `void`: Al ser un método future, no devuelve valor.

Proceso

1. Consulta hasta 5,000 registros de `EventLog__b`.
2. Si existen registros, los elimina usando `Database.deleteImmediate`.
3. Maneja excepciones internamente.

Dependencias

- **`EventLog__b`**: Big Object que almacena los registros de eventos principales.
- **`EventLog_Subset__c`**: Objeto personalizado que almacena un subconjunto de eventos para análisis.
- **`FeatureFlags`**: Clase que maneja feature flags para controlar la funcionalidad.
- **`Core`**: Clase utilitaria para registro de depuración.
- **`EventLogManagerMock`**: Clase mock para proporcionar datos de prueba.

Arquitectura de Datos

EventLog__b (Big Object)

Big Object optimizado para grandes volúmenes de datos de eventos:

Campo	Tipo	Descripción
Id	Text	Identificador único
Date__c	DateTime	Fecha y hora del evento
Product__c	Text	Producto relacionado
Level__c	Text	Nivel del evento (Info, Warning, Error, etc.)
Platform__c	Text	Plataforma de origen
Process__c	Text	Proceso que generó el evento
Details__c	LongTextArea	Detalles del evento
Data__c	LongTextArea	Datos adicionales
Type__c	Text	Tipo de evento
UserId__c	Text	ID del usuario
Response__c	LongTextArea	Respuesta del evento

EventLog_Subset__c (Objeto Personalizado)

Objeto estándar para análisis y reportes:

Campo	Tipo	Descripción
Name	Text	Nombre del registro
Date__c	DateTime	Fecha y hora del evento
Product__c	Text	Producto relacionado
Level__c	Text	Nivel del evento
Platform__c	Text	Plataforma de origen
Process__c	Text	Proceso que generó el evento
Details__c	LongTextArea	Detalles del evento
Type__c	Text	Tipo de evento
Response__c	LongTextArea	Respuesta del evento
Data__c	LongTextArea	Datos adicionales

Ejemplos de Uso

Ejemplo 1: Guardar Eventos de Forma Síncrona

```
public class EventLogger {

    public static void logUserActivity(String userId, String activity, String details) {
        List<Object> events = new List<Object>();

        Map<String, Object> eventData = new Map<String, Object>{
            'Level' => 'Info',
            'Type' => 'USER_ACTIVITY',
            'Product' => 'CRM',
            'Platform' => 'Salesforce',
            'Process' => 'User Management',
            'Details' => details,
            'UserId' => userId,
            'Data' => JSON.serialize(new Map<String, Object>{
                'activity' => activity,
                'timestamp' => System.now(),
                'sessionId' => UserInfo.getSessionId()
            })
        };

        events.add(eventData);

        Boolean success = EventLogManager.save(events);

        if (success) {
            System.debug('Evento de actividad de usuario registrado exitosamente');
        } else {
```

```
            System.debug('Error al registrar evento de actividad de usuario');
        }
    }
}

// Uso
EventLogger.logUserActivity(UserInfo.getUserId(), 'LOGIN', 'Usuario inició sesión exitosamente');
```

Ejemplo 2: Guardar Múltiples Eventos Asincrónicamente

```
public class BatchEventLogger {

    public static void logIntegrationEvents(List<Map<String, Object>> integrationResults) {
        List<Object> events = new List<Object>();

        for (Map<String, Object> result : integrationResults) {
            String level = (Boolean) result.get('success') ? 'Info' : 'Error';

            Map<String, Object> eventData = new Map<String, Object>{
                'Level' => level,
                'Type' => 'INTEGRATION',
                'Product' => (String) result.get('product'),
                'Platform' => 'External API',
                'Process' => 'Data Integration',
                'Details' => (String) result.get('message'),
                'UserId' => UserInfo.getUserId(),
                'Response' => JSON.serialize(result.get('response')),
                'Data' => JSON.serialize(new Map<String, Object>{
                    'endpoint' => result.get('endpoint'),
                    'duration' => result.get('duration'),
                    'statusCode' => result.get('statusCode')
                })
            };

            events.add(eventData);
        }

        // Guardar asincrónicamente
        String serializedEvents = JSON.serialize(events);
        EventLogManager.saveAsync(serializedEvents);

        System.debug('Enviados ' + events.size() + ' eventos para procesamiento asíncrono');
    }
}

// Uso
List<Map<String, Object>> integrationResults = new List<Map<String, Object>>{
    new Map<String, Object>{
        'success' => true,
        'product' => 'Insurance',
        'message' => 'Póliza creada exitosamente',
        'endpoint' => '/api/v1/policies',
        'duration' => 1250,
        'statusCode' => 200,
        'response' => new Map<String, Object>{'policyId' => 'POL-123456'}
    },
    new Map<String, Object>{
```

```
        'success' => false,
        'product' => 'Insurance',
        'message' => 'Error al validar datos del cliente',
        'endpoint' => '/api/v1/customers/validate',
        'duration' => 500,
        'statusCode' => 400,
        'response' => new Map<String, Object>{'error' => 'Invalid customer data'}
    }
};

BatchEventLogger.logIntegrationEvents(integrationResults);
```

Ejemplo 3: Análisis y Consulta de Eventos

```
public class EventAnalytics {

    public static Map<String, Object> generateDailyReport(Date reportDate) {
        Datetime startOfDay = Datetime.newInstance(reportDate, Time.newInstance(0, 0, 0, 0));
        Datetime endOfDay = Datetime.newInstance(reportDate, Time.newInstance(23, 59, 59, 999));

        // Obtener eventos del día
        List<EventLog__b> dailyEvents = EventLogManager.fetchAll(startOfDay, endOfDay);

        // Análisis básico
        Map<String, Integer> eventsByLevel = new Map<String, Integer>();
        Map<String, Integer> eventsByProduct = new Map<String, Integer>();
        Map<String, Integer> eventsByType = new Map<String, Integer>();

        for (EventLog__b event : dailyEvents) {
            // Contar por nivel
            String level = event.Level__c;
            eventsByLevel.put(level, eventsByLevel.containsKey(level) ?
                              eventsByLevel.get(level) + 1 : 1);

            // Contar por producto
            String product = event.Product__c;
            eventsByProduct.put(product, eventsByProduct.containsKey(product) ?
                                eventsByProduct.get(product) + 1 : 1);

            // Contar por tipo
            String type = event.Type__c;
            eventsByType.put(type, eventsByType.containsKey(type) ?
                             eventsByType.get(type) + 1 : 1);
        }

        return new Map<String, Object>{
            'date' => reportDate,
            'totalEvents' => dailyEvents.size(),
            'eventsByLevel' => eventsByLevel,
            'eventsByProduct' => eventsByProduct,
            'eventsByType' => eventsByType,
            'generatedAt' => System.now()
        };
    }

    public static void refreshAnalyticsForPeriod(Date startDate, Date endDate) {
        Datetime startDatetime = Datetime.newInstance(startDate, Time.newInstance(0, 0, 0, 0));
```

```
        Datetime endDatetime = Datetime.newInstance(endDate, Time.newInstance(23, 59, 59, 999));

        // Refrescar análisis asincrónicamente
        EventLogManager.refreshAnalyticsAsync(startDatetime, endDatetime, true);

        System.debug('Actualización de análisis iniciada para el período: ' +
                    startDate + ' - ' + endDate);
    }
}

// Uso
Map<String, Object> dailyReport = EventAnalytics.generateDailyReport(Date.today());
System.debug('Reporte diario: ' + JSON.serializePretty(dailyReport));

// Refrescar análisis para la última semana
Date lastWeek = Date.today().addDays(-7);
EventAnalytics.refreshAnalyticsForPeriod(lastWeek, Date.today());
```

Ejemplo 4: Mantenimiento y Limpieza de Logs

```
public class EventLogMaintenance {

    @InvocableMethod(label='Limpiar Logs Antiguos' description='Elimina logs de eventos antiguos')
    public static void cleanupOldLogs() {
        // Limpiar logs del Big Object asincrónicamente
        EventLogManager.clearLogsAsync();

        System.debug('Proceso de limpieza de logs iniciado');
    }

    public static void scheduleAnalyticsRefresh() {
        // Refrescar análisis para los últimos 30 días
        Datetime thirtyDaysAgo = Datetime.now().addDays(-30);
        Datetime now = Datetime.now();

        EventLogManager.refreshAnalyticsAsync(thirtyDaysAgo, now, false);

        System.debug('Actualización programada de análisis iniciada');
    }

    public static void clearAnalyticsData() {
        Boolean success = EventLogManager.clearAnalytics();

        if (success) {
            System.debug('Datos analíticos limpiados exitosamente');
        } else {
            System.debug('Error al limpiar datos analíticos');
        }
    }
}

// Programar limpieza semanal
public class EventLogMaintenanceSchedule implements Schedulable {
    public void execute(SchedulableContext sc) {
        EventLogMaintenance.cleanupOldLogs();
        EventLogMaintenance.scheduleAnalyticsRefresh();
    }
```

```
}

// Programar la limpieza para ejecutarse cada domingo a las 2 AM
String cronExp = '0 0 2 ? * SUN';
String jobId = System.schedule('Weekly Event Log Cleanup', cronExp, new EventLogMaintenanceSchedule());
```

Mejores Prácticas Implementadas

1. **Procesamiento por Lotes**: La clase procesa registros en lotes de 200 para optimizar el rendimiento.
2. **Validación de Datos**: Valida campos obligatorios antes de procesar los eventos.
3. **Feature Flags**: Utiliza feature flags para controlar la funcionalidad de logging.
4. **Procesamiento Asíncrono**: Proporciona métodos asíncronos para operaciones que pueden tomar tiempo.
5. **Manejo de Excepciones**: Incluye manejo robusto de excepciones con logging de errores.
6. **Soporte para Testing**: Incluye verificaciones para contextos de prueba y usa mocks.

Consideraciones de Rendimiento

1. **Límites de Consulta**: Las consultas están limitadas (50,000 para fetch, 5,000 para limpieza) para evitar timeouts.
2. **Big Objects**: Utiliza Big Objects para almacenar grandes volúmenes de datos de eventos de forma eficiente.
3. **Procesamiento Asíncrono**: Los métodos future permiten procesar operaciones pesadas sin bloquear la transacción principal.
4. **Lotes de Inserción**: Procesa inserciones en lotes para optimizar el uso de recursos.

Observaciones sobre el Código

Código Comentado

La clase contiene código comentado que sugiere implementaciones alternativas o funcionalidades no activas:

- Limitaciones de longitud de campos comentadas
- Lógica de procesamiento por lotes comentada en algunos métodos
- Diferentes enfoques para eliminar registros

Mejoras Potenciales

1. **Validación de Entrada**: Podría beneficiarse de validaciones más robustas de los datos de entrada.
2. **Gestión de Errores**: Mejorar el reporte de errores específicos durante las operaciones batch.
3. **Configuración Externalizada**: Hacer configurables los límites de registros y tamaños de lote.
4. **Métricas de Rendimiento**: Añadir métricas para monitorear el rendimiento de las operaciones.

Notas Adicionales

1. **Seguridad**: La clase utiliza `WITH SECURITY_ENFORCED` en algunas consultas para respetar las reglas de seguridad.
2. **Dual Storage**: Mantiene datos tanto en Big Objects (para volumen) como en objetos estándar (para análisis).
3. **Feature Flag Integration**: La funcionalidad de guardado depende de la feature flag 'sura_foundation_event_logs'.

4. **Testing Support**: Incluye soporte completo para testing con mocks y verificaciones de contexto.

5. **Flexibilidad**: El diseño permite diferentes tipos de eventos y productos, siendo flexible para diversos casos de uso.

CustomMetadataCallback

Descripción General

`CustomMetadataCallback` es una clase que implementa la interfaz `Metadata.DeployCallback` de Salesforce. Esta clase está diseñada para manejar los resultados de operaciones de despliegue de metadatos personalizados, proporcionando funcionalidad de callback que se ejecuta cuando una operación de despliegue de metadatos se completa (exitosa o fallida).

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

28 de octubre de 2024

Estructura de la Clase

```
public class CustomMetadataCallback implements Metadata.DeployCallback {
  public void handleResult(
    Metadata.DeployResult result,
    Metadata.DeployCallbackContext context
  ) {
    if (!Test.isRunningTest()) {
      Core.debug(
        String.format(
          '[{0}] Metadata updated job queued with jobId="{1}"',
          new List<String>{
            result.status == Metadata.DeployStatus.Succeeded
              ? 'Succeeded'
              : 'Failed',
            result.status == Metadata.DeployStatus.Succeeded
              ? context.getCallbackJobId().toString()
              : 'No created job'
          }
        )
      );
    }
  }
}
```

Interfaz Implementada

Metadata.DeployCallback

La clase implementa la interfaz `Metadata.DeployCallback` de Salesforce, que es parte de la API de Metadatos. Esta interfaz define el método `handleResult` que debe ser implementado para procesar los resultados de operaciones de despliegue de metadatos.

Métodos

handleResult(Metadata.DeployResult result, Metadata.DeployCallbackContext context)

Descripción

Método que se ejecuta automáticamente cuando una operación de despliegue de metadatos se completa. Procesa el resultado de la operación y registra información sobre el estado del despliegue.

Parámetros

- `result` (Metadata.DeployResult): Objeto que contiene información sobre el resultado del despliegue de metadatos.
- `context` (Metadata.DeployCallbackContext): Contexto del callback que proporciona información adicional sobre la operación.

Retorno

- `void`: Este método no devuelve ningún valor.

Proceso

1. Verifica si el código no se está ejecutando en un contexto de prueba.
2. Determina el estado del despliegue (Succeeded o Failed).
3. Obtiene el ID del trabajo de callback si el despliegue fue exitoso.
4. Formatea y registra un mensaje de depuración con la información del resultado.

Código

```
public void handleResult(
  Metadata.DeployResult result,
  Metadata.DeployCallbackContext context
) {
  if (!Test.isRunningTest()) {
    Core.debug(
      String.format(
        '[{0}] Metadata updated job queued with jobId="{1}"',
        new List<String>{
          result.status == Metadata.DeployStatus.Succeeded
            ? 'Succeeded'
            : 'Failed',
          result.status == Metadata.DeployStatus.Succeeded
            ? context.getCallbackJobId().toString()
            : 'No created job'
        }
      )
    );
  }
}
```

Dependencias

- `Metadata.DeployCallback`: Interfaz estándar de Salesforce para callbacks de despliegue de metadatos.
- `Metadata.DeployResult`: Clase que encapsula el resultado de una operación de despliegue de metadatos.
- `Metadata.DeployCallbackContext`: Clase que proporciona contexto adicional para el callback.
- `Core`: Clase utilitaria que proporciona el método `debug` para registro de actividad.

Propósito y Casos de Uso

Esta clase es especialmente útil en escenarios como:

1. **Despliegue Programático de Metadatos**: Cuando se necesita desplegar metadatos personalizados desde código Apex y monitorear el resultado.
2. **Operaciones Asíncronas**: Para manejar operaciones de metadatos que se ejecutan de forma asíncrona.

3. **Auditoría y Registro**: Para mantener un registro de las operaciones de despliegue de metadatos y su estado.
4. **Integración con Procesos de Negocio**: Cuando las actualizaciones de metadatos forman parte de procesos de negocio más amplios.

Contexto de la API de Metadatos de Salesforce

La API de Metadatos de Salesforce permite crear, actualizar y eliminar metadatos de forma programática. Los callbacks como este se utilizan porque las operaciones de metadatos son inherentemente asíncronas y pueden tomar tiempo en completarse.

Ejemplos de Uso

Ejemplo 1: Despliegue de Metadatos Personalizados con Callback

```
public class MetadataDeploymentService {

    public static void deployCustomMetadata(String metadataName, Map<String, Object> fieldValues) {
        try {
            // Crear el registro de metadatos personalizados
            Metadata.CustomMetadata customMetadata = new Metadata.CustomMetadata();
            customMetadata.fullName = metadataName;
            customMetadata.label = metadataName;

            // Añadir valores de campos
            for (String fieldName : fieldValues.keySet()) {
                Metadata.CustomMetadataValue metadataValue = new Metadata.CustomMetadataValue();
                metadataValue.field = fieldName;
                metadataValue.value = fieldValues.get(fieldName);
                customMetadata.values.add(metadataValue);
            }

            // Crear el contenedor de despliegue
            Metadata.DeployContainer deployContainer = new Metadata.DeployContainer();
            deployContainer.addMetadata(customMetadata);

            // Crear la instancia del callback
            CustomMetadataCallback callback = new CustomMetadataCallback();

            // Ejecutar el despliegue con el callback
            Id deployJobId = Metadata.Operations.enqueueDeployment(deployContainer, callback);

            System.debug('Despliegue de metadatos iniciado con ID: ' + deployJobId);
        } catch (Exception e) {
            System.debug('Error al desplegar metadatos: ' + e.getMessage());
            throw e;
        }
    }
}

// Uso del servicio
Map<String, Object> configValues = new Map<String, Object>{
    'API_Endpoint__c' => 'https://api.example.com',
    'Timeout__c' => 30000,
    'Is_Active__c' => true
};

MetadataDeploymentService.deployCustomMetadata('MyConfig.Production', configValues);
```

Ejemplo 2: Clase de Callback Extendida con Funcionalidad Adicional

```
public class EnhancedCustomMetadataCallback implements Metadata.DeployCallback {

    private String operationId;
    private String operationType;

    public EnhancedCustomMetadataCallback(String operationId, String operationType) {
        this.operationId = operationId;
        this.operationType = operationType;
    }

    public void handleResult(
        Metadata.DeployResult result,
        Metadata.DeployCallbackContext context
    ) {
        if (!Test.isRunningTest()) {
            // Registro detallado del resultado
            logDeploymentResult(result, context);

            // Enviar notificaciones según el resultado
            if (result.status == Metadata.DeployStatus.Succeeded) {
                handleSuccessfulDeployment(result, context);
            } else {
                handleFailedDeployment(result, context);
            }
        }
    }

    private void logDeploymentResult(
        Metadata.DeployResult result,
        Metadata.DeployCallbackContext context
    ) {
        String logMessage = String.format(
            '[{0}] Metadata deployment {1} - Operation: {2}, JobId: {3}',
            new List<String>{
                result.status.name(),
                result.status == Metadata.DeployStatus.Succeeded ? 'completed successfully' : 'failed',
                this.operationType,
                result.status == Metadata.DeployStatus.Succeeded
                    ? context.getCallbackJobId().toString()
                    : 'N/A'
            }
        );

        Core.debug(logMessage);

        // También registrar detalles adicionales si están disponibles
        if (result.details != null) {
            Core.debug('Deployment details: ' + JSON.serialize(result.details));
        }
    }

    private void handleSuccessfulDeployment(
        Metadata.DeployResult result,
        Metadata.DeployCallbackContext context
```

```
    ) {
        try {
            // Crear registro de auditoría
            createAuditRecord('SUCCESS', result, context);

            // Enviar notificación de éxito
            sendNotification('Deployment Successful',
                             'Metadata deployment completed successfully for operation: ' + this.operationType);

            // Ejecutar acciones post-despliegue si las hay
            executePostDeploymentActions();
        } catch (Exception e) {
            Core.debug('Error in success handler: ' + e.getMessage());
        }
    }

    private void handleFailedDeployment(
        Metadata.DeployResult result,
        Metadata.DeployCallbackContext context
    ) {
        try {
            // Crear registro de auditoría para el fallo
            createAuditRecord('FAILED', result, context);

            // Enviar notificación de error
            sendNotification('Deployment Failed',
                             'Metadata deployment failed for operation: ' + this.operationType);

            // Ejecutar acciones de rollback si las hay
            executeRollbackActions();
        } catch (Exception e) {
            Core.debug('Error in failure handler: ' + e.getMessage());
        }
    }

    private void createAuditRecord(
        String status,
        Metadata.DeployResult result,
        Metadata.DeployCallbackContext context
    ) {
        // Ejemplo de creación de registro de auditoría
        // Esto podría ser un objeto personalizado para tracking de deployments
        /*
        Metadata_Deployment_Log__c auditRecord = new Metadata_Deployment_Log__c(
            Operation_Id__c = this.operationId,
            Operation_Type__c = this.operationType,
            Status__c = status,
            Job_Id__c = context.getCallbackJobId()?.toString(),
            Deployment_Date__c = System.now(),
            Details__c = result.details != null ? JSON.serialize(result.details) : null
        );

        insert auditRecord;
        */
    }

    private void sendNotification(String subject, String body) {
```

```
        // Implementar lógica de notificación
        // Esto podría incluir emails, platform events, etc.
    }

    private void executePostDeploymentActions() {
        // Ejecutar acciones específicas después de un despliegue exitoso
        // Por ejemplo, actualizar caché, notificar a otros sistemas, etc.
    }

    private void executeRollbackActions() {
        // Ejecutar acciones de rollback en caso de fallo
        // Por ejemplo, restaurar configuraciones anteriores, notificar errores, etc.
    }
}
```

Ejemplo 3: Uso en un Proceso de Configuración Dinámico

```
public class DynamicConfigurationManager {

    public class ConfigurationItem {
        public String name;
        public String value;
        public String type;

        public ConfigurationItem(String name, String value, String type) {
            this.name = name;
            this.value = value;
            this.type = type;
        }
    }

    public static void updateConfiguration(
        String configurationSetName,
        List<ConfigurationItem> items
    ) {
        try {
            // Crear metadatos para cada item de configuración
            Metadata.DeployContainer deployContainer = new Metadata.DeployContainer();

            for (ConfigurationItem item : items) {
                Metadata.CustomMetadata customMetadata = createMetadataFromItem(
                    configurationSetName, item
                );
                deployContainer.addMetadata(customMetadata);
            }

            // Crear callback con información contextual
            EnhancedCustomMetadataCallback callback = new EnhancedCustomMetadataCallback(
                generateOperationId(),
                'CONFIGURATION_UPDATE'
            );

            // Ejecutar el despliegue
            Id deployJobId = Metadata.Operations.enqueueDeployment(deployContainer, callback);

            System.debug('Configuration update initiated with job ID: ' + deployJobId);
        } catch (Exception e) {
```

```
            System.debug('Error updating configuration: ' + e.getMessage());
            throw new ConfigurationException('Failed to update configuration: ' + e.getMessage(), e);
        }
    }

    private static Metadata.CustomMetadata createMetadataFromItem(
        String setName,
        ConfigurationItem item
    ) {
        Metadata.CustomMetadata customMetadata = new Metadata.CustomMetadata();
        customMetadata.fullName = setName + '.' + item.name;
        customMetadata.label = item.name;

        // Añadir valor del item
        Metadata.CustomMetadataValue valueField = new Metadata.CustomMetadataValue();
        valueField.field = 'Value__c';
        valueField.value = item.value;
        customMetadata.values.add(valueField);

        // Añadir tipo del item
        Metadata.CustomMetadataValue typeField = new Metadata.CustomMetadataValue();
        typeField.field = 'Type__c';
        typeField.value = item.type;
        customMetadata.values.add(typeField);

        return customMetadata;
    }

    private static String generateOperationId() {
        return 'OP_' + String.valueOf(System.currentTimeMillis());
    }

    public class ConfigurationException extends Exception {}
}

// Uso del manager
List<DynamicConfigurationManager.ConfigurationItem> configItems =
    new List<DynamicConfigurationManager.ConfigurationItem>{
        new DynamicConfigurationManager.ConfigurationItem('API_URL', 'https://new-api.com', 'String'),
        new DynamicConfigurationManager.ConfigurationItem('TIMEOUT', '45000', 'Number'),
        new DynamicConfigurationManager.ConfigurationItem('ENABLED', 'true', 'Boolean')
    };

DynamicConfigurationManager.updateConfiguration('ProductionConfig', configItems);
```

Mejoras Propuestas para la Implementación Actual

Versión Mejorada con Funcionalidad Adicional

```
/**
 * @description       : Callback para manejar resultados de despliegue de metadatos personalizados
 * @author            : Soulberto Lorenzo <soulberto@cloudblue.us>
 * @group             : Metadata Management
 * @last modified on  : 10-28-2024
 * @last modified by  : Soulberto Lorenzo <soulberto@cloudblue.us>
 **/
```

```
public class CustomMetadataCallback implements Metadata.DeployCallback {

    private String operationContext;
    private Map<String, Object> additionalData;

    /**
     * Constructor por defecto
     */
    public CustomMetadataCallback() {
        this.operationContext = 'DEFAULT';
        this.additionalData = new Map<String, Object>();
    }

    /**
     * Constructor con contexto
     * @param operationContext Contexto de la operación
     */
    public CustomMetadataCallback(String operationContext) {
        this.operationContext = operationContext;
        this.additionalData = new Map<String, Object>();
    }

    /**
     * Constructor con contexto y datos adicionales
     * @param operationContext Contexto de la operación
     * @param additionalData Datos adicionales para el callback
     */
    public CustomMetadataCallback(String operationContext, Map<String, Object> additionalData) {
        this.operationContext = operationContext;
        this.additionalData = additionalData != null ? additionalData : new Map<String, Object>();
    }

    /**
     * Maneja el resultado del despliegue de metadatos
     * @param result Resultado del despliegue
     * @param context Contexto del callback
     */
    public void handleResult(
        Metadata.DeployResult result,
        Metadata.DeployCallbackContext context
    ) {
        if (!Test.isRunningTest()) {
            try {
                // Registro básico del resultado
                logBasicResult(result, context);

                // Registro detallado si está disponible
                logDetailedResult(result);

                // Procesar según el estado del resultado
                if (result.status == Metadata.DeployStatus.Succeeded) {
                    handleSuccess(result, context);
                } else {
                    handleFailure(result, context);
                }
            } catch (Exception e) {
                Core.debug('Error in CustomMetadataCallback.handleResult: ' + e.getMessage());
```

```
            logCallbackError(e);
        }
    }
}

/**
 * Registra información básica del resultado
 */
private void logBasicResult(Metadata.DeployResult result, Metadata.DeployCallbackContext context) {
    String statusText = result.status == Metadata.DeployStatus.Succeeded ? 'Succeeded' : 'Failed';
    String jobId = result.status == Metadata.DeployStatus.Succeeded
        ? context.getCallbackJobId()?.toString()
        : 'No created job';

    Core.debug(
        String.format(
            '[{0}] Metadata deployment {1} - Context: {2}, JobId: {3}',
            new List<String>{ statusText, statusText.toLowerCase(), this.operationContext, jobId }
        )
    );
}

/**
 * Registra información detallada del resultado si está disponible
 */
private void logDetailedResult(Metadata.DeployResult result) {
    if (result.details != null) {
        // Registrar detalles específicos del despliegue
        Core.debug('Deployment details available - Components: ' +
                   (result.details.componentSuccesses?.size() ?? 0) + ' successful, ' +
                   (result.details.componentFailures?.size() ?? 0) + ' failed');

        // Registrar errores específicos si los hay
        if (result.details.componentFailures != null && !result.details.componentFailures.isEmpty()) {
            for (Metadata.DeployMessage failure : result.details.componentFailures) {
                Core.debug('Component failure: ' + failure.fullName + ' - ' + failure.problem);
            }
        }
    }
}

/**
 * Maneja el caso de despliegue exitoso
 */
private void handleSuccess(Metadata.DeployResult result, Metadata.DeployCallbackContext context) {
    Core.debug('Metadata deployment completed successfully for context: ' + this.operationContext);

    // Ejecutar acciones post-éxito si están definidas en additionalData
    if (additionalData.containsKey('onSuccess') && additionalData.get('onSuccess') != null) {
        executeCallback('onSuccess');
    }
}

/**
 * Maneja el caso de despliegue fallido
 */
private void handleFailure(Metadata.DeployResult result, Metadata.DeployCallbackContext context) {
```

```
        Core.debug('Metadata deployment failed for context: ' + this.operationContext);

        // Ejecutar acciones post-fallo si están definidas en additionalData
        if (additionalData.containsKey('onFailure') && additionalData.get('onFailure') != null) {
            executeCallback('onFailure');
        }
    }

    /**
     * Ejecuta callbacks adicionales definidos en additionalData
     */
    private void executeCallback(String callbackType) {
        try {
            // Aquí se podría implementar lógica para ejecutar callbacks específicos
            // Por ejemplo, invocar métodos específicos, enviar platform events, etc.
            Core.debug('Executing ' + callbackType + ' callback for context: ' + this.operationContext);
        } catch (Exception e) {
            Core.debug('Error executing ' + callbackType + ' callback: ' + e.getMessage());
        }
    }

    /**
     * Registra errores del callback mismo
     */
    private void logCallbackError(Exception e) {
        Core.debug('CustomMetadataCallback error - Context: ' + this.operationContext +
                   ', Error: ' + e.getMessage() + ', Stack: ' + e.getStackTraceString());
    }
}
```

Consideraciones de Rendimiento y Limitaciones

1. **Límites de API de Metadatos**: Las operaciones de metadatos están sujetas a límites específicos de Salesforce. Es importante estar consciente de estos límites al diseñar soluciones.
2. **Asincronía**: Las operaciones de metadatos son inherentemente asíncronas. El callback se ejecuta cuando la operación se completa, no inmediatamente.
3. **Contexto de Ejecución**: Los callbacks se ejecutan en un contexto específico y pueden tener limitaciones en cuanto a las operaciones que pueden realizar.
4. **Manejo de Errores**: Es importante incluir manejo robusto de errores en los callbacks para evitar que fallos en el callback afecten la operación principal.

Pruebas Unitarias

```
@IsTest
private class CustomMetadataCallbackTest {

    @IsTest
    static void testSuccessfulDeployment() {
        Test.startTest();

        // Crear mock del resultado exitoso
        Metadata.DeployResult mockResult = new Metadata.DeployResult();
        mockResult.status = Metadata.DeployStatus.Succeeded;
```

```
        // Crear mock del contexto
        // Nota: En pruebas reales, necesitarías usar mocks más sofisticados
        // ya que Metadata.DeployCallbackContext no se puede instanciar directamente

        CustomMetadataCallback callback = new CustomMetadataCallback();

        // Ejecutar el método (no debería registrar nada durante las pruebas)
        callback.handleResult(mockResult, null);

        Test.stopTest();

        // En un escenario real, verificarías los efectos secundarios
        // como registros creados, notificaciones enviadas, etc.
        System.assert(true, 'Callback should execute without errors');
    }

    @IsTest
    static void testFailedDeployment() {
        Test.startTest();

        // Crear mock del resultado fallido
        Metadata.DeployResult mockResult = new Metadata.DeployResult();
        mockResult.status = Metadata.DeployStatus.Failed;

        CustomMetadataCallback callback = new CustomMetadataCallback();

        // Ejecutar el método
        callback.handleResult(mockResult, null);

        Test.stopTest();

        // Verificar que el método maneja correctamente los fallos
        System.assert(true, 'Callback should handle failures gracefully');
    }
}
```

Notas Adicionales

1. Esta clase forma parte del ecosistema de gestión de metadatos de la organización y trabaja en conjunto con la API de Metadatos de Salesforce.
2. La verificación `!Test.isRunningTest()` es importante para evitar registrar información durante las pruebas unitarias.
3. El uso de `Core.debug()` sugiere que hay una clase utilitaria centralizada para el registro de actividad.
4. Esta implementación es básica pero funcional, y puede ser extendida según las necesidades específicas del proyecto.
5. Las operaciones de metadatos son poderosas pero deben usarse con cuidado, especialmente en entornos de producción.

CumuloMissingParameterException

Descripción General

`CumuloMissingParameterException` es una clase de excepción personalizada que extiende la clase `Exception` estándar de Apex. Esta clase está diseñada específicamente para representar errores que ocurren cuando faltan parámetros requeridos en operaciones relacionadas con el framework Cumulo o procesos específicos que requieren validación de parámetros obligatorios.

Autor

Felipe Correa \<felipe.correa@nespon.com\>

Proyecto

Foundation Salesforce - Suratech - UH-17589

Última Modificación

21 de noviembre de 2024

Estructura

```
public with sharing class CumuloMissingParameterException extends Exception {
}
```

La clase tiene una implementación minimalista, sin agregar métodos o propiedades adicionales más allá de los heredados de la clase base `Exception`.

Herencia

- **Clase Base**: `Exception`
- **Modificadores**: `public with sharing`

Propósito

El propósito principal de esta clase es proporcionar un tipo de excepción específico para errores relacionados con parámetros faltantes en el contexto del framework Cumulo, permitiendo:

1. **Especificidad del Error**: Distinguir claramente los errores de parámetros faltantes de otros tipos de excepciones en el sistema.
2. **Validación de Entrada**: Facilitar la validación de parámetros requeridos en métodos y procesos críticos.
3. **Manejo Específico**: Permitir que el código que captura excepciones pueda identificar y manejar específicamente casos donde faltan parámetros obligatorios.
4. **Claridad en el Código**: Mejorar la legibilidad del código al indicar explícitamente cuando se está tratando con un error de parámetro faltante.
5. **Debugging y Mantenimiento**: Facilitar la identificación de problemas relacionados con configuración o llamadas incorrectas a métodos.

Contexto del Framework Cumulo

El nombre "Cumulo" sugiere que esta excepción forma parte de un framework o sistema específico dentro de la organización Suratech. Los frameworks suelen requerir configuraciones específicas y parámetros obligatorios para funcionar correctamente, por lo que tener una excepción específica para parámetros faltantes es una práctica común y recomendada.

Métodos Heredados

Al extender la clase `Exception` estándar, `CumuloMissingParameterException` hereda los siguientes métodos y propiedades:

Constructores

- **`CumuloMissingParameterException()`**: Constructor por defecto.
- **`CumuloMissingParameterException(String message)`**: Constructor que acepta un mensaje de error.
- **`CumuloMissingParameterException(Exception cause)`**: Constructor que acepta otra excepción como causa.
- **`CumuloMissingParameterException(String message, Exception cause)`**: Constructor que acepta un mensaje y otra excepción como causa.

Métodos

- **`getMessage()`**: Devuelve el mensaje de error asociado con la excepción.
- **`getCause()`**: Devuelve la excepción que causó esta excepción, si existe.
- **`getTypeName()`**: Devuelve el nombre del tipo de excepción.
- **`setMessage(String message)`**: Establece el mensaje de error para la excepción.
- **`getLineNumber()`**: Devuelve el número de línea donde se generó la excepción.
- **`getStackTraceString()`**: Devuelve una representación de texto del seguimiento de la pila.

Ejemplos de Uso

Ejemplo 1: Validación de Parámetros de Configuración

```
public class CumuloConfigurationManager {

    public static void initializeFramework(Map<String, Object> config) {
        // Validar parámetros requeridos
        validateRequiredParameters(config);

        // Proceder con la inicialización
        String endpoint = (String) config.get('endpoint');
        String apiKey = (String) config.get('apiKey');
        Integer timeout = (Integer) config.get('timeout');

        // Lógica de inicialización del framework
        // ...
    }

    private static void validateRequiredParameters(Map<String, Object> config) {
        if (config == null) {
            throw new CumuloMissingParameterException('La configuración no puede ser nula');
        }

        List<String> requiredParams = new List<String>{'endpoint', 'apiKey', 'timeout'};

        for (String param : requiredParams) {
            if (!config.containsKey(param) || config.get(param) == null) {
                throw new CumuloMissingParameterException(
                    'Parámetro requerido faltante: ' + param
```

```
                );
            }

            if (config.get(param) instanceof String && String.isBlank((String)config.get(param))) {
                throw new CumuloMissingParameterException(
                    'El parámetro ' + param + ' no puede estar vacío'
                );
            }
        }
    }
}

// Uso
try {
    Map<String, Object> config = new Map<String, Object>{
        'endpoint' => 'https://api.cumulo.com',
        'apiKey' => 'sk_test_123456'
        // Falta el parámetro 'timeout'
    };

    CumuloConfigurationManager.initializeFramework(config);
} catch (CumuloMissingParameterException e) {
    System.debug('Error de configuración: ' + e.getMessage());
    // Manejar el error específico de parámetro faltante
}
```

Ejemplo 2: Validación en Métodos de Servicio

```
public class CumuloDataService {

    public static Map<String, Object> processData(
        String recordId,
        String operation,
        Map<String, Object> data
    ) {
        // Validar parámetros de entrada
        validateInputParameters(recordId, operation, data);

        // Procesar los datos
        Map<String, Object> result = new Map<String, Object>();

        switch on operation.toUpperCase() {
            when 'CREATE' {
                result = createRecord(data);
            }
            when 'UPDATE' {
                result = updateRecord(recordId, data);
            }
            when 'DELETE' {
                result = deleteRecord(recordId);
            }
            when else {
                throw new IllegalArgumentException('Operación no soportada: ' + operation);
            }
        }

        return result;
```

```
    }

    private static void validateInputParameters(
        String recordId,
        String operation,
        Map<String, Object> data
    ) {
        if (String.isBlank(operation)) {
            throw new CumuloMissingParameterException('El parámetro "operation" es requerido');
        }

        Set<String> operationsRequiringId = new Set<String>{'UPDATE', 'DELETE'};
        if (operationsRequiringId.contains(operation.toUpperCase()) && String.isBlank(recordId)) {
            throw new CumuloMissingParameterException(
                'El parámetro "recordId" es requerido para la operación ' + operation
            );
        }

        Set<String> operationsRequiringData = new Set<String>{'CREATE', 'UPDATE'};
        if (operationsRequiringData.contains(operation.toUpperCase())) {
            if (data == null || data.isEmpty()) {
                throw new CumuloMissingParameterException(
                    'El parámetro "data" es requerido para la operación ' + operation
                );
            }
        }
    }

    private static Map<String, Object> createRecord(Map<String, Object> data) {
        // Lógica de creación
        return new Map<String, Object>{'success' => true, 'id' => 'new_record_id'};
    }

    private static Map<String, Object> updateRecord(String recordId, Map<String, Object> data) {
        // Lógica de actualización
        return new Map<String, Object>{'success' => true, 'id' => recordId};
    }

    private static Map<String, Object> deleteRecord(String recordId) {
        // Lógica de eliminación
        return new Map<String, Object>{'success' => true, 'deleted' => true};
    }
}

// Uso
try {
    // Esto lanzará CumuloMissingParameterException porque falta recordId para UPDATE
    CumuloDataService.processData(null, 'UPDATE', new Map<String, Object>{'name' => 'Test'});
} catch (CumuloMissingParameterException e) {
    System.debug('Error: ' + e.getMessage());
    // Manejar error de parámetro faltante
}
```

Ejemplo 3: Validación en Procedimientos de Integración

```
public class CumuloIntegrationProcedure {
```

```
public static Map<String, Object> executeIntegration(Map<String, Object> input) {
    try {
        // Validar estructura de entrada
        validateIntegrationInput(input);

        // Extraer parámetros validados
        String integrationPoint = (String) input.get('integrationPoint');
        Map<String, Object> payload = (Map<String, Object>) input.get('payload');
        Map<String, Object> headers = (Map<String, Object>) input.get('headers');

        // Ejecutar la integración
        return performIntegration(integrationPoint, payload, headers);

    } catch (CumuloMissingParameterException e) {
        // Devolver error estructurado
        return new Map<String, Object>{
            'success' => false,
            'errorType' => 'MISSING_PARAMETER',
            'errorMessage' => e.getMessage()
        };
    } catch (Exception e) {
        // Manejar otros errores
        return new Map<String, Object>{
            'success' => false,
            'errorType' => 'INTEGRATION_ERROR',
            'errorMessage' => e.getMessage()
        };
    }
}

private static void validateIntegrationInput(Map<String, Object> input) {
    if (input == null) {
        throw new CumuloMissingParameterException('Los datos de entrada no pueden ser nulos');
    }

    // Validar integrationPoint
    if (!input.containsKey('integrationPoint') ||
        String.isBlank((String)input.get('integrationPoint'))) {
        throw new CumuloMissingParameterException(
            'El parámetro "integrationPoint" es requerido'
        );
    }

    // Validar payload
    if (!input.containsKey('payload') || input.get('payload') == null) {
        throw new CumuloMissingParameterException(
            'El parámetro "payload" es requerido'
        );
    }

    // Validar que el payload sea un mapa
    if (!(input.get('payload') instanceof Map<String, Object>)) {
        throw new CumuloMissingParameterException(
            'El parámetro "payload" debe ser un objeto válido'
        );
    }
```

```
        // Headers es opcional, pero si está presente debe ser un mapa
        if (input.containsKey('headers') && input.get('headers') != null) {
            if (!(input.get('headers') instanceof Map<String, Object>)) {
                throw new CumuloMissingParameterException(
                    'El parámetro "headers" debe ser un objeto válido si se proporciona'
                );
            }
        }
    }

    private static Map<String, Object> performIntegration(
        String integrationPoint,
        Map<String, Object> payload,
        Map<String, Object> headers
    ) {
        // Lógica de integración
        return new Map<String, Object>{
            'success' => true,
            'integrationPoint' => integrationPoint,
            'processedAt' => System.now()
        };
    }
}
```

Ejemplo 4: Validación con Información Detallada

```
public class CumuloValidator {

    public static void validateBusinessObject(Map<String, Object> businessObject, String objectType) {
        if (String.isBlank(objectType)) {
            throw new CumuloMissingParameterException('El tipo de objeto es requerido');
        }

        // Definir campos requeridos por tipo de objeto
        Map<String, List<String>> requiredFieldsByType = new Map<String, List<String>>{
            'CUSTOMER' => new List<String>{'name', 'email', 'phone'},
            'PRODUCT' => new List<String>{'name', 'sku', 'price'},
            'ORDER' => new List<String>{'customerId', 'productIds', 'totalAmount'}
        };

        if (!requiredFieldsByType.containsKey(objectType.toUpperCase())) {
            throw new IllegalArgumentException('Tipo de objeto no soportado: ' + objectType);
        }

        List<String> requiredFields = requiredFieldsByType.get(objectType.toUpperCase());
        List<String> missingFields = new List<String>();

        for (String field : requiredFields) {
            if (!businessObject.containsKey(field) ||
                businessObject.get(field) == null ||
                (businessObject.get(field) instanceof String &&
                 String.isBlank((String)businessObject.get(field)))) {
                missingFields.add(field);
            }
        }

        if (!missingFields.isEmpty()) {
```

```
            throw new CumuloMissingParameterException(
                'Faltan los siguientes campos requeridos para ' + objectType + ': ' +
                String.join(missingFields, ', ')
            );
        }
    }
}

// Uso
try {
    Map<String, Object> customer = new Map<String, Object>{
        'name' => 'Juan Pérez',
        'email' => 'juan@example.com'
        // Falta 'phone'
    };

    CumuloValidator.validateBusinessObject(customer, 'CUSTOMER');
} catch (CumuloMissingParameterException e) {
    System.debug('Error de validación: ' + e.getMessage());
    // Resultado: "Faltan los siguientes campos requeridos para CUSTOMER: phone"
}
```

Implementación Extendida Propuesta

Aunque la implementación actual es minimalista, podría extenderse para incluir información más específica:

```
/**
 * @description       : Excepción para parámetros faltantes en el framework Cumulo
 * @author            : felipe.correa@nespon.com
 * @project           : Foundation Salesforce - Suratech - UH-17589
 * @last modified on  : 11-21-2024
 * @last modified by  : felipe.correa@nespon.com
 **/
public with sharing class CumuloMissingParameterException extends Exception {

    private String parameterName;
    private String parameterType;
    private String contextMethod;
    private List<String> missingParameters;

    /**
     * Constructor por defecto
     */
    public CumuloMissingParameterException() {
        super();
    }

    /**
     * Constructor con mensaje de error
     * @param message Mensaje descriptivo del error
     */
    public CumuloMissingParameterException(String message) {
        super(message);
    }

    /**
     * Constructor con información específica del parámetro
```

```
     * @param message Mensaje descriptivo del error
     * @param parameterName Nombre del parámetro faltante
     * @param contextMethod Método donde ocurrió el error
     */
    public CumuloMissingParameterException(String message, String parameterName, String contextMethod) {
        super(message);
        this.parameterName = parameterName;
        this.contextMethod = contextMethod;
    }

    /**
     * Constructor para múltiples parámetros faltantes
     * @param message Mensaje descriptivo del error
     * @param missingParameters Lista de parámetros faltantes
     * @param contextMethod Método donde ocurrió el error
     */
    public CumuloMissingParameterException(
        String message,
        List<String> missingParameters,
        String contextMethod
    ) {
        super(message);
        this.missingParameters = missingParameters;
        this.contextMethod = contextMethod;
    }

    /**
     * Constructor con excepción causa
     * @param message Mensaje descriptivo del error
     * @param cause Excepción que causó esta excepción
     */
    public CumuloMissingParameterException(String message, Exception cause) {
        super(message, cause);
    }

    /**
     * Obtiene el nombre del parámetro faltante
     * @return Nombre del parámetro
     */
    public String getParameterName() {
        return this.parameterName;
    }

    /**
     * Obtiene el tipo del parámetro faltante
     * @return Tipo del parámetro
     */
    public String getParameterType() {
        return this.parameterType;
    }

    /**
     * Obtiene el método donde ocurrió el error
     * @return Nombre del método
     */
    public String getContextMethod() {
        return this.contextMethod;
```

```
    }

    /**
     * Obtiene la lista de parámetros faltantes
     * @return Lista de parámetros faltantes
     */
    public List<String> getMissingParameters() {
        return this.missingParameters != null ? this.missingParameters : new List<String>();
    }

    /**
     * Genera un mensaje completo con toda la información disponible
     * @return Mensaje completo
     */
    public String getDetailedMessage() {
        String detailedMessage = getMessage();

        if (String.isNotBlank(contextMethod)) {
            detailedMessage += '\nMétodo: ' + contextMethod;
        }

        if (String.isNotBlank(parameterName)) {
            detailedMessage += '\nParámetro faltante: ' + parameterName;
        }

        if (missingParameters != null && !missingParameters.isEmpty()) {
            detailedMessage += '\nParámetros faltantes: ' + String.join(missingParameters, ', ');
        }

        return detailedMessage;
    }
}
```

Mejores Prácticas para su Uso

1. **Mensajes Específicos**: Proporcionar mensajes de error claros que indiquen exactamente qué parámetro falta y en qué contexto.
2. **Validación Temprana**: Utilizar esta excepción al inicio de los métodos para validar parámetros de entrada.
3. **Agrupación de Validaciones**: Cuando sea posible, validar todos los parámetros requeridos de una vez y reportar todos los faltantes.
4. **Documentación de Parámetros**: Asegurar que la documentación de los métodos indique claramente qué parámetros son requeridos.
5. **Manejo Diferenciado**: Capturar y manejar esta excepción específicamente para proporcionar mensajes de error más útiles a los usuarios.

Consideraciones de Diseño

Ventajas de la Implementación Actual

1. **Simplicidad**: La implementación minimalista es fácil de entender y utilizar.
2. **Enfoque Específico**: Se centra exclusivamente en representar errores de parámetros faltantes.
3. **Respeto a la Seguridad**: El modificador `with sharing` asegura que se respeten las reglas de compartición.

Posibles Mejoras

1. **Información Detallada**: Podría extenderse para incluir información más específica sobre los parámetros faltantes.
2. **Métodos de Utilidad**: Podrían añadirse métodos estáticos para facilitar la validación común de parámetros.
3. **Integración con Logging**: Podría incluir funcionalidad para registrar automáticamente los errores de parámetros faltantes.

Integración con Pruebas Unitarias

```
@IsTest
private class CumuloMissingParameterExceptionTest {

    @IsTest
    static void testBasicException() {
        String expectedMessage = 'Parámetro requerido faltante: testParam';

        try {
            throw new CumuloMissingParameterException(expectedMessage);
        } catch (CumuloMissingParameterException e) {
            System.assertEquals(expectedMessage, e.getMessage());
            System.assertEquals('CumuloMissingParameterException', e.getTypeName());
        }
    }

    @IsTest
    static void testValidationMethod() {
        Map<String, Object> incompleteConfig = new Map<String, Object>{
            'endpoint' => 'https://test.com'
            // Falta apiKey
        };

        try {
            CumuloConfigurationManager.initializeFramework(incompleteConfig);
            System.assert(false, 'Debería haber lanzado CumuloMissingParameterException');
        } catch (CumuloMissingParameterException e) {
            System.assert(e.getMessage().contains('apiKey'));
        }
    }
}
```

Notas Adicionales

1. El nombre "Cumulo" sugiere que forma parte de un framework específico dentro de la organización Suratech.
2. El modificador `with sharing` indica que respeta las reglas de compartición de Salesforce, lo cual es apropiado para validaciones que pueden involucrar datos sensibles.
3. Esta excepción complementa otras excepciones del sistema como `ProductHandlerImplementationException`, proporcionando un ecosistema completo de manejo de errores específicos.
4. La referencia al proyecto "UH-17589" indica que forma parte de un desarrollo específico y estructurado dentro del framework Foundation Salesforce.

5. Es una práctica excelente tener excepciones específicas para diferentes tipos de errores de validación, mejorando tanto la experiencia del desarrollador como la capacidad de depuración del sistema.

LocalEmisionNotificationHandlerTest

Descripción General

`LocalEmisionNotificationHandlerTest` es una clase de prueba de Apex diseñada para validar el funcionamiento de la clase `LocalEmisionNotificationHandler`, que gestiona notificaciones de emisión local para productos.

Autor y Proyecto

- **Autor**: felipe.correa@nespon.com
- **Proyecto**: Foundation Salesforce - Suratech - UH-17589
- **Última modificación**: 31 de enero de 2025

Estructura de la Clase

Método @testSetup

```
@testSetup
static void makeData() {
  TestUtils.insertTestUsers();
}
```

Este método configura los datos de prueba necesarios, específicamente insertando usuarios de prueba mediante la clase utilitaria `TestUtils`.

Métodos de Prueba

1. testLocalEmision_Success

```
@isTest
static void testLocalEmision_Success()
```

Valida el flujo exitoso de la notificación de emisión local:

- Configura un mock HTTP que devuelve código 200 (OK)
- Ejecuta la prueba en el contexto de un usuario de prueba
- Crea un mapa de entrada con el producto 'viajes'
- Encola un trabajo asíncrono con `LocalEmisionNotificationHandler`
- Verifica que el estado del trabajo sea 'Completed'

2. testLocalEmision_Exception_Failure

```
@isTest
static void testLocalEmision_Exception_Failure()
```

Prueba el manejo de errores cuando el servicio HTTP falla:

- Configura un mock HTTP que devuelve código 400 con un error
- Ejecuta la prueba en el contexto de un usuario de prueba
- Intenta encolar un trabajo con `LocalEmisionNotificationHandler`

- Verifica que se lance una excepción con el mensaje 'Error en el servicio'

3. testLocalEmision_NoProduct_Failure

```
@isTest
static void testLocalEmision_NoProduct_Failure()
```

Prueba el escenario de fallo cuando no se proporciona un producto:

- Ejecuta la prueba en el contexto de un usuario de prueba
- Crea un mapa de entrada vacío (sin producto)
- Intenta encolar un trabajo con `LocalEmisionNotificationHandler`
- Verifica que se lance una excepción con el mensaje 'No se encontro el producto'

Dependencias

- `TestUtils`: Clase utilitaria para crear datos de prueba
- `LocalEmisionNotificationHandler`: Clase principal que se está probando
- `LocalEmisionNotificationHandlerMock`: Mock para simular respuestas HTTP

Notas Técnicas

- La clase utiliza el framework de pruebas de Apex con anotaciones `@isTest`
- Implementa pruebas para escenarios positivos y negativos
- Utiliza mocks HTTP para simular llamadas a servicios externos
- Emplea trabajos asíncronos con `System.enqueueJob()`
- Verifica el estado de los trabajos asíncronos después de su ejecución

IFeatureFlags

Descripción General

`IFeatureFlags` es una interfaz que define el contrato para la evaluación de banderas de características (feature flags) en el sistema. Esta interfaz establece el método necesario para evaluar si una característica específica está habilitada o no, devolviendo un resultado detallado de la evaluación.

Autor

Fabian Lopez \<fabian.lopez@nespon.com\>

Última Modificación

11 de abril de 2025

Definición de la Interfaz

```
public interface IFeatureFlags {
    FeatureFlags.FeatureEvaluationResult evaluate(String featureName);
}
```

Métodos

evaluate(String featureName)

Descripción

Este método evalúa si una característica específica está habilitada, basándose en el nombre de la característica proporcionado.

Parámetros

- `featureName` (String): Nombre de la característica a evaluar.

Retorno

- `FeatureFlags.FeatureEvaluationResult`: Objeto que contiene el resultado detallado de la evaluación de la característica, incluyendo si está habilitada y posiblemente información adicional sobre el motivo de la decisión.

Dependencias

- `FeatureFlags.FeatureEvaluationResult`: Clase anidada o interna que define la estructura del resultado de evaluación de una característica.

Propósito y Aplicaciones

El propósito principal de esta interfaz es proporcionar una abstracción para el mecanismo de evaluación de banderas de características, permitiendo:

1. **Decisiones Dinámicas**: Tomar decisiones en tiempo de ejecución sobre qué características están disponibles.
2. **Flexibilidad de Implementación**: Permitir diferentes estrategias de evaluación que pueden ser intercambiadas o modificadas sin afectar al código que las utiliza.
3. **Pruebas Simplificadas**: Facilitar la creación de implementaciones mock para pruebas unitarias.
4. **Desacoplamiento**: Separar la lógica de decisión sobre características de la lógica de negocio que depende de esas decisiones.

Esta interfaz es especialmente útil en escenarios como:

- **Despliegue Progresivo**: Habilitar nuevas características gradualmente para diferentes segmentos de usuarios.
- **Pruebas A/B**: Mostrar diferentes variantes de una característica a diferentes usuarios para medir su eficacia.
- **Switches de Emergencia**: Desactivar rápidamente características problemáticas sin necesidad de redeployer.
- **Personalización por Usuario/Perfil**: Adaptar la experiencia del usuario basándose en su perfil, permisos o preferencias.

Clase FeatureEvaluationResult Inferida

Aunque no se muestra en el código proporcionado, podemos inferir que existe una clase `FeatureEvaluationResult` dentro de una clase principal `FeatureFlags`. Esta clase probablemente tendría una estructura similar a la siguiente:

```
public class FeatureFlags {

    public class FeatureEvaluationResult {
        private final Boolean isEnabled;
        private final String reason;
        private final Map<String, Object> metadata;

        public FeatureEvaluationResult(Boolean isEnabled, String reason) {
            this.isEnabled = isEnabled;
            this.reason = reason;
            this.metadata = new Map<String, Object>();
        }

        public FeatureEvaluationResult(Boolean isEnabled, String reason, Map<String, Object> metadata) {
            this.isEnabled = isEnabled;
            this.reason = reason;
            this.metadata = metadata != null ? metadata : new Map<String, Object>();
        }

        public Boolean isEnabled() {
            return isEnabled;
        }

        public String getReason() {
            return reason;
        }

        public Map<String, Object> getMetadata() {
            return metadata;
        }

        public Object getMetadataValue(String key) {
            return metadata.containsKey(key) ? metadata.get(key) : null;
        }
    }

    // Otros métodos y clases de la clase principal FeatureFlags...
}
```

Ejemplos de Implementación

Implementación Básica

```
public class BasicFeatureFlags implements IFeatureFlags {

    private Map<String, Boolean> featuresMap;

    public BasicFeatureFlags() {
        // Inicializar con algunas características predeterminadas
        featuresMap = new Map<String, Boolean>{
            'NewUserInterface' => true,
            'AdvancedReporting' => false,
            'MultiLanguageSupport' => true
        };
    }

    public FeatureFlags.FeatureEvaluationResult evaluate(String featureName) {
        if (String.isBlank(featureName)) {
            return new FeatureFlags.FeatureEvaluationResult(false, 'Feature name is blank');
        }

        Boolean isEnabled = featuresMap.containsKey(featureName) ? featuresMap.get(featureName) : false;
        String reason = isEnabled ? 'Feature explicitly enabled' : 'Feature not enabled or not found';

        return new FeatureFlags.FeatureEvaluationResult(isEnabled, reason);
    }
}
```

Implementación Basada en Metadatos Personalizados

```
public class MetadataFeatureFlags implements IFeatureFlags {

    private IFeatureFlagProvider provider;

    public MetadataFeatureFlags() {
        this.provider = new DefaultFeatureFlagProvider();
    }

    public MetadataFeatureFlags(IFeatureFlagProvider provider) {
        this.provider = provider;
    }

    public FeatureFlags.FeatureEvaluationResult evaluate(String featureName) {
        if (String.isBlank(featureName)) {
            return new FeatureFlags.FeatureEvaluationResult(false, 'Feature name is blank');
        }

        Map<String, FeatureFlag__mdt> featureFlags = provider.getFeatureFlags();

        // Si la bandera no existe, la característica está deshabilitada
        if (!featureFlags.containsKey(featureName)) {
            return new FeatureFlags.FeatureEvaluationResult(
                false,
                'Feature not found in metadata'
            );
        }

        FeatureFlag__mdt flag = featureFlags.get(featureName);
```

```
        // Si la bandera no está activa, la característica está deshabilitada
        if (!flag.Is_Active__c) {
            return new FeatureFlags.FeatureEvaluationResult(
                false,
                'Feature is defined but not active'
            );
        }

        // Si la bandera requiere un permiso personalizado, verificar si el usuario actual lo tiene
        if (String.isNotBlank(flag.Custom_Permission_Name__c)) {
            Boolean hasPermission = FeatureManagement.checkPermission(flag.Custom_Permission_Name__c);

            if (!hasPermission) {
                return new FeatureFlags.FeatureEvaluationResult(
                    false,
                    'User lacks required permission: ' + flag.Custom_Permission_Name__c
                );
            }
        }

        // Crear un mapa con metadatos adicionales que podrían ser útiles
        Map<String, Object> metadata = new Map<String, Object>{
            'flagId' => flag.DeveloperName,
            'lastModified' => flag.LastModifiedDate,
            'description' => flag.Description__c
        };

        // Si no se requiere permiso personalizado y la bandera está activa, la característica está habilitada
        return new FeatureFlags.FeatureEvaluationResult(
            true,
            'Feature is active' + (String.isNotBlank(flag.Custom_Permission_Name__c) ?
                                   ' and user has required permission' : ''),
            metadata
        );
    }
}
```

Implementación Basada en Usuarios

```
public class UserBasedFeatureFlags implements IFeatureFlags {

    private Set<String> betaUserEmails;
    private Set<String> betaFeatures;

    public UserBasedFeatureFlags() {
        // Inicializar conjuntos de usuarios beta y características beta
        betaUserEmails = new Set<String>{
            'test.user@example.com',
            'beta.tester@example.com'
        };

        betaFeatures = new Set<String>{
            'NewSearchAlgorithm',
            'RedesignedProfile',
            'EnhancedDashboards'
        };
    }
```

```
    public FeatureFlags.FeatureEvaluationResult evaluate(String featureName) {
        if (String.isBlank(featureName)) {
            return new FeatureFlags.FeatureEvaluationResult(false, 'Feature name is blank');
        }

        // Si no es una característica beta, siempre está habilitada
        if (!betaFeatures.contains(featureName)) {
            return new FeatureFlags.FeatureEvaluationResult(true, 'Not a beta feature');
        }

        // Obtener el email del usuario actual
        String userEmail = UserInfo.getUserEmail();

        // Verificar si el usuario es un beta tester
        Boolean isBetaUser = betaUserEmails.contains(userEmail);

        // Crear metadatos con información adicional
        Map<String, Object> metadata = new Map<String, Object>{
            'isBetaFeature' => true,
            'userEmail' => userEmail
        };

        if (isBetaUser) {
            return new FeatureFlags.FeatureEvaluationResult(
                true,
                'User is a beta tester with access to this feature',
                metadata
            );
        } else {
            return new FeatureFlags.FeatureEvaluationResult(
                false,
                'User is not a beta tester',
                metadata
            );
        }
    }
}
```

Implementación Compuesta

```
public class CompositeFeatureFlags implements IFeatureFlags {

    private List<IFeatureFlags> evaluators;

    public CompositeFeatureFlags() {
        // Inicializar con evaluadores predeterminados
        evaluators = new List<IFeatureFlags>{
            new MetadataFeatureFlags(),
            new UserBasedFeatureFlags()
        };
    }

    public CompositeFeatureFlags(List<IFeatureFlags> evaluators) {
        this.evaluators = evaluators != null ? evaluators : new List<IFeatureFlags>();
    }
```

```
    public FeatureFlags.FeatureEvaluationResult evaluate(String featureName) {
        if (String.isBlank(featureName)) {
            return new FeatureFlags.FeatureEvaluationResult(false, 'Feature name is blank');
        }

        // Si cualquier evaluador habilita la característica, se considera habilitada
        for (IFeatureFlags evaluator : evaluators) {
            FeatureFlags.FeatureEvaluationResult result = evaluator.evaluate(featureName);
            if (result.isEnabled()) {
                return result;
            }
        }

        // Si ningún evaluador habilitó la característica, devolver el resultado del último evaluador
        // o un resultado predeterminado si no hay evaluadores
        if (!evaluators.isEmpty()) {
            return evaluators[evaluators.size() - 1].evaluate(featureName);
        } else {
            return new FeatureFlags.FeatureEvaluationResult(false, 'No feature flag evaluators available');
        }
    }

    public void addEvaluator(IFeatureFlags evaluator) {
        if (evaluator != null) {
            evaluators.add(evaluator);
        }
    }
}
```

Implementación Mock para Pruebas

```
@IsTest
public class MockFeatureFlags implements IFeatureFlags {

    private Map<String, Boolean> featuresMap;
    private Map<String, String> reasonsMap;

    public MockFeatureFlags() {
        featuresMap = new Map<String, Boolean>();
        reasonsMap = new Map<String, String>();
    }

    public FeatureFlags.FeatureEvaluationResult evaluate(String featureName) {
        Boolean isEnabled = featuresMap.containsKey(featureName) ? featuresMap.get(featureName) : false;
        String reason = reasonsMap.containsKey(featureName) ? reasonsMap.get(featureName) : 'Default mock
reason';

        return new FeatureFlags.FeatureEvaluationResult(isEnabled, reason);
    }

    // Métodos adicionales para configurar el mock

    public void setFeatureEnabled(String featureName, Boolean isEnabled) {
        featuresMap.put(featureName, isEnabled);
    }

    public void setFeatureReason(String featureName, String reason) {
```

```
        reasonsMap.put(featureName, reason);
    }

    public void reset() {
        featuresMap.clear();
        reasonsMap.clear();
    }
}
```

Ejemplo de Uso

```
public class ProductController {

    private final IFeatureFlags featureFlags;

    public ProductController() {
        // Usar la implementación predeterminada
        this.featureFlags = new MetadataFeatureFlags();
    }

    // Constructor con inyección de dependencias para pruebas
    @TestVisible
    private ProductController(IFeatureFlags featureFlags) {
        this.featureFlags = featureFlags;
    }

    public List<Product2> getProductsForDisplay() {
        List<Product2> products = [SELECT Id, Name, Description, Family, IsActive FROM Product2 WHERE IsActive
= true];

        // Verificar si la característica de filtrado avanzado está habilitada
        FeatureFlags.FeatureEvaluationResult advancedFilteringResult =
            featureFlags.evaluate('AdvancedProductFiltering');

        if (advancedFilteringResult.isEnabled()) {
            // Aplicar filtrado avanzado
            return applyAdvancedFiltering(products);
        } else {
            // Aplicar filtrado básico
            return applyBasicFiltering(products);
        }
    }

    private List<Product2> applyAdvancedFiltering(List<Product2> products) {
        // Implementación del filtrado avanzado
        // ...
        return products;
    }

    private List<Product2> applyBasicFiltering(List<Product2> products) {
        // Implementación del filtrado básico
        // ...
        return products;
    }
}
```

Ejemplo de Uso en Controlador Lightning

```
// productController.js
import { LightningElement, wire } from 'lwc';
import checkFeatureEnabled from '@salesforce/apex/FeatureFlagController.checkFeatureEnabled';
import getProducts from '@salesforce/apex/ProductController.getProducts';

export default class ProductList extends LightningElement {
    products;
    error;
    showAdvancedSearch = false;

    connectedCallback() {
        // Verificar si la característica de búsqueda avanzada está habilitada
        checkFeatureEnabled({ featureName: 'AdvancedProductSearch' })
            .then(result => {
                this.showAdvancedSearch = result.isEnabled;
                if (result.isEnabled) {
                    console.log('Advanced search enabled: ' + result.reason);
                }
            })
            .catch(error => {
                console.error('Error checking feature flag:', error);
            });
    }

    @wire(getProducts)
    wiredProducts({ error, data }) {
        if (data) {
            this.products = data;
            this.error = undefined;
        } else if (error) {
            this.error = error;
            this.products = undefined;
        }
    }

    // Resto del controlador...
}
```

```
// FeatureFlagController.cls
public with sharing class FeatureFlagController {

    private static IFeatureFlags featureFlags = new MetadataFeatureFlags();

    @AuraEnabled(cacheable=true)
    public static Map<String, Object> checkFeatureEnabled(String featureName) {
        FeatureFlags.FeatureEvaluationResult result = featureFlags.evaluate(featureName);

        // Convertir el resultado a un mapa que pueda ser serializado para JavaScript
        return new Map<String, Object>{
            'isEnabled' => result.isEnabled(),
            'reason' => result.getReason()
        };
```

```
    }
}
```

Ejemplo de Uso en Pruebas

```
@IsTest
private class ProductControllerTest {

    @IsTest
    static void testGetProductsWithAdvancedFilteringEnabled() {
        // Configurar datos de prueba
        createTestProducts();

        // Crear un mock de IFeatureFlags
        MockFeatureFlags mockFeatureFlags = new MockFeatureFlags();
        mockFeatureFlags.setFeatureEnabled('AdvancedProductFiltering', true);
        mockFeatureFlags.setFeatureReason('AdvancedProductFiltering', 'Enabled for test');

        // Crear una instancia del controlador con el mock
        ProductController controller = new ProductController(mockFeatureFlags);

        // Ejecutar el método a probar
        Test.startTest();
        List<Product2> products = controller.getProductsForDisplay();
        Test.stopTest();

        // Verificar que se aplicó el filtrado avanzado
        // (Aquí se agregarían las aserciones específicas según la implementación del filtrado)
        System.assertEquals(5, products.size(), 'Should return 5 products with advanced filtering');
    }

    @IsTest
    static void testGetProductsWithAdvancedFilteringDisabled() {
        // Configurar datos de prueba
        createTestProducts();

        // Crear un mock de IFeatureFlags
        MockFeatureFlags mockFeatureFlags = new MockFeatureFlags();
        mockFeatureFlags.setFeatureEnabled('AdvancedProductFiltering', false);
        mockFeatureFlags.setFeatureReason('AdvancedProductFiltering', 'Disabled for test');

        // Crear una instancia del controlador con el mock
        ProductController controller = new ProductController(mockFeatureFlags);

        // Ejecutar el método a probar
        Test.startTest();
        List<Product2> products = controller.getProductsForDisplay();
        Test.stopTest();

        // Verificar que se aplicó el filtrado básico
        // (Aquí se agregarían las aserciones específicas según la implementación del filtrado)
        System.assertEquals(3, products.size(), 'Should return 3 products with basic filtering');
    }

    private static void createTestProducts() {
        // Lógica para crear productos de prueba
```

```
        // ...
    }
}
```

Relación con IFeatureFlagProvider

La interfaz `IFeatureFlags` probablemente trabaja en conjunto con la interfaz `IFeatureFlagProvider` en un sistema de gestión de banderas de características más amplio:

- `IFeatureFlagProvider`: Se enfoca en obtener y proporcionar acceso a la configuración de las banderas de características.
- `IFeatureFlags`: Se enfoca en la evaluación lógica de si una característica está habilitada o no para un contexto particular.

Una implementación típica de `IFeatureFlags` (como la clase `MetadataFeatureFlags` en los ejemplos) podría utilizar un `IFeatureFlagProvider` para obtener la configuración de las banderas y luego aplicar la lógica de evaluación.

Mejores Prácticas

1. **Inyección de Dependencias**: Utilizar inyección de dependencias para proporcionar implementaciones específicas de `IFeatureFlags` a las clases que lo necesitan, facilitando las pruebas y la flexibilidad.
2. **Resultados Informativos**: Proporcionar razones claras y detalladas en los resultados de evaluación para facilitar la depuración y el entendimiento.
3. **Caché Inteligente**: Implementar mecanismos de caché para evitar evaluaciones repetitivas de las mismas características en una misma transacción.
4. **Auditoría**: Considerar registrar evaluaciones importantes para análisis y depuración.
5. **Comportamiento Predeterminado Seguro**: Si una característica no está definida, normalmente debería considerarse como deshabilitada.

Consideraciones de Diseño

Ventajas

1. **Flexibilidad**: Permite intercambiar diferentes estrategias de evaluación sin modificar el código cliente.
2. **Testabilidad**: Facilita la prueba de código que depende de banderas de características.
3. **Separación de Responsabilidades**: Separa la lógica de evaluación de características de la lógica de negocio.
4. **Extensibilidad**: Permite añadir nuevas estrategias de evaluación sin modificar el código existente.

Desafíos

1. **Complejidad**: Introduce una capa adicional de abstracción.
2. **Overhead de Performance**: Puede añadir cierto overhead de rendimiento, especialmente si la evaluación es compleja.
3. **Mantenimiento**: Requiere mantener la configuración de banderas actualizada y relevante.

Notas Adicionales

1. La interfaz `IFeatureFlags` es parte de un sistema más amplio de gestión de características que probablemente incluye otros componentes como proveedores de configuración, servicios de administración, etc.
2. La clase `FeatureEvaluationResult` proporciona más información que un simple booleano, permitiendo entender por qué una

característica está habilitada o deshabilitada.

3. Este enfoque a menudo se utiliza en conjunto con estrategias de despliegue continuo, permitiendo liberar código que contiene características incompletas o experimentales sin exponerlas a todos los usuarios.
4. Las implementaciones concretas pueden variar significativamente según los requisitos específicos del sistema, desde simples verificaciones de configuración hasta algoritmos complejos que consideran múltiples factores como perfiles de usuario, regiones geográficas, horarios, etc.

IFeatureFlagProvider

Descripción General

`IFeatureFlagProvider` es una interfaz que define el contrato para proveedores de banderas de características (feature flags) en el sistema. Esta interfaz establece los métodos necesarios para acceder tanto a los nombres de permisos personalizados como a las configuraciones de banderas de características almacenadas en metadatos personalizados.

Definición de la Interfaz

```
public interface IFeatureFlagProvider {

    Set<String> getCustomPermissionNames();
    Map<String,FeatureFlag__mdt> getFeatureFlags();

}
```

Métodos

getCustomPermissionNames()

Descripción

Este método debe devolver un conjunto de nombres de permisos personalizados disponibles en el sistema. Los permisos personalizados suelen utilizarse como uno de los mecanismos para controlar la activación de características específicas para diferentes usuarios o perfiles.

Parámetros

Ninguno.

Retorno

- `Set<String>`: Conjunto de nombres de permisos personalizados.

getFeatureFlags()

Descripción

Este método debe devolver un mapa de banderas de características definidas en el sistema. Las banderas de características están almacenadas en registros de metadatos personalizados (`FeatureFlag__mdt`) y permiten configurar características específicas que pueden ser activadas o desactivadas de forma dinámica.

Parámetros

Ninguno.

Retorno

- `Map<String, FeatureFlag__mdt>`: Mapa donde la clave es el nombre de la bandera de característica y el valor es el registro de metadatos personalizado correspondiente.

Dependencias

- `FeatureFlag__mdt`: Tipo de metadatos personalizado que contiene la configuración de las banderas de características.

Propósito y Aplicaciones

El propósito principal de esta interfaz es proporcionar una abstracción para el acceso a la configuración de banderas de características. Esto permite:

1. **Gestión Centralizada**: Centralizar la lógica de acceso a banderas de características.
2. **Flexibilidad de Implementación**: Permitir diferentes implementaciones según el entorno o las necesidades.
3. **Facilidad de Pruebas**: Facilitar la creación de implementaciones mock para pruebas unitarias.
4. **Desacoplamiento**: Desacoplar la gestión de características del código que las utiliza.

Esta interfaz es particularmente útil en escenarios como:

- **Despliegue Progresivo**: Liberar características a grupos selectos de usuarios antes de un lanzamiento general.
- **Test A/B**: Probar diferentes variantes de una característica con diferentes grupos de usuarios.
- **Configuración por Entorno**: Activar o desactivar características específicas según el entorno (desarrollo, prueba, producción).
- **Interruptores de Emergencia**: Proporcionar la capacidad de desactivar rápidamente características problemáticas sin necesidad de desplegar nuevo código.

Ejemplos de Implementación

Implementación Básica

```
public class DefaultFeatureFlagProvider implements IFeatureFlagProvider {

    public Set<String> getCustomPermissionNames() {
        // Obtener todos los permisos personalizados disponibles en el sistema
        Set<String> permissionNames = new Set<String>();

        for (CustomPermission cp : [SELECT Id, DeveloperName FROM CustomPermission]) {
            permissionNames.add(cp.DeveloperName);
        }

        return permissionNames;
    }

    public Map<String, FeatureFlag__mdt> getFeatureFlags() {
        // Obtener todas las banderas de características desde los metadatos personalizados
        Map<String, FeatureFlag__mdt> featureFlags = new Map<String, FeatureFlag__mdt>();

        for (FeatureFlag__mdt flag : [SELECT DeveloperName, MasterLabel, Is_Active__c,
                                      Description__c, Custom_Permission_Name__c,
                                      Min_Api_Version__c, Max_Api_Version__c
                                      FROM FeatureFlag__mdt]) {
            featureFlags.put(flag.DeveloperName, flag);
        }

        return featureFlags;
    }
}
```

Implementación con Caché

```
public class CachedFeatureFlagProvider implements IFeatureFlagProvider {

    private static Set<String> cachedPermissionNames;
    private static Map<String, FeatureFlag__mdt> cachedFeatureFlags;
    private static Datetime lastCacheRefresh;
    private static final Integer CACHE_EXPIRY_MINS = 15;
```

```
    public Set<String> getCustomPermissionNames() {
        if (shouldRefreshCache()) {
            refreshCache();
        }
        return cachedPermissionNames;
    }

    public Map<String, FeatureFlag__mdt> getFeatureFlags() {
        if (shouldRefreshCache()) {
            refreshCache();
        }
        return cachedFeatureFlags;
    }

    private Boolean shouldRefreshCache() {
        if (cachedPermissionNames == null || cachedFeatureFlags == null || lastCacheRefresh == null) {
            return true;
        }

        Datetime now = Datetime.now();
        Long millisSinceRefresh = now.getTime() - lastCacheRefresh.getTime();
        Long millisToExpiry = CACHE_EXPIRY_MINS * 60 * 1000;

        return millisSinceRefresh > millisToExpiry;
    }

    private void refreshCache() {
        // Obtener permisos personalizados
        cachedPermissionNames = new Set<String>();
        for (CustomPermission cp : [SELECT DeveloperName FROM CustomPermission]) {
            cachedPermissionNames.add(cp.DeveloperName);
        }

        // Obtener banderas de características
        cachedFeatureFlags = new Map<String, FeatureFlag__mdt>();
        for (FeatureFlag__mdt flag : [SELECT DeveloperName, MasterLabel, Is_Active__c,
                                      Description__c, Custom_Permission_Name__c,
                                      Min_Api_Version__c, Max_Api_Version__c
                                      FROM FeatureFlag__mdt]) {
            cachedFeatureFlags.put(flag.DeveloperName, flag);
        }

        // Actualizar timestamp de refresco
        lastCacheRefresh = Datetime.now();
    }
}
```

Mock para Pruebas

```
@IsTest
public class MockFeatureFlagProvider implements IFeatureFlagProvider {

    private Set<String> mockPermissions;
    private Map<String, FeatureFlag__mdt> mockFeatureFlags;

    public MockFeatureFlagProvider() {
```

```
        // Inicializar con valores predeterminados vacíos
        mockPermissions = new Set<String>();
        mockFeatureFlags = new Map<String, FeatureFlag__mdt>();
    }

    public Set<String> getCustomPermissionNames() {
        return mockPermissions;
    }

    public Map<String, FeatureFlag__mdt> getFeatureFlags() {
        return mockFeatureFlags;
    }

    // Métodos adicionales para configurar el mock para pruebas

    public void addCustomPermission(String permissionName) {
        mockPermissions.add(permissionName);
    }

    public void setCustomPermissions(Set<String> permissions) {
        mockPermissions = permissions;
    }

    public void addFeatureFlag(String flagName, Boolean isActive, String permissionName) {
        FeatureFlag__mdt flag = createMockFeatureFlag(flagName, isActive, permissionName);
        mockFeatureFlags.put(flagName, flag);
    }

    public void setFeatureFlags(Map<String, FeatureFlag__mdt> featureFlags) {
        mockFeatureFlags = featureFlags;
    }

    private FeatureFlag__mdt createMockFeatureFlag(String name, Boolean isActive, String permissionName) {
        // Crear una instancia de FeatureFlag__mdt utilizando técnicas de prueba
        // Nota: Como los registros de metadatos personalizados no se pueden crear en tiempo de ejecución,
        // se necesitaría utilizar técnicas como la reflexión o una clase wrapper

        // Ejemplo simplificado (no funcionará directamente, es ilustrativo)
        FeatureFlag__mdt mockFlag = new FeatureFlag__mdt();
        mockFlag.DeveloperName = name;
        mockFlag.MasterLabel = name;
        mockFlag.Is_Active__c = isActive;
        mockFlag.Custom_Permission_Name__c = permissionName;

        return mockFlag;
    }
}
```

Clase de Servicio de Feature Flags

Esta interfaz normalmente sería utilizada por una clase de servicio que proporciona la lógica para verificar si una característica está habilitada:

```
public class FeatureFlagService {

    private static FeatureFlagService instance;
    private IFeatureFlagProvider provider;
```

```
    // Constructor privado para patrón Singleton
    private FeatureFlagService() {
        this.provider = new DefaultFeatureFlagProvider();
    }

    // Método para obtener la instancia singleton
    public static FeatureFlagService getInstance() {
        if (instance == null) {
            instance = new FeatureFlagService();
        }
        return instance;
    }

    // Método para inyectar un proveedor personalizado (útil para pruebas)
    public void setProvider(IFeatureFlagProvider customProvider) {
        this.provider = customProvider;
    }

    // Verificar si una característica está habilitada
    public Boolean isFeatureEnabled(String featureName) {
        Map<String, FeatureFlag__mdt> featureFlags = provider.getFeatureFlags();

        // Si la bandera no existe, la característica está deshabilitada
        if (!featureFlags.containsKey(featureName)) {
            return false;
        }

        FeatureFlag__mdt flag = featureFlags.get(featureName);

        // Si la bandera no está activa, la característica está deshabilitada
        if (!flag.Is_Active__c) {
            return false;
        }

        // Si la bandera requiere un permiso personalizado, verificar si el usuario actual lo tiene
        if (String.isNotBlank(flag.Custom_Permission_Name__c)) {
            return FeatureManagement.checkPermission(flag.Custom_Permission_Name__c);
        }

        // Si no se requiere permiso personalizado y la bandera está activa, la característica está habilitada
        return true;
    }

    // Verificar si una característica está habilitada y cumple con restricciones de versión API
    public Boolean isFeatureEnabledForApiVersion(String featureName, Decimal apiVersion) {
        Map<String, FeatureFlag__mdt> featureFlags = provider.getFeatureFlags();

        // Si la bandera no existe, la característica está deshabilitada
        if (!featureFlags.containsKey(featureName)) {
            return false;
        }

        FeatureFlag__mdt flag = featureFlags.get(featureName);

        // Verificar si la bandera está habilitada
        if (!isFeatureEnabled(featureName)) {
```

```
            return false;
        }

        // Verificar restricciones de versión API
        if (flag.Min_Api_Version__c != null && apiVersion < flag.Min_Api_Version__c) {
            return false;
        }

        if (flag.Max_Api_Version__c != null && apiVersion > flag.Max_Api_Version__c) {
            return false;
        }

        return true;
    }
}
```

Ejemplo de Uso

```
// Obtener instancia del servicio
FeatureFlagService featureFlagService = FeatureFlagService.getInstance();

// Verificar si una característica está habilitada
if (featureFlagService.isFeatureEnabled('NewUserInterface')) {
    // Usar la nueva interfaz de usuario
    showNewUI();
} else {
    // Usar la interfaz de usuario clásica
    showClassicUI();
}

// Ejemplo con versión API
Decimal currentApiVersion = 54.0;
if (featureFlagService.isFeatureEnabledForApiVersion('AdvancedSearch', currentApiVersion)) {
    // Usar funcionalidad de búsqueda avanzada
    performAdvancedSearch();
} else {
    // Usar búsqueda estándar
    performStandardSearch();
}
```

Ejemplo de uso en Pruebas

```
@IsTest
private class FeatureFlagServiceTest {

    @IsTest
    static void testFeatureEnabled() {
        // Crear un mock del proveedor
        MockFeatureFlagProvider mockProvider = new MockFeatureFlagProvider();

        // Configurar el mock para la prueba
        mockProvider.addFeatureFlag('TestFeature', true, null);

        // Inyectar el mock en el servicio
        FeatureFlagService service = FeatureFlagService.getInstance();
```

```
        service.setProvider(mockProvider);

        // Verificar que la característica está habilitada
        System.assertEquals(true, service.isFeatureEnabled('TestFeature'));
    }

    @IsTest
    static void testFeatureDisabled() {
        // Crear un mock del proveedor
        MockFeatureFlagProvider mockProvider = new MockFeatureFlagProvider();

        // Configurar el mock para la prueba
        mockProvider.addFeatureFlag('TestFeature', false, null);

        // Inyectar el mock en el servicio
        FeatureFlagService service = FeatureFlagService.getInstance();
        service.setProvider(mockProvider);

        // Verificar que la característica está deshabilitada
        System.assertEquals(false, service.isFeatureEnabled('TestFeature'));
    }

    // Otros casos de prueba...
}
```

Estructura del Objeto Personalizado FeatureFlag__mdt

El tipo de metadatos personalizado `FeatureFlag__mdt` probablemente tendría la siguiente estructura:

Campo	Tipo	Descripción
DeveloperName	String	Nombre único de la bandera de característica
MasterLabel	String	Etiqueta visible para la bandera
Is_Active__c	Boolean	Indica si la característica está activa
Description__c	Text	Descripción de la característica
Custom_Permission_Name__c	String	Nombre del permiso personalizado requerido (si aplica)
Min_Api_Version__c	Number	Versión mínima de API compatible (si aplica)
Max_Api_Version__c	Number	Versión máxima de API compatible (si aplica)

Mejores Prácticas

1. **Caché Eficiente**: Implementar estrategias de caché para evitar consultas repetitivas a los metadatos.
2. **Manejo de Errores**: Incluir manejo de errores robusto en las implementaciones para evitar que fallos en la configuración de banderas afecten la funcionalidad principal.
3. **Despliegue Controlado**: Utilizar las banderas de características para implementar un despliegue controlado de nuevas funcionalidades.

4. **Documentación**: Mantener documentación clara sobre cada bandera de característica y su propósito.
5. **Limpieza**: Implementar un proceso regular para revisar y eliminar banderas de características obsoletas.

Consideraciones de Rendimiento

1. **Consultas Optimizadas**: Minimizar el número de consultas SOQL cargando todos los datos necesarios de una vez.
2. **Uso de Caché**: Implementar mecanismos de caché para reducir las consultas a la base de datos.
3. **Tamaño de los Datos**: Ser consciente del número de banderas de características y permisos personalizados, ya que un número grande podría afectar el rendimiento.

Consideraciones de Seguridad

1. **Control de Acceso**: Asegurar que los permisos personalizados estén configurados correctamente para controlar el acceso a las características.
2. **Validación de Entradas**: Validar los nombres de características antes de utilizarlos en las consultas.
3. **Exposición de Características**: Tener cuidado con la exposición de información sobre características en desarrollo a usuarios no autorizados.

Notas Adicionales

1. Esta interfaz es parte de un sistema más amplio de gestión de características que probablemente incluye otros componentes como servicios de administración, páginas de configuración, etc.
2. La implementación concreta podría necesitar ajustes dependiendo de los requisitos específicos y la arquitectura del sistema.
3. En sistemas con un gran número de banderas de características, podría ser beneficioso implementar categorías o grupos para una mejor organización.
4. La interfaz podría extenderse para incluir métodos adicionales, como la capacidad de filtrar banderas por categoría o buscar banderas que coincidan con ciertos criterios.

FeatureFlags

Descripción General

Clase principal para la evaluación y gestión de feature flags en el sistema. Implementa la interfaz `IFeatureFlags` y proporciona un mecanismo robusto para activar/desactivar funcionalidades basado en Custom Permissions y Custom Metadata Types.

Información de la Clase

- **Autor:** felipe.correa@nespon.com
- **Proyecto:** Foundation Salesforce - Suratech - UH-17589
- **Última modificación:** 04-11-2025
- **Modificado por:** fabian.lopez@nespon.com
- **Tipo:** Clase pública con sharing
- **Implementa:** IFeatureFlags

Propiedades Privadas

features

```
private Map<String, FeatureFlag__mdt> features
```

Mapa que contiene todos los feature flags definidos como Custom Metadata Types.

customPermissionNames

```
private Set<String> customPermissionNames
```

Conjunto de nombres de Custom Permissions disponibles en la organización.

mockValues

```
private static Map<String, Boolean> mockValues
```

Mapa estático para almacenar valores mock durante la ejecución de tests.

Constructores

Constructor con Provider

```
public FeatureFlags(IFeatureFlagProvider provider)
```

Descripción: Constructor que permite inyección de dependencias para el proveedor de feature flags.

Parámetros:

- `provider` (IFeatureFlagProvider): Implementación del proveedor de feature flags

Comportamiento:

- Inicializa `features` usando `provider.getFeatureFlags()`
- Inicializa `customPermissionNames` usando `provider.getCustomPermissionNames()`

Constructor por Defecto

```
public FeatureFlags()
```

Descripción: Constructor por defecto que utiliza `FeatureFlagProvider` como implementación estándar.

Comportamiento:

- Llama al constructor principal con una nueva instancia de `FeatureFlagProvider`

Métodos Principales

evaluate

```
public FeatureEvaluationResult evaluate(String featureName)
```

Descripción: Método principal para evaluar el estado de un feature flag específico.

Parámetros:

- `featureName` (String): Nombre del feature flag a evaluar

Retorna:

- `FeatureEvaluationResult`: Objeto que contiene el resultado, nombre del feature y razón de la evaluación

Lógica de Evaluación:

1. **Contexto de Test:**
   - Si `Test.isRunningTest()` es true, configura valores mock para features específicos
   - `sura_foundation_platform_events` = true (para tests de PlatformEventIntegrationObserver)
   - `sura_foundation_event_logs` = true (para tests de EventLogManager)
   - Retorna valor mock si existe
2. **Evaluación por Custom Permission:**
   - Si el feature name existe en `customPermissionNames`
   - Usa `FeatureManagement.checkPermission()` para verificar el permiso
   - Retorna `HAS_CUSTOM_PERMISSION` o `MISSING_CUSTOM_PERMISSION`
3. **Evaluación por Custom Metadata Type:**
   - Si el feature existe en el mapa `features`
   - Verifica el campo `Is_Active__c`
   - Retorna `CUSTOM_METADATA_TYPE_ENABLED` o `CUSTOM_METADATA_TYPE_DISABLED`
4. **Feature No Encontrado:**
   - Si el feature no existe en ninguna fuente
   - Retorna `FLAG_NOT_FOUND`

lwcEvaluate (Método Estático para LWC)

```
@AuraEnabled(cacheable=true)
public static Boolean lwcEvaluate(String featureName)
```

Descripción: Método estático optimizado para componentes Lightning Web Components.

Parámetros:

- `featureName` (String): Nombre del feature flag a evaluar

Retorna:

- `Boolean`: Estado del feature flag (true/false)

Características:

- Anotado con `@AuraEnabled(cacheable=true)` para optimización de cache
- Crea una nueva instancia de `FeatureFlags` para cada evaluación
- Simplifica el resultado a un boolean

setMockValue (Método de Test)

```
@TestVisible
private static void setMockValue(String featureName, Boolean value)
```

Descripción: Método utilitario para configurar valores mock durante tests.

Parámetros:

- `featureName` (String): Nombre del feature flag
- `value` (Boolean): Valor mock a asignar

Visibilidad: Solo visible durante tests (`@TestVisible`)

Clases Internas

FeatureEvaluationResult

```
public class FeatureEvaluationResult
```

Descripción: Clase interna que encapsula el resultado de una evaluación de feature flag.

Propiedades:

- `result` (Boolean): Estado del feature flag
- `featureName` (String): Nombre del feature evaluado
- `reason` (FeatureReason): Razón de la evaluación

Métodos:

- `isEnabled()`: Retorna el estado boolean del feature
- `getFeatureName()`: Retorna el nombre del feature
- `getReason()`: Retorna la razón de la evaluación

Constructor:

```
public FeatureEvaluationResult(Boolean result, String featureName, FeatureReason reason)
```

FeatureReason (Enum)

```
public enum FeatureReason
```

Valores:

- `HAS_CUSTOM_PERMISSION`: Feature habilitado por Custom Permission
- `MISSING_CUSTOM_PERMISSION`: Feature deshabilitado por falta de Custom Permission
- `CUSTOM_METADATA_TYPE_ENABLED`: Feature habilitado por Custom Metadata Type
- `CUSTOM_METADATA_TYPE_DISABLED`: Feature deshabilitado por Custom Metadata Type
- `FLAG_NOT_FOUND`: Feature flag no encontrado en el sistema
- `MOCK_VALUE`: Valor configurado para testing

Estrategias de Evaluación

Jerarquía de Evaluación

1. **Mock Values** (solo en tests)
2. **Custom Permissions** (basado en usuario/perfil)
3. **Custom Metadata Types** (configuración global)
4. **Default** (feature no encontrado = false)

Ventajas del Diseño

- **Flexibilidad:** Soporta múltiples fuentes de configuración
- **Granularidad:** Control a nivel de usuario (permissions) y global (metadata)
- **Testabilidad:** Sistema de mocking integrado
- **Performance:** Cache habilitado para LWC
- **Trazabilidad:** Razones detalladas de evaluación

Integración con Lightning Web Components

Uso en LWC

```
import { wire } from 'lwc';
import lwcEvaluate from '@salesforce/apex/FeatureFlags.lwcEvaluate';

export default class MyComponent extends LightningElement {
    @wire(lwcEvaluate, { featureName: 'my_feature' })
    featureResult;

    get isFeatureEnabled() {
        return this.featureResult.data === true;
```

```
    }
}
```

Casos de Uso

Desarrollo

- Control de features en desarrollo vs producción
- Activación gradual de funcionalidades (rollout progresivo)
- A/B testing basado en perfiles de usuario

Administración

- Habilitar/deshabilitar features sin deployments
- Control granular por usuario o perfil
- Configuración centralizada de comportamientos

Testing

- Simulación de diferentes estados de features
- Tests independientes del estado real de configuración
- Validación de comportamiento con features activadas/desactivadas

Dependencias

- **IFeatureFlags:** Interfaz implementada
- **IFeatureFlagProvider:** Para obtención de datos
- **FeatureFlagProvider:** Implementación por defecto del proveedor
- **FeatureFlag__mdt:** Custom Metadata Type para configuración
- **FeatureManagement:** API de Salesforce para Custom Permissions

Consideraciones de Performance

- Custom Metadata Types están cacheados automáticamente
- Custom Permissions se evalúan en tiempo real
- Método LWC con cache habilitado para mejor performance en UI
- Mock values solo se procesan durante tests

Mejoras Potenciales

- Implementar cache local para Custom Permissions
- Agregar logging de evaluaciones para auditoría
- Soporte para feature flags con valores complejos (no solo boolean)

- Integración con herramientas de monitoreo para uso de features
- Implementar feature flags con fecha de expiración

FeatureFlagProvider

Descripción General

Clase que implementa la interfaz `IFeatureFlagProvider` para proporcionar funcionalidad de gestión de feature flags y permisos personalizados en Salesforce. Permite obtener información sobre permisos personalizados y feature flags configurados en el sistema.

Información de la Clase

- **Tipo:** Clase pública con sharing
- **Implementa:** IFeatureFlagProvider
- **Propósito:** Proveedor de feature flags y permisos personalizados

Métodos

getCustomPermissionNames

```
public Set<String> getCustomPermissionNames()
```

Descripción: Obtiene todos los nombres de desarrollador de los permisos personalizados disponibles en la organización.

Parámetros: Ninguno

Retorna:

- `Set<String>`: Conjunto de nombres de desarrollador de permisos personalizados

Comportamiento:

1. Crea un conjunto vacío para almacenar los nombres de permisos
2. Ejecuta una consulta SOQL para obtener todos los permisos personalizados
3. Itera sobre los resultados y agrega el `DeveloperName` de cada permiso al conjunto
4. Retorna el conjunto completo de nombres

Consulta SOQL utilizada:

```
SELECT Id, DeveloperName FROM CustomPermission
```

getFeatureFlags

```
public Map<String,FeatureFlag__mdt> getFeatureFlags()
```

Descripción: Obtiene todos los feature flags configurados como Custom Metadata Types.

Parámetros: Ninguno

Retorna:

- `Map<String,FeatureFlag__mdt>`: Mapa con todos los feature flags donde la clave es el nombre del registro y el valor es el objeto metadata

Comportamiento:

- Utiliza el método estático `getAll()` de Custom Metadata Types para obtener todos los registros de `FeatureFlag__mdt`
- Retorna directamente el mapa completo sin filtrado

Funcionalidad Principal

Gestión de Permisos Personalizados

- Proporciona acceso programático a todos los permisos personalizados de la organización
- Facilita la validación de permisos en tiempo de ejecución
- Permite implementar lógica condicional basada en permisos

Gestión de Feature Flags

- Acceso centralizado a los feature flags configurados
- Utiliza Custom Metadata Types para configuración sin código
- Permite activar/desactivar funcionalidades dinámicamente

Consideraciones de Diseño

Sharing Settings

- Declarada como `with sharing` para respetar las reglas de seguridad del usuario actual
- Asegura que las consultas respeten los permisos de acceso a datos

Performance

- `getCustomPermissionNames()`: Ejecuta consulta SOQL, considerar caché si se llama frecuentemente
- `getFeatureFlags()`: Utiliza Custom Metadata Types que están cacheados automáticamente

Dependencias

- Requiere Custom Metadata Type `FeatureFlag__mdt`
- Depende de la existencia de Custom Permissions en la organización
- Implementa interfaz `IFeatureFlagProvider`

Casos de Uso

1. **Validación de Permisos:** Verificar si un usuario tiene permisos específicos antes de ejecutar operaciones
2. **Feature Toggling:** Activar/desactivar funcionalidades basado en feature flags
3. **Configuración Dinámica:** Cambiar comportamiento de la aplicación sin despliegues de código
4. **Control de Acceso:** Implementar lógica de autorización granular

Mejoras Potenciales

- Implementar caché para `getCustomPermissionNames()` si se requiere alto rendimiento

- Agregar métodos para verificar permisos específicos de usuarios
- Incluir logging para auditoría de acceso a feature flags

AccountConfigurator

Descripción General

Interfaz global que define el contrato para configuradores de objetos `Account`. Implementa el patrón Strategy permitiendo aplicar diferentes configuraciones a instancias de Account de manera flexible y extensible. Esta interfaz trabaja en conjunto con `AccountBuilder` para proporcionar un sistema de configuración modular.

Información de la Interfaz

- **Autor:** Jean Carlos Melendez
- **Última modificación:** 01-30-2025
- **Modificado por:** Jean Carlos Melendez
- **Tipo:** Interfaz global
- **Propósito:** Contrato para configuradores de Account en patrón Strategy

Método de la Interfaz

configure

```
void configure(Account account)
```

Descripción: Método que debe ser implementado por todas las clases configuradoras para aplicar configuraciones específicas a un objeto Account.

Parámetros:

- `account` (Account): Instancia del objeto Account a configurar

Retorna: `void` - Modifica el Account por referencia

Responsabilidades de la Implementación:

- Aplicar configuraciones específicas al Account recibido
- Modificar campos, establecer valores por defecto, o aplicar lógica de negocio
- Manejar excepciones de configuración si es necesario

Patrón de Diseño

Strategy Pattern

Esta interfaz implementa el **patrón Strategy**, que permite:

- **Intercambiabilidad:** Diferentes estrategias de configuración sin cambiar el código cliente
- **Extensibilidad:** Agregar nuevas configuraciones sin modificar código existente
- **Separación de responsabilidades:** Cada configurador maneja una preocupación específica
- **Reutilización:** Configuradores pueden ser compartidos entre diferentes builders

Integración con Builder Pattern

Trabaja en conjunto con `AccountBuilder` para crear un sistema flexible:

```
// El Builder utiliza la interfaz
global AccountBuilder addConfigurators(List<AccountConfigurator> configuratorsList)

// Durante build(), aplica todos los configuradores
for (AccountConfigurator configurator : configurators) {
    configurator.configure(account);
}
```

Implementaciones Recomendadas

Configurador de Industria

```
global class IndustryAccountConfigurator implements AccountConfigurator {
    private String industry;

    global IndustryAccountConfigurator(String industry) {
        this.industry = industry;
    }

    global void configure(Account account) {
        account.Industry = this.industry;

        // Configuración específica por industria
        switch on industry {
            when 'Technology' {
                account.Type = 'Customer - Direct';
                account.Rating = 'Hot';
            }
            when 'Healthcare' {
                account.Type = 'Customer - Channel';
                account.Rating = 'Warm';
            }
        }
    }
}
```

Configurador de Dirección Completa

```
global class AddressAccountConfigurator implements AccountConfigurator {
    private String street, city, state, country, postalCode;

    global AddressAccountConfigurator(String street, String city, String state, String country, String
postalCode) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.country = country;
        this.postalCode = postalCode;
    }

    global void configure(Account account) {
        // Billing Address
        account.BillingStreet = this.street;
        account.BillingCity = this.city;
```

```
        account.BillingState = this.state;
        account.BillingCountry = this.country;
        account.BillingPostalCode = this.postalCode;

        // Shipping Address (mismo que billing por defecto)
        account.ShippingStreet = this.street;
        account.ShippingCity = this.city;
        account.ShippingState = this.state;
        account.ShippingCountry = this.country;
        account.ShippingPostalCode = this.postalCode;
    }
}
```

Configurador de Campos Personalizados

```
global class CustomFieldAccountConfigurator implements AccountConfigurator {
    private Map<String, Object> customFields;

    global CustomFieldAccountConfigurator(Map<String, Object> customFields) {
        this.customFields = customFields;
    }

    global void configure(Account account) {
        for (String fieldName : customFields.keySet()) {
            try {
                account.put(fieldName, customFields.get(fieldName));
            } catch (Exception e) {
                System.debug('Error setting custom field ' + fieldName + ': ' + e.getMessage());
            }
        }
    }
}
```

Configurador de Integración Externa

```
global class ExternalSystemAccountConfigurator implements AccountConfigurator {
    private String externalSystemId;
    private String systemName;

    global ExternalSystemAccountConfigurator(String externalSystemId, String systemName) {
        this.externalSystemId = externalSystemId;
        this.systemName = systemName;
    }

    global void configure(Account account) {
        // Configurar campos de integración
        account.put('External_System_ID__c', this.externalSystemId);
        account.put('Source_System__c', this.systemName);
        account.put('Integration_Status__c', 'Pending');
        account.put('Last_Sync_Date__c', System.now());
    }
}
```

Configurador Condicional

```
global class ConditionalAccountConfigurator implements AccountConfigurator {
    private Map<String, Object> conditions;
    private List<AccountConfigurator> configurators;

    global ConditionalAccountConfigurator(Map<String, Object> conditions, List<AccountConfigurator>
configurators) {
        this.conditions = conditions;
        this.configurators = configurators;
    }

    global void configure(Account account) {
        if (shouldApplyConfiguration(account)) {
            for (AccountConfigurator configurator : configurators) {
                configurator.configure(account);
            }
        }
    }

    private Boolean shouldApplyConfiguration(Account account) {
        // Lógica para evaluar condiciones
        for (String field : conditions.keySet()) {
            if (account.get(field) != conditions.get(field)) {
                return false;
            }
        }
        return true;
    }
}
```

Casos de Uso Completos

Configuración Empresarial Completa

```
List<AccountConfigurator> enterpriseConfigurators = new List<AccountConfigurator>{
    new IndustryAccountConfigurator('Technology'),
    new AddressAccountConfigurator('123 Tech St', 'San Francisco', 'CA', 'USA', '94105'),
    new CustomFieldAccountConfigurator(new Map<String, Object>{
        'Annual_Revenue__c' => 1000000,
        'Employee_Count__c' => 50,
        'Support_Level__c' => 'Premium'
    }),
    new ExternalSystemAccountConfigurator('EXT-12345', 'CRM_Legacy')
};

Account enterpriseAccount = new AccountBuilder()
    .setName('TechCorp Solutions')
    .setPhone('555-123-4567')
    .addConfigurators(enterpriseConfigurators)
    .buildAndInsert();
```

Configuración con Validación

```
global class ValidatedAccountConfigurator implements AccountConfigurator {
    private List<AccountConfigurator> configurators;

    global void configure(Account account) {
```

```
        // Pre-validación
        validateAccount(account);

        // Aplicar configuradores
        for (AccountConfigurator configurator : configurators) {
            configurator.configure(account);
        }

        // Post-validación
        validateConfiguredAccount(account);
    }

    private void validateAccount(Account account) {
        if (String.isBlank(account.Name)) {
            throw new ConfigurationException('Account name is required before configuration');
        }
    }

    private void validateConfiguredAccount(Account account) {
        // Validaciones post-configuración
    }
}
```

Ventajas del Diseño

Flexibilidad

- **Composición dinámica:** Diferentes combinaciones de configuradores para diferentes tipos de Account
- **Configuración condicional:** Aplicar configuraciones basadas en condiciones específicas
- **Reutilización:** Configuradores pueden usarse en múltiples contextos

Mantenibilidad

- **Separación de responsabilidades:** Cada configurador maneja una preocupación específica
- **Fácil testing:** Configuradores pueden probarse independientemente
- **Código modular:** Cambios en configuraciones no afectan otras partes del sistema

Extensibilidad

- **Nuevas configuraciones:** Agregar configuradores sin modificar código existente
- **Configuraciones complejas:** Combinar múltiples configuradores simples
- **Integración con sistemas externos:** Configuradores específicos para integraciones

Testing Strategies

Test de Configurador Individual

```
@isTest
private class IndustryAccountConfiguratorTest {
    @isTest
    static void testTechnologyIndustryConfiguration() {
        Account testAccount = new Account(Name = 'Test Account');
```

```
        AccountConfigurator configurator = new IndustryAccountConfigurator('Technology');

        configurator.configure(testAccount);

        System.assertEquals('Technology', testAccount.Industry);
        System.assertEquals('Customer - Direct', testAccount.Type);
        System.assertEquals('Hot', testAccount.Rating);
    }
}
```

Test de Múltiples Configuradores

```
@isTest
static void testMultipleConfigurators() {
    Account testAccount = new Account(Name = 'Test Account');
    List<AccountConfigurator> configurators = new List<AccountConfigurator>{
        new IndustryAccountConfigurator('Healthcare'),
        new AddressAccountConfigurator('456 Health Ave', 'Boston', 'MA', 'USA', '02101')
    };

    for (AccountConfigurator configurator : configurators) {
        configurator.configure(testAccount);
    }

    System.assertEquals('Healthcare', testAccount.Industry);
    System.assertEquals('456 Health Ave', testAccount.BillingStreet);
    System.assertEquals('Boston', testAccount.BillingCity);
}
```

Consideraciones de Implementación

Performance

- Configuradores deben ser eficientes, especialmente en operaciones bulk
- Evitar consultas SOQL dentro de configuradores cuando sea posible
- Considerar cache para configuraciones complejas

Error Handling

- Configuradores deben manejar errores gracefully
- Logging apropiado para debugging
- Validación de datos de entrada

Security

- Validar permisos de campo antes de configurar
- Sanitizar datos de entrada
- Respetar field-level security

Mejores Prácticas

Naming Convention

- Sufijo `AccountConfigurator` para todas las implementaciones
- Nombres descriptivos que indiquen el propósito: `IndustryAccountConfigurator`

Constructor Pattern

- Configuradores deben recibir datos necesarios en el constructor
- Evitar configuradores con estado mutable

Validation

- Validar parámetros en constructores
- Manejar casos edge apropiadamente
- Documentar precondiciones y postcondiciones

Logging

- Log configuraciones aplicadas para auditoría
- Debug level apropiado para diferentes entornos

Dependencias

- **Account:** Objeto estándar de Salesforce a configurar
- **AccountBuilder:** Clase que utiliza esta interfaz
- **System.debug:** Para logging (recomendado en implementaciones)

Esta interfaz es fundamental para crear un sistema flexible y extensible de configuración de Accounts, permitiendo que el código sea mantenible y fácil de extender con nuevas funcionalidades.

AccountBuilder

Descripción General

Clase global que implementa el patrón Builder para la construcción flexible y configuración de objetos `Account`. Proporciona una interfaz fluida para crear cuentas con diferentes configuraciones, soporte para configuradores personalizados, y métodos de conveniencia para conversión de Leads a Accounts.

Información de la Clase

- **Autor:** Jean Carlos Melendez
- **Última modificación:** 01-30-2025
- **Modificado por:** Jean Carlos Melendez
- **Tipo:** Clase global
- **Propósito:** Constructor flexible de objetos Account con patrón Builder

Propiedades Globales

account

```
global Account account
```

Instancia del objeto `Account` que se está construyendo.

configurators

```
global List<AccountConfigurator> configurators
```

Lista de configuradores personalizados que se aplicarán durante la construcción del Account.

Constructor

AccountBuilder()

```
global AccountBuilder()
```

Descripción: Constructor por defecto que inicializa un nuevo objeto Account vacío.

Comportamiento:

- Crea una nueva instancia de `Account`
- Inicializa la lista de configuradores vacía

Métodos Builder (Fluent Interface)

setId

```
global AccountBuilder setId(String Id)
```

Descripción: Establece el ID del Account (típicamente para updates).

Parámetros:

- `Id` (String): ID del Account

Retorna: `AccountBuilder` para encadenamiento de métodos

setName

```
global AccountBuilder setName(String name)
```

Descripción: Establece el nombre del Account.

Parámetros:

- `name` (String): Nombre del Account

Retorna: `AccountBuilder` para encadenamiento de métodos

setPhone

```
global AccountBuilder setPhone(String phone)
```

Descripción: Establece el teléfono del Account.

Parámetros:

- `phone` (String): Número de teléfono

Retorna: `AccountBuilder` para encadenamiento de métodos

setBillingAddress

```
global AccountBuilder setBillingAddress(String city, String street)
```

Descripción: Establece la dirección de facturación del Account.

Parámetros:

- `city` (String): Ciudad de facturación
- `street` (String): Calle de facturación

Retorna: `AccountBuilder` para encadenamiento de métodos

Nota: Solo configura ciudad y calle; otros campos de dirección pueden necesitar métodos adicionales.

addConfigurators

```
global AccountBuilder addConfigurators(List<AccountConfigurator> configuratorsList)
```

Descripción: Agrega una lista de configuradores personalizados que se aplicarán durante la construcción.

Parámetros:

- `configuratorsList` (List
  ): Lista de configuradores a agregar

Retorna: `AccountBuilder` para encadenamiento de métodos

Comportamiento:

- Verifica que la lista no esté vacía antes de agregar
- Utiliza `addAll()` para agregar todos los configuradores de una vez

Métodos de Construcción

build

```
global Account build()
```

Descripción: Construye el objeto Account aplicando todos los configuradores sin persistirlo en la base de datos.

Retorna: `Account` configurado pero no insertado

Proceso:

1. Verifica si hay configuradores en la lista
2. Itera sobre cada configurador y llama a `configure(account)`
3. Retorna el Account configurado

buildAndInsert

```
global Account buildAndInsert()
```

Descripción: Construye el objeto Account, aplica configuradores, y lo inserta en la base de datos.

Retorna: `Account` configurado e insertado con ID asignado

Proceso:

1. Aplica todos los configuradores (mismo proceso que `build()`)
2. Ejecuta `insert account`
3. Retorna el Account con ID asignado

Métodos de Servicio

AccountService (con Record Type)

```
global Account AccountService(Lead foundLead, String recordTypeAPIName)
```

Descripción: Método de servicio para crear un Account basado en un Lead con un Record Type específico.

Parámetros:

- `foundLead` (Lead): Lead fuente para extraer datos
- `recordTypeAPIName` (String): API Name del Record Type a asignar

Retorna: `Account` creado e insertado

Proceso:

1. Consulta el Record Type por `DeveloperName`
2. Crea un nuevo AccountBuilder
3. Configura nombre (concatenando LastName + FirstName del Lead)
4. Configura teléfono desde `MobilePhone` del Lead
5. Ejecuta `buildAndInsert()`

Nota: El código tiene una consulta SOQL para Record Type pero no se asigna al Account.

AccountService (Record Type por Defecto)

```
global Account AccountService(Lead foundLead)
```

Descripción: Sobrecarga del método anterior que usa 'PersonAccount' como Record Type por defecto.

Parámetros:

- `foundLead` (Lead): Lead fuente para extraer datos

Retorna: `Account` creado e insertado

Comportamiento: Llama a `AccountService(foundLead, 'PersonAccount')`

Patrón de Diseño Implementado

Builder Pattern

Ventajas:

- **Fluent Interface:** Métodos encadenables para construcción intuitiva
- **Flexibilidad:** Configuración opcional de diferentes campos
- **Extensibilidad:** Soporte para configuradores personalizados
- **Reutilización:** Mismo builder puede crear múltiples variaciones

Estructura:

```
Account acc = new AccountBuilder()
    .setName("Test Company")
    .setPhone("555-1234")
    .setBillingAddress("Miami", "123 Main St")
    .addConfigurators(customConfigurators)
    .buildAndInsert();
```

Patrón Strategy (con Configuradores)

AccountConfigurator Interface

Los configuradores implementan el patrón Strategy para aplicar configuraciones específicas:

```
public interface AccountConfigurator {
    void configure(Account account);
}
```

Ejemplos de Configuradores:

- Configurador de campos personalizados
- Configurador de integración externa
- Configurador específico por industria

Casos de Uso

Construcción Básica

```
Account basicAccount = new AccountBuilder()
    .setName("ACME Corp")
    .setPhone("555-0123")
    .build();
```

Conversión de Lead a Account

```
Lead myLead = [SELECT LastName, FirstName, MobilePhone FROM Lead LIMIT 1];
Account convertedAccount = new AccountBuilder().AccountService(myLead);
```

Configuración Avanzada con Configuradores

```
List<AccountConfigurator> configs = new List<AccountConfigurator>{
    new IndustryConfigurator('Technology'),
    new CustomFieldConfigurator()
};

Account complexAccount = new AccountBuilder()
    .setName("Tech Solutions Inc")
    .addConfigurators(configs)
    .buildAndInsert();
```

Account con Record Type Específico

```
Account personAccount = new AccountBuilder()
    .AccountService(leadRecord, 'PersonAccount');
```

Ventajas del Diseño

Flexibilidad

- Configuración incremental de campos
- Soporte para extensiones a través de configuradores
- Métodos de conveniencia para casos comunes

Mantenibilidad

- Separación clara entre construcción y configuración
- Código reutilizable para diferentes tipos de Account
- Fácil testing a través de builders

Usabilidad

- Interfaz fluida y legible
- Métodos de servicio para casos específicos
- Soporte tanto para build como insert

Consideraciones y Mejoras

Problemas Identificados

1. **Record Type no asignado:** El método `AccountService` consulta Record Type pero no lo asigna
2. **Validación limitada:** Falta validación de campos requeridos
3. **Manejo de errores:** No hay manejo de excepciones en inserts

Mejoras Sugeridas

Corrección de Record Type

```
global Account AccountService(Lead foundLead, String recordTypeAPIName) {
    Id recordTypeId = [SELECT Id FROM RecordType
                       WHERE DeveloperName = :recordTypeAPIName LIMIT 1].Id;

    return new AccountBuilder()
        .setId(null) // Nuevo método para Record Type
        .setRecordTypeId(recordTypeId) // Método faltante
        .setName(foundLead?.LastName + ' ' + foundLead?.FirstName)
        .setPhone(foundLead?.MobilePhone)
        .buildAndInsert();
}
```

Métodos Adicionales Recomendados

```
global AccountBuilder setRecordTypeId(Id recordTypeId)
global AccountBuilder setIndustry(String industry)
global AccountBuilder setWebsite(String website)
global AccountBuilder setShippingAddress(String city, String street)
```

Validación

```
global Account build() {
    validateRequiredFields();
    applyConfigurators();
    return account;
}

private void validateRequiredFields() {
    if (String.isBlank(account.Name)) {
        throw new BuilderException('Account Name is required');
    }
}
```

Dependencias

- **Account:** Objeto estándar de Salesforce
- **Lead:** Para métodos de conversión
- **RecordType:** Para configuración de tipos de cuenta
- **AccountConfigurator:** Interfaz para configuradores personalizados (debe ser implementada)

Testing Considerations

- Crear builders de test con datos mock
- Probar encadenamiento de métodos

- Validar configuradores personalizados
- Testing de métodos de servicio con Leads reales

QuotingException

Descripción General

`QuotingException` es una clase de excepción personalizada que extiende la clase `Exception` estándar de Apex. Esta clase está diseñada para representar errores específicos que ocurren durante los procesos de cotización en el sistema.

Autor

Soulberto Lorenzo \<soulberto@cloudblue.us\>

Última Modificación

16 de agosto de 2024

Estructura

```
public class QuotingException extends Exception {

}
```

La clase no define métodos o propiedades adicionales más allá de los heredados de la clase base `Exception`.

Herencia

- **Clase Base**: `Exception`
- **Paquete**: Estándar de Salesforce

Propósito

El propósito principal de esta clase es proporcionar un tipo de excepción específico para errores relacionados con el proceso de cotización, permitiendo:

1. **Categorización de Errores**: Distinguir errores de cotización de otros tipos de excepciones en el sistema.
2. **Manejo Específico**: Permitir que el código que captura excepciones pueda identificar y manejar específicamente errores del proceso de cotización.
3. **Claridad en el Código**: Mejorar la legibilidad del código al indicar explícitamente cuando un error está relacionado con el proceso de cotización.
4. **Seguimiento Específico**: Facilitar el seguimiento y análisis de errores específicos del proceso de cotización.

Métodos Heredados

Al extender la clase `Exception` estándar, `QuotingException` hereda los siguientes métodos y propiedades:

Constructores

- **`QuotingException()`**: Constructor por defecto.
- **`QuotingException(String message)`**: Constructor que acepta un mensaje de error.
- **`QuotingException(Exception cause)`**: Constructor que acepta otra excepción como causa.
- **`QuotingException(String message, Exception cause)`**: Constructor que acepta un mensaje y otra excepción como causa.

Métodos

- `getMessage()`: Devuelve el mensaje de error asociado con la excepción.
- `getCause()`: Devuelve la excepción que causó esta excepción, si existe.
- `getTypeName()`: Devuelve el nombre del tipo de excepción.
- `setMessage(String message)`: Establece el mensaje de error para la excepción.
- `getLineNumber()`: Devuelve el número de línea donde se generó la excepción.
- `getStackTraceString()`: Devuelve una representación de texto del seguimiento de la pila.

Uso en el Sistema

Esta excepción está diseñada para ser utilizada por clases que participan en el proceso de cotización, tales como:

1. `QuoteProductHandler`: Puede lanzar esta excepción cuando hay problemas específicos en el proceso de cotización.
2. `QuoteUtils`: Puede lanzar esta excepción cuando hay problemas en la validación o manipulación de cotizaciones.
3. **Clases de validación de datos**: Pueden lanzar esta excepción cuando los datos de entrada para una cotización no cumplen con los requisitos.
4. **Procedimientos de integración**: Pueden lanzar esta excepción cuando hay problemas al interactuar con sistemas externos durante el proceso de cotización.

Ejemplo de Lanzamiento

```
public Map<String, Object> calculatePremium(Map<String, Object> quoteData) {
    try {
        // Validar datos de entrada
        if (!quoteData.containsKey('insuredItems') || quoteData.get('insuredItems') == null) {
            throw new QuotingException('Los elementos asegurados son requeridos para calcular la prima');
        }

        List<Object> insuredItems = (List<Object>)quoteData.get('insuredItems');
        if (insuredItems.isEmpty()) {
            throw new QuotingException('Debe proporcionar al menos un elemento asegurado');
        }

        // Calcular la prima basada en los elementos asegurados
        Decimal totalPremium = 0;
        for (Object item : insuredItems) {
            Map<String, Object> insuredItem = (Map<String, Object>)item;

            if (!insuredItem.containsKey('value') || insuredItem.get('value') == null) {
                throw new QuotingException('Cada elemento asegurado debe tener un valor especificado');
            }

            Decimal itemValue = (Decimal)insuredItem.get('value');
            Decimal itemPremium = calculateItemPremium(itemValue, insuredItem);
            totalPremium += itemPremium;
        }

        // Crear respuesta
        return new Map<String, Object>{
            'totalPremium' => totalPremium,
```

```
            'currency' => 'USD',
            'calculationDate' => System.now()
        };
    } catch (QuotingException qe) {
        // Relanzar la excepción específica de cotización
        throw qe;
    } catch (Exception e) {
        // Convertir excepciones genéricas en QuotingException
        throw new QuotingException('Error al calcular la prima: ' + e.getMessage(), e);
    }
}

private Decimal calculateItemPremium(Decimal itemValue, Map<String, Object> itemData) {
    // Lógica de cálculo de prima para un elemento específico
    // ...

    // Si hay algún problema específico del cálculo
    if (itemValue <= 0) {
        throw new QuotingException('El valor del elemento asegurado debe ser mayor que cero');
    }

    return itemValue * 0.05; // Ejemplo simplificado
}
```

Ejemplo de Captura

```
try {
    // Obtener datos de cotización
    Map<String, Object> quoteData = getQuoteDataFromRequest();

    // Calcular la prima
    Map<String, Object> premiumResult = calculatePremium(quoteData);

    // Procesar el resultado
    processQuoteResult(premiumResult);
} catch (QuotingException qe) {
    // Manejo específico para errores de cotización
    System.debug('Error en el proceso de cotización: ' + qe.getMessage());

    // Registrar el error específico de cotización
    Quoting_Error__c errorRecord = new Quoting_Error__c(
        Error_Message__c = qe.getMessage(),
        Stack_Trace__c = qe.getStackTraceString(),
        Timestamp__c = Datetime.now(),
        Quote_Data__c = JSON.serialize(quoteData)
    );
    insert errorRecord;

    // Devolver una respuesta de error específica para cotizaciones
    return createQuotingErrorResponse(qe.getMessage());
} catch (Exception e) {
    // Manejo genérico para otros tipos de excepciones
    System.debug('Error general: ' + e.getMessage());

    // Devolver una respuesta de error genérica
```

```
    return createGenericErrorResponse(e.getMessage());
}
```

Extensión Personalizada

Aunque la implementación actual es minimalista, la clase puede ser extendida para incluir información adicional específica del proceso de cotización:

```
public class EnhancedQuotingException extends QuotingException {
    private String errorCode;
    private String quoteId;
    private String productCode;
    private Decimal attemptedPremium;

    public EnhancedQuotingException(String message) {
        super(message);
    }

    public EnhancedQuotingException(String message, String errorCode) {
        super(message);
        this.errorCode = errorCode;
    }

    public EnhancedQuotingException(
        String message,
        String errorCode,
        String quoteId,
        String productCode
    ) {
        super(message);
        this.errorCode = errorCode;
        this.quoteId = quoteId;
        this.productCode = productCode;
    }

    public EnhancedQuotingException(
        String message,
        String errorCode,
        String quoteId,
        String productCode,
        Decimal attemptedPremium,
        Exception cause
    ) {
        super(message, cause);
        this.errorCode = errorCode;
        this.quoteId = quoteId;
        this.productCode = productCode;
        this.attemptedPremium = attemptedPremium;
    }

    public String getErrorCode() {
        return this.errorCode;
    }

    public String getQuoteId() {
        return this.quoteId;
    }
```

```
    public String getProductCode() {
        return this.productCode;
    }

    public Decimal getAttemptedPremium() {
        return this.attemptedPremium;
    }
}
```

Mejores Prácticas

1. **Mensajes Descriptivos**: Incluir información específica en los mensajes de error, como campos faltantes, valores inválidos o condiciones no cumplidas.
2. **Captura Selectiva**: Capturar esta excepción específicamente cuando se quiera manejar errores de cotización de manera diferenciada.
3. **Información de Contexto**: Al lanzar la excepción, proporcionar información de contexto que pueda ayudar a diagnosticar el problema, como los datos de entrada que causaron el error.
4. **Encadenamiento de Excepciones**: Utilizar el constructor que acepta una excepción causa para preservar el seguimiento de la pila original cuando se convierten excepciones genéricas en `QuotingException`.
5. **Registro Adecuado**: Registrar las excepciones capturadas para su posterior análisis y solución de problemas, posiblemente en un objeto personalizado para errores de cotización.

Puntos a Considerar

1. **Granularidad**: Considerar si es necesario tener subtipos más específicos de `QuotingException` para diferentes aspectos del proceso de cotización (por ejemplo, `QuoteValidationException`, `PremiumCalculationException`, etc.).
2. **Códigos de Error**: Implementar un sistema de códigos de error para categorizar diferentes tipos de errores de cotización de manera más estructurada.
3. **Información Contextual**: Evaluar la necesidad de incluir información contextual adicional en la excepción, como se muestra en el ejemplo de extensión personalizada.
4. **Integración con Sistemas de Monitoreo**: Asegurar que estas excepciones se integren adecuadamente con los sistemas de monitoreo y alerta existentes.

Integración con Otras Clases

Relación con QuoteProductHandler

La clase `QuoteProductHandler` podría utilizar `QuotingException` para manejar errores específicos del proceso de cotización:

```
// Ejemplo de uso en QuoteProductHandler
public override Map<String, Object> process(
  Map<String, Object> options,
  Map<String, Object> input,
  Map<String, Object> output
) {
  try {
    // Validar datos de entrada
    Map<Boolean, Object> validation = QuoteUtils.validateInputMapForQuoting(input);
    if (validation.containsKey(false)) {
      throw new QuotingException(validation.get(false).toString());
```

```
    }

    // Procesar la cotización
    // ...

    return output;
  } catch (QuotingException qe) {
    // Manejar o relanzar la excepción específica de cotización
    throw qe;
  } catch (Exception e) {
    // Convertir otras excepciones en QuotingException
    throw new QuotingException('Error en el proceso de cotización: ' + e.getMessage(), e);
  }
}
```

Relación con QuoteUtils

La clase `QuoteUtils` podría lanzar `QuotingException` en sus métodos de validación:

```
// Ejemplo de cómo QuoteUtils podría lanzar QuotingException
public static void validateQuoteData(Map<String, Object> quoteData) {
  if (!quoteData.containsKey(MAIN_NODE) || quoteData.get(MAIN_NODE) == null) {
    throw new QuotingException('Falta el nodo principal requerido: ' + MAIN_NODE);
  }

  Map<String, Object> mainNode = (Map<String, Object>)quoteData.get(MAIN_NODE);

  for (String requiredParam : REQUIRED_PARAMETERS) {
    if (!mainNode.containsKey(requiredParam) || mainNode.get(requiredParam) == null) {
      throw new QuotingException('Falta el parámetro requerido: ' + requiredParam);
    }
  }
}
```

Notas Adicionales

1. Esta clase forma parte del ecosistema de cotización de la organización y se utiliza en conjunto con otras clases como `QuoteProductHandler` y `QuoteUtils`.
2. La simplicidad de la implementación sugiere que la clase se utiliza principalmente para categorización de errores más que para transportar información adicional específica.
3. En un entorno de producción, considerar la implementación de mecanismos de registro y notificación automatizados para errores de cotización.
4. Esta excepción podría ser parte de una jerarquía más amplia de excepciones específicas del dominio, posiblemente con una superclase común como `DomainException` o similar.

APIs y Endpoints - SURA Foundation Framework

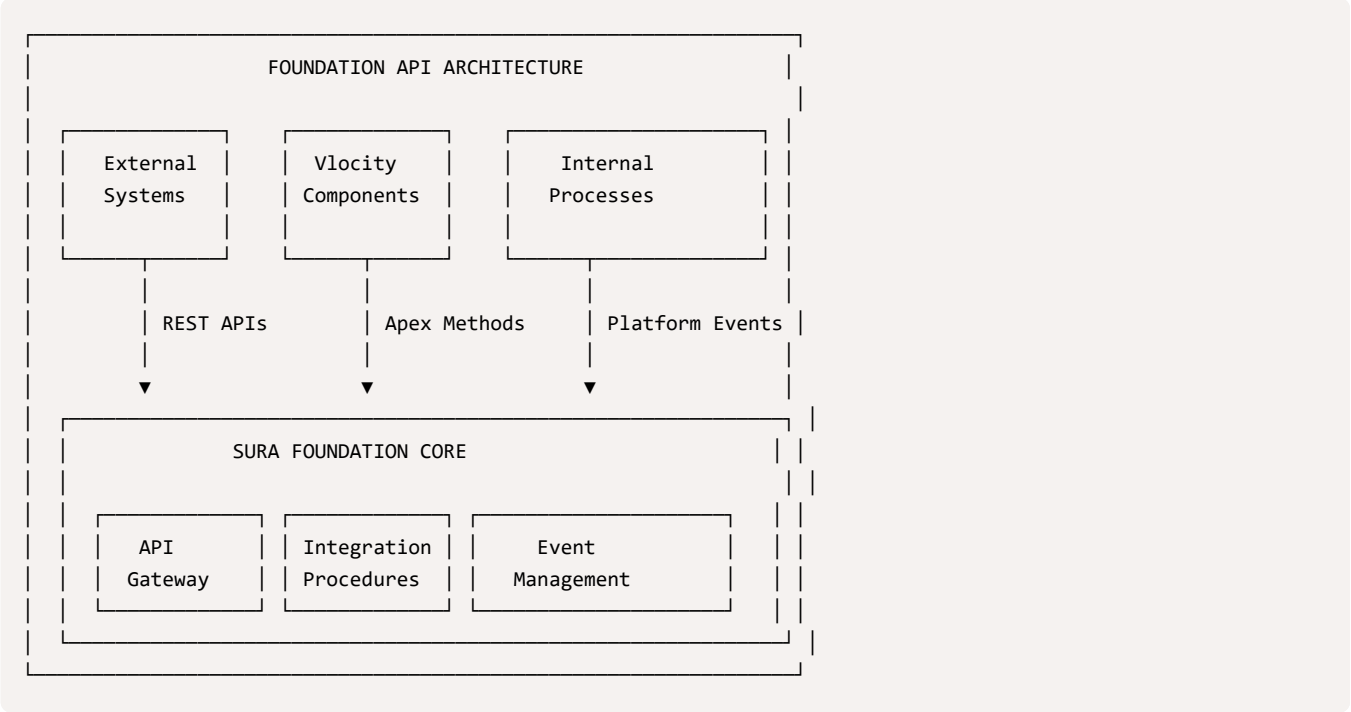
Resumen Ejecutivo

Este documento describe las APIs, endpoints y métodos de integración disponibles en **SURA Foundation Framework**. El framework expone servicios REST, métodos Apex invocables, y Integration Procedures para facilitar la integración con sistemas externos y componentes de Vlocity/OmniStudio.

🏗️ Arquitectura de APIs

Principios de Diseño

1. **RESTful**: Seguimiento de estándares REST para APIs HTTP
2. **Versionado**: Control de versiones para compatibilidad
3. **Seguridad**: Autenticación y autorización robusta
4. **Observabilidad**: Logging completo de requests/responses
5. **Resiliencia**: Manejo de errores y reintentos configurables



🌐 REST APIs

Base URL y Versionado

Base URL: `/services/apexrest/sura/foundation/v1/`

Estructura de Endpoints:

```
https://{instance}.salesforce.com/services/apexrest/sura/foundation/v1/{resource}
```

Autenticación

Métodos Soportados:

- OAuth 2.0 (Recomendado)
- Session ID
- Connected Apps

Headers Requeridos:

```
Authorization: Bearer {access_token}
Content-Type: application/json
Accept: application/json
X-SURA-API-Version: v1
```

📋 Endpoints Principales

1. Process Management APIs

POST /process/execute

Propósito: Ejecutar un proceso específico del framework

Request:

```
POST /services/apexrest/sura/foundation/v1/process/execute
Content-Type: application/json

{
  "processName": "LeadKnowing",
  "productFamily": "Seguros de Motos",
  "stage": "Conocimiento",
  "recordId": "00Q000000123456",
  "parameters": {
    "vehicleType": "Moto",
    "engine": "150CC",
    "model": "2024",
    "validateRUNT": true
  },
  "executeAsync": false
}
```

Response:

```
{
  "success": true,
  "processExecutionId": "a0X000000123456",
  "status": "Completed",
  "duration": 1250,
  "result": {
    "processedRecords": 1,
    "validationsPassed": true,
    "integrationsCalled": ["RUNT", "FASECOLDA"],
    "nextStage": "Tarificación"
  },
  "errors": [],
  "warnings": [
    {
      "code": "WARN_001",
      "message": "Vehículo no encontrado en FASECOLDA, usando valores por defecto"
    }
```

```
  ]
}
```

GET /process/status/

Propósito: Consultar el estado de una ejecución

Response:

```
{
  "executionId": "a0X000000123456",
  "status": "Running",
  "progress": 75,
  "startTime": "2025-05-23T10:30:00Z",
  "estimatedCompletion": "2025-05-23T10:32:30Z",
  "currentPhase": "In",
  "processedSteps": [
    {
      "stepName": "Validation",
      "status": "Completed",
      "duration": 150
    },
    {
      "stepName": "Integration",
      "status": "Running",
      "duration": 0
    }
  ]
}
```

2. Integration APIs

POST /integration/external/call

Propósito: Realizar llamadas a servicios externos con retry logic

Request:

```
POST /services/apexrest/sura/foundation/v1/integration/external/call
Content-Type: application/json

{
  "serviceName": "RUNT_VehicleConsultation",
  "endpoint": "/api/vehicles/lookup",
  "method": "POST",
  "headers": {
    "X-API-Key": "encrypted_key",
    "Content-Type": "application/json"
  },
  "payload": {
    "plate": "ABC123",
    "documentType": "CC",
    "documentNumber": "12345678"
  },
  "retryConfig": {
    "maxRetries": 3,
    "retryPattern": "Fibonacci",
    "retryOnCodes": [500, 502, 503, 504]
  }
}
```

Response:

```
{
  "integrationId": "IL-000123",
  "success": true,
  "httpStatus": 200,
  "duration": 850,
  "retryAttempts": 0,
  "response": {
    "vehicleFound": true,
    "vehicleData": {
      "brand": "YAMAHA",
      "model": "YBR 125",
      "year": 2023,
      "engine": "125CC",
      "fasecolda": "123456"
    }
  }
}
```

3. Event Management APIs

POST /events/publish

Propósito: Publicar eventos en el sistema

Request:

```
{
  "eventType": "ProcessCompleted",
  "category": "Business",
  "recordId": "00Q000000123456",
  "payload": {
    "processName": "LeadKnowing",
    "stage": "Conocimiento",
    "result": "Success",
    "nextActions": ["StartRating"]
  },
  "publishAsync": true
}
```

GET /events/history/

Propósito: Obtener historial de eventos de un registro

Response:

```
{
  "recordId": "00Q000000123456",
  "totalEvents": 15,
  "events": [
    {
      "eventId": "EL-000456",
      "eventType": "ProcessStarted",
      "timestamp": "2025-05-23T10:30:00Z",
      "processName": "LeadKnowing",
      "status": "Success",
      "duration": 150
    },
    {
```

```
      "eventId": "EL-000457",
      "eventType": "IntegrationCalled",
      "timestamp": "2025-05-23T10:30:05Z",
      "serviceName": "RUNT",
      "status": "Success",
      "duration": 850
    }
  ]
}
```

4. Configuration APIs

GET /config/processes

Propósito: Obtener configuración de procesos

Response:

```
{
  "processes": [
    {
      "developerName": "Lead_Motos_Conocimiento",
      "productFamily": "Seguros de Motos",
      "stage": "Conocimiento",
      "handlerClass": "MT_Kn_123",
      "isActive": true,
      "order": 0,
      "configuration": {
        "validateRUNT": true,
        "requireFasecolda": true
      }
    }
  ]
}
```

PUT /config/feature-flags/

Propósito: Actualizar feature flags

Request:

```
{
  "isEnabled": true,
  "environment": "Production",
  "effectiveDate": "2025-06-01",
  "configuration": {
    "rolloutPercentage": 50
  }
}
```

Métodos Apex Invocables

1. ProcessExecutor

Propósito: Ejecutar procesos desde Flows o Process Builder

```
@InvocableMethod(
  label='Execute Foundation Process'
  description='Executes a Foundation framework process'
```

```
)
public static List<ProcessResult> executeProcess(List<ProcessRequest> requests)
```

Input:

```
public class ProcessRequest {
  @InvocableVariable(required=true)
  public String processName;

  @InvocableVariable(required=true)
  public String productFamily;

  @InvocableVariable(required=true)
  public String recordId;

  @InvocableVariable
  public String parametersJSON;
}
```

Output:

```
public class ProcessResult {
  @InvocableVariable
  public Boolean success;

  @InvocableVariable
  public String executionId;

  @InvocableVariable
  public String status;

  @InvocableVariable
  public String errorMessage;

  @InvocableVariable
  public String resultJSON;
}
```

2. EventPublisher

Propósito: Publicar eventos desde automaciones

```
@InvocableMethod(
  label='Publish Foundation Event'
  description='Publishes an event to the Foundation event system'
)
public static List<EventResult> publishEvent(List<EventRequest> requests)
```

3. IntegrationCaller

Propósito: Llamar servicios externos desde Flows

```
@InvocableMethod(
  label='Call External Service'
  description='Calls an external service with retry logic'
)
public static List<IntegrationResult> callExternalService(List<IntegrationRequest> requests)
```

🔗 Integration Procedures (Vlocity)

1. SFFoundation_ProcessLauncher

Propósito: Lanzar procesos Foundation desde OmniScript

Input Node:

```
{
  "processConfig": {
    "processName": "%processName%",
    "productFamily": "%productFamily%",
    "stage": "%stage%"
  },
  "recordData": {
    "recordId": "%recordId%",
    "objectType": "%objectType%"
  },
  "parameters": "%processParameters%"
}
```

Output Node:

```
{
  "executionResult": {
    "success": true,
    "executionId": "a0X000000123456",
    "status": "Completed",
    "duration": 1250,
    "nextStage": "Tarificación"
  }
}
```

2. SFFoundation_ConfigurationReader

Propósito: Leer configuraciones del framework

Input:

```
{
  "configType": "ProcessConfiguration",
  "filters": {
    "productFamily": "Seguros de Motos",
    "isActive": true
  }
}
```

Output:

```
{
  "configurations": [
    {
      "processName": "LeadKnowing",
      "handlerClass": "MT_Kn_123",
      "order": 0,
      "settings": {}
    }
```

```
  ]
}
```

3. SFFoundation_EventHistoryReader

Propósito: Obtener historial de eventos

Input:

```
{
  "recordId": "%recordId%",
  "eventTypes": ["ProcessStarted", "ProcessCompleted"],
  "dateRange": {
    "startDate": "%startDate%",
    "endDate": "%endDate%"
  }
}
```

📡 Platform Events

Suscripciones Disponibles

SFFoundation_ProcessEvent__e

Trigger: Cambios en estado de procesos

Payload:

```
{
  "EventType__c": "ProcessCompleted",
  "ProcessName__c": "LeadKnowing",
  "ProductFamily__c": "Seguros de Motos",
  "Stage__c": "Conocimiento",
  "Status__c": "Success",
  "RecordId__c": "00Q000000123456",
  "Payload__c": "{\"duration\": 1250, \"nextStage\": \"Tarificación\"}"
}
```

SFFoundation_IntegrationEvent__e

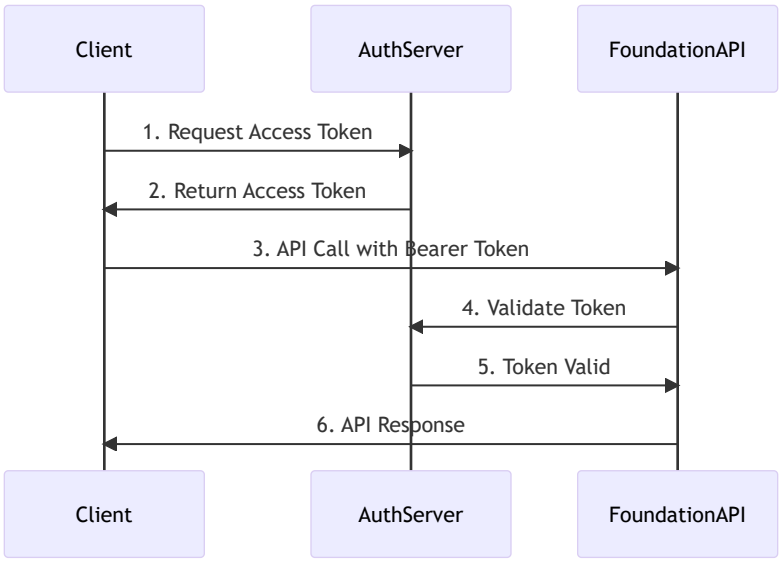
Trigger: Eventos de integraciones externas

Payload:

```
{
  "ServiceName__c": "RUNT_VehicleConsultation",
  "EventType__c": "Response",
  "Status__c": "Success",
  "HttpStatus__c": 200,
  "Duration__c": 850,
  "RequestId__c": "REQ-123456"
}
```

🔒 Seguridad y Autenticación

OAuth 2.0 Flow



Permisos Requeridos

Custom Permissions:

- `SFFoundation_ProcessExecutor`: Ejecutar procesos
- `SFFoundation_ConfigurationManager`: Gestionar configuraciones
- `SFFoundation_EventPublisher`: Publicar eventos
- `SFFoundation_IntegrationCaller`: Llamar servicios externos

Object Permissions:

- `SFFoundation_ProcessExecution__c`: Create, Read
- `SFFoundation_EventLog__c`: Create, Read
- `SFFoundation_IntegrationLog__c`: Create, Read

📊 Rate Limiting y Quotas

Límites por Endpoint

Endpoint	Requests/Hour	Burst Limit
`/process/execute`	1000	10/min
`/integration/external/call`	500	5/min
`/events/publish`	2000	20/min
`/config/*`	100	2/min

Manejo de Rate Limiting

Response Headers:

```
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 999
```

```
X-RateLimit-Reset: 1621840800
Retry-After: 60
```

Error Response (429):

```
{
  "error": {
    "code": "RATE_LIMIT_EXCEEDED",
    "message": "Rate limit exceeded. Retry after 60 seconds.",
    "retryAfter": 60
  }
}
```

🔍 Monitoreo y Observabilidad

Request/Response Logging

Automated Logging:

- Todas las API calls son loggeadas automáticamente
- Headers, payload y response capturados
- Métricas de performance incluidas

Log Structure:

```
{
  "requestId": "REQ-123456",
  "endpoint": "/process/execute",
  "method": "POST",
  "userId": "005000000123456",
  "requestTime": "2025-05-23T10:30:00Z",
  "responseTime": "2025-05-23T10:30:01.250Z",
  "duration": 1250,
  "httpStatus": 200,
  "success": true,
  "requestHeaders": {},
  "requestBody": {},
  "responseBody": {}
}
```

Health Check Endpoint

GET /health

Response:

```
{
  "status": "healthy",
  "version": "1.0.6",
  "timestamp": "2025-05-23T10:30:00Z",
  "components": {
    "database": {
      "status": "healthy",
      "responseTime": 15
    },
    "externalServices": {
      "RUNT": {
```

```
        "status": "healthy",
        "responseTime": 250
      },
      "FASECOLDA": {
        "status": "degraded",
        "responseTime": 1500,
        "lastError": "Timeout"
      }
    },
    "eventSystem": {
      "status": "healthy",
      "queueDepth": 12
    }
  }
}
```

📝 Códigos de Error Estándar

HTTP Status Codes

Código	Descripción	Acción Recomendada
200	Success	Continuar
201	Created	Recurso creado exitosamente
400	Bad Request	Verificar parámetros de entrada
401	Unauthorized	Verificar autenticación
403	Forbidden	Verificar permisos
404	Not Found	Verificar endpoint y recursos
429	Rate Limited	Implementar retry con backoff
500	Internal Error	Contactar soporte
503	Service Unavailable	Retry más tarde

Error Response Format

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid process configuration",
    "details": [
      {
        "field": "processName",
        "message": "Process name is required"
      },
      {
        "field": "productFamily",
        "message": "Invalid product family specified"
```

```
      }
    ],
    "requestId": "REQ-123456",
    "timestamp": "2025-05-23T10:30:00Z"
  }
}
```

🧪 Ejemplos de Integración

JavaScript/Node.js

```javascript
const axios = require('axios');

class FoundationClient {
  constructor(instanceUrl, accessToken) {
    this.baseURL = `${instanceUrl}/services/apexrest/sura/foundation/v1`;
    this.headers = {
      'Authorization': `Bearer ${accessToken}`,
      'Content-Type': 'application/json',
      'X-SURA-API-Version': 'v1'
    };
  }

  async executeProcess(processConfig) {
    try {
      const response = await axios.post(
        `${this.baseURL}/process/execute`,
        processConfig,
        { headers: this.headers }
      );
      return response.data;
    } catch (error) {
      throw new Error(`Process execution failed: ${error.message}`);
    }
  }

  async getProcessStatus(executionId) {
    try {
      const response = await axios.get(
        `${this.baseURL}/process/status/${executionId}`,
        { headers: this.headers }
      );
      return response.data;
    } catch (error) {
      throw new Error(`Status check failed: ${error.message}`);
    }
  }
}

// Uso
const client = new FoundationClient(
  'https://myorg.salesforce.com',
  'access_token_here'
);

const result = await client.executeProcess({
```

```
  processName: 'LeadKnowing',
  productFamily: 'Seguros de Motos',
  recordId: '00Q000000123456',
  parameters: {
    vehicleType: 'Moto',
    validateRUNT: true
  }
});
```

cURL Examples

```
# Execute Process
curl -X POST \
  https://myorg.salesforce.com/services/apexrest/sura/foundation/v1/process/execute \
  -H 'Authorization: Bearer ACCESS_TOKEN' \
  -H 'Content-Type: application/json' \
  -H 'X-SURA-API-Version: v1' \
  -d '{
    "processName": "LeadKnowing",
    "productFamily": "Seguros de Motos",
    "recordId": "00Q000000123456",
    "parameters": {
      "vehicleType": "Moto",
      "validateRUNT": true
    }
  }'

# Check Process Status
curl -X GET \
  https://myorg.salesforce.com/services/apexrest/sura/foundation/v1/process/status/a0X000000123456 \
  -H 'Authorization: Bearer ACCESS_TOKEN' \
  -H 'X-SURA-API-Version: v1'
```

📋 Postman Collection

Collection Structure

```
{
  "info": {
    "name": "SURA Foundation Framework APIs",
    "description": "Complete API collection for Foundation Framework",
    "version": "1.0.0"
  },
  "variable": [
    {
      "key": "baseUrl",
      "value": "{{instanceUrl}}/services/apexrest/sura/foundation/v1"
    },
    {
      "key": "accessToken",
      "value": "{{token}}"
    }
  ],
  "auth": {
    "type": "bearer",
    "bearer": [
```

```
      {
        "key": "token",
        "value": "{{accessToken}}"
      }
    ]
  }
}
```

🚀 Best Practices

Para Desarrolladores

1. **Versionado**: Siempre especificar versión de API
2. **Error Handling**: Implementar retry logic para errores 5xx
3. **Timeouts**: Configurar timeouts apropiados (30s recomendado)
4. **Logging**: Loggear requests/responses para debugging
5. **Rate Limiting**: Implementar backoff exponencial

Para Integraciones

1. **Async Processing**: Usar ejecución asíncrona para procesos largos
2. **Batch Operations**: Agrupar operaciones cuando sea posible
3. **Caching**: Cachear configuraciones que no cambian frecuentemente
4. **Monitoring**: Implementar health checks y alertas
5. **Security**: Renovar tokens antes de expiración

Fecha de Actualización: Mayo 2025**Versión de API:** v1.0**Mantenido por:** Equipo de Integración SURA Foundation

QuoteUtils

Descripción General

`QuoteUtils` es una clase virtual global que proporciona utilidades para trabajar con cotizaciones (Quote) en Salesforce. Esta clase ofrece métodos para validar, actualizar y consultar cotizaciones, así como para establecer relaciones entre cotizaciones, cuentas y oportunidades.

Autor

Jean Carlos Melendez \<jean@cloudblue.us\>

Última Modificación

31 de enero de 2025 por Esneyder Zabala

Constantes

Nombre	Tipo	Descripción
`MAIN_NODE`	`String`	Clave del nodo principal en los datos de entrada para cotizaciones: 'quotepolicyJson'
`REQUIRED_PARAMETERS`	`Set<String>`	Conjunto de parámetros requeridos para la cotización: 'term', 'productConfigurationDetail', 'insuredItems', 'additionalFields', 'OpportunityDetails'

Métodos

isValidSalesforceId(String objectId)

Descripción

Comprueba si un ID proporcionado es un ID de Salesforce válido que corresponde a algún objeto.

Parámetros

- `objectId` (String): El ID de Salesforce a validar.

Retorno

- `Boolean`: `true` si el ID es válido y corresponde a un objeto, `false` en caso contrario.

Proceso

1. Intenta convertir la cadena a un ID de Salesforce.
2. Si la conversión es exitosa, obtiene el tipo de objeto asociado al ID.
3. Verifica que el tipo de objeto no esté en blanco.
4. Si ocurre alguna excepción durante el proceso, retorna `false`.

Código

```
global static Boolean isValidSalesforceId(String objectId) {
  try {
    Id validId = Id.valueOf(objectId); // Lanza excepción si el ID no es válido
    String sObjectType = validId.getSObjectType().getDescribe().getName();
    return String.isNotBlank(sObjectType);
```

```
  } catch (Exception e) {
    return false; // El ID no es válido o no pertenece a un objeto conocido
  }
}
```

validateInputMapForQuoting(Map<String, Object> inputMap)

Descripción

Verifica si todos los nodos principales requeridos están presentes en el mapa de entrada para un proceso de cotización.

Parámetros

- `inputMap` (Map<String, Object>): Mapa de entrada que contiene los datos a validar.

Retorno

- `Map<Boolean, Object>`: Mapa con el resultado de la validación. La clave es un booleano que indica si la validación fue exitosa, y el valor es un mensaje descriptivo.

Proceso

1. Verifica la existencia del nodo principal (`MAIN_NODE`).
2. Obtiene el mapa correspondiente al nodo principal.
3. Verifica la existencia de cada uno de los parámetros requeridos definidos en `REQUIRED_PARAMETERS`.
4. Devuelve un mapa con el resultado de la validación y un mensaje descriptivo.

Código

```
global static Map<Boolean, Object> validateInputMapForQuoting(
  Map<String, Object> inputMap
) {
  if (!inputMap.containsKey(MAIN_NODE) || inputMap?.get(MAIN_NODE) == null) {
    return createValidationResult(
      false,
      'Falta el nodo principal requerido: ' + MAIN_NODE
    );
  }
  Map<String, Object> mainNode = (Map<String, Object>) inputMap.get(
    MAIN_NODE
  );

  for (String node : REQUIRED_PARAMETERS) {
    if (!mainNode.containsKey(node)) {
      return createValidationResult(
        false,
        'Falta el nodo requerido: ' + node
      );
    }
  }
  System.debug('Todos los nodos requeridos están presentes.');
  return createValidationResult(
    true,
    'Todos los parámetros requeridos están presentes.'
  );
}
```

createValidationResult(Boolean isValid, String message)

Descripción

Método privado estático que crea un mapa estándar para representar el resultado de una validación.

Parámetros

- `isValid` (Boolean): Indicador de si la validación fue exitosa.
- `message` (String): Mensaje descriptivo del resultado.

Retorno

- `Map<Boolean, Object>`: Mapa con el estado de validación y el mensaje.

Código

```
private static Map<Boolean, Object> createValidationResult(
  Boolean isValid,
  String message
) {
  return new Map<Boolean, Object>{ isValid => message };
}
```

updateQuoteState(Id quoteId, String status)

Descripción

Actualiza el estado de una cotización, verificando primero que el valor del estado sea válido.

Parámetros

- `quoteId` (Id): Id de la cotización a actualizar.
- `status` (String): Nuevo valor para el campo Status.

Retorno

- `SObject`: La cotización actualizada.

Proceso

1. Obtiene los metadatos del objeto Quote para acceder a la descripción del campo Status.
2. Verifica que el valor de estado proporcionado sea válido según los valores de la lista de selección (picklist).
3. Si el valor no es válido, lanza una excepción.
4. Consulta la cotización por su ID.
5. Actualiza el campo Status con el nuevo valor.
6. Guarda los cambios.
7. Devuelve la cotización actualizada.

Código

```
global static SObject updateQuoteState(Id quoteId, String status) {
  // Obtener la metadata del objeto Quote
  Schema.SObjectType quoteSchema = Schema.getGlobalDescribe().get('Quote');
  Schema.DescribeSObjectResult quoteDescribe = quoteSchema.getDescribe();
```

```
  Map<String, Schema.SObjectField> fieldMap = quoteDescribe.fields.getMap();
  Schema.DescribeFieldResult statusFieldDescribe = fieldMap.get('Status')
    .getDescribe();

  if (!CorePicklistUtils.containsVal(fieldMap.get('Status'), status)) {
    throw new ProductHandlerImplementationException(
      'invalid opportunity stage',
      'UNPROCESSABLE_ENTITY',
      422
    );
  }
  String query = 'SELECT Id, Status FROM Quote WHERE Id = :quoteId';
  List<SObject> results = Database.query(query);
  SObject quote = results[0];
  quote.put('Status', status);
  System.debug('quote utils: ' + quote);
  update quote;
  System.debug('quote utils update ok: ' + quote);
  return quote;
}
```

getQuoteById(Id quoteId)

Descripción

Obtiene una cotización por su ID, incluyendo campos específicos.

Parámetros

- `quoteId` (Id): Id de la cotización a consultar.

Retorno

- `SObject`: La cotización encontrada.

Código

```
global static SObject getQuoteById(Id quoteId) {
  String query = 'SELECT Id, Name, AccountId, OpportunityId, Status, OwnerId FROM Quote WHERE Id = :quoteId
LIMIT 1';
  List<SObject> quotes = Database.query(query);
  return quotes[0];
}
```

setInsuredPartyToQuoteAndOpp(List<Map<String, Object>> insuredPeople, SObject quote)

Descripción

Método virtual que establece las partes aseguradas en una cotización y actualiza la cuenta asociada a la oportunidad relacionada.

Parámetros

- `insuredPeople` (List<Map<String, Object>>): Lista de mapas que representan las personas aseguradas.
- `quote` (SObject): La cotización a la que se asociarán las partes aseguradas.

Retorno

- `void`

Proceso

1. Obtiene el ID de la oportunidad asociada a la cotización.
2. Consulta la oportunidad para obtener sus detalles.
3. Recorre la lista de personas aseguradas.
4. Para cada persona, si contiene una clave 'Asegurado' con una cuenta asociada, actualiza la cuenta de la oportunidad.
5. Actualiza tanto la cotización como la oportunidad para guardar los cambios.

Código

```
global virtual void setInsuredPartyToQuoteAndOpp(
  List<Map<String, Object>> insuredPeople,
  SObject quote
) {
  // Obtener OpportunityId de la Quote
  Id opportunityId = (Id) quote.get('OpportunityId');

  Opportunity opp = [
    SELECT Id, AccountId
    FROM Opportunity
    WHERE Id = :opportunityId
  ];

  for (Map<String, Object> person : insuredPeople) {
    System.debug(person);
    if (person.containsKey('Asegurado')) {
      Account account = (Account) person.get('Asegurado');
      System.debug(account);
      if (account != null) {
        opp.AccountId = account.Id;
      }
    }
  }
  update quote;
  update opp;
}
```

Excepciones

ProductHandlerImplementationException

Esta excepción se lanza en el método `updateQuoteState` cuando se proporciona un valor de estado no válido para la cotización.

Dependencias

- `CorePicklistUtils`: Clase utilitaria que contiene el método `containsVal` para verificar si un valor está presente en una lista de selección.
- `ProductHandlerImplementationException`: Excepción personalizada lanzada cuando hay errores en la implementación del manejador de productos.
- `Quote`: Objeto estándar de Salesforce para cotizaciones.
- `Opportunity`: Objeto estándar de Salesforce para oportunidades.
- `Account`: Objeto estándar de Salesforce para cuentas.

Estructura de Datos Esperada

La clase espera que los datos de entrada para la validación de cotizaciones tengan la siguiente estructura:

```
{
  "quotepolicyJson": {
    "term": { /* ... */ },
    "productConfigurationDetail": { /* ... */ },
    "insuredItems": [ /* ... */ ],
    "additionalFields": { /* ... */ },
    "OpportunityDetails": { /* ... */ }
  },
  /* Otros campos opcionales */
}
```

Ejemplo de Uso

Validar datos de entrada para cotización

```
// Preparar datos de entrada para validación
Map<String, Object> term = new Map<String, Object>{
  'startDate' => Date.today(),
  'endDate' => Date.today().addYears(1)
};

Map<String, Object> productConfigDetail = new Map<String, Object>{
  'productCode' => 'AUTO-001',
  'coverageLevel' => 'Premium'
};

List<Map<String, Object>> insuredItems = new List<Map<String, Object>>{
  new Map<String, Object>{
    'type' => 'Vehicle',
    'make' => 'Toyota',
    'model' => 'Corolla',
    'year' => 2020
  }
};

Map<String, Object> additionalFields = new Map<String, Object>{
  'discountCode' => 'NEWCUST10'
};

Map<String, Object> opportunityDetails = new Map<String, Object>{
  'opportunityId' => '0061x00000AbCdEf',
  'accountId' => '0011x00000GhIjKl'
};

Map<String, Object> quotepolicyJson = new Map<String, Object>{
  'term' => term,
  'productConfigurationDetail' => productConfigDetail,
  'insuredItems' => insuredItems,
  'additionalFields' => additionalFields,
  'OpportunityDetails' => opportunityDetails
};
```

```
Map<String, Object> input = new Map<String, Object>{
  'quotepolicyJson' => quotepolicyJson
};

// Validar los datos de entrada
Map<Boolean, Object> validationResult = QuoteUtils.validateInputMapForQuoting(input);
if (validationResult.containsKey(false)) {
  System.debug('Error de validación: ' + validationResult.get(false));
  // Manejar el error
} else {
  System.debug('Validación exitosa: ' + validationResult.get(true));
  // Continuar con el proceso
}
```

Actualizar el estado de una cotización

```
// ID de la cotización a actualizar
Id quoteId = '0Q00x000000XXXXX';

try {
  // Actualizar el estado a "Approved"
  SObject updatedQuote = QuoteUtils.updateQuoteState(quoteId, 'Approved');
  System.debug('Cotización actualizada: ' + updatedQuote);
} catch (ProductHandlerImplementationException e) {
  System.debug('Error al actualizar el estado: ' + e.getMessage());
}
```

Obtener una cotización por ID

```
// ID de la cotización a consultar
Id quoteId = '0Q00x000000XXXXX';

// Obtener la cotización
SObject quote = QuoteUtils.getQuoteById(quoteId);
System.debug('Cotización obtenida: ' + quote);
```

Establecer partes aseguradas

```
// ID de la cotización
Id quoteId = '0Q00x000000XXXXX';

// Obtener la cotización
SObject quote = QuoteUtils.getQuoteById(quoteId);

// Crear cuenta para el asegurado
Account insuredAccount = new Account(
  Name = 'Asegurado Principal',
  Phone = '555-1234',
  BillingCity = 'Ciudad Ejemplo'
);
insert insuredAccount;

// Crear lista de partes aseguradas
List<Map<String, Object>> insuredParties = new List<Map<String, Object>>{
  new Map<String, Object>{
    'Asegurado' => insuredAccount,
```

```
    'TipoAsegurado' => 'Principal'
  }
};

// Establecer las partes aseguradas
QuoteUtils quoteUtils = new QuoteUtils();
quoteUtils.setInsuredPartyToQuoteAndOpp(insuredParties, quote);
```

Extensión de la Clase

Siendo una clase virtual, `QuoteUtils` puede ser extendida para personalizar comportamientos específicos:

```
public class CustomQuoteUtils extends QuoteUtils {

    // Sobrescribir el método para añadir lógica personalizada
    global override void setInsuredPartyToQuoteAndOpp(
        List<Map<String, Object>> insuredPeople,
        SObject quote
    ) {
        // Obtener OpportunityId de la Quote
        Id opportunityId = (Id) quote.get('OpportunityId');

        Opportunity opp = [
            SELECT Id, AccountId, StageName, Amount
            FROM Opportunity
            WHERE Id = :opportunityId
        ];

        // Lógica personalizada para determinar la cuenta principal
        Account primaryAccount = null;
        Map<String, Account> accountsByType = new Map<String, Account>();

        // Clasificar cuentas por tipo
        for (Map<String, Object> person : insuredPeople) {
            if (person.containsKey('Asegurado')) {
                Account account = (Account) person.get('Asegurado');
                String tipoAsegurado = (String) person.get('TipoAsegurado');

                if (account != null) {
                    accountsByType.put(tipoAsegurado, account);

                    if (tipoAsegurado == 'Principal' || primaryAccount == null) {
                        primaryAccount = account;
                    }
                }
            }
        }

        // Establecer la cuenta principal en la oportunidad
        if (primaryAccount != null) {
            opp.AccountId = primaryAccount.Id;
        }

        // Actualizar campos adicionales en la oportunidad basados en la cotización
        quote.put('Insured_Party_Count__c', insuredPeople.size());

        // Llamar al método personalizado para crear contactos de roles
```

```
        createContactRoles(opp.Id, accountsByType);

        // Guardar cambios
        update quote;
        update opp;
    }

    // Método adicional para crear roles de contacto
    private void createContactRoles(Id opportunityId, Map<String, Account> accountsByType) {
        List<OpportunityContactRole> roles = new List<OpportunityContactRole>();

        for (String type : accountsByType.keySet()) {
            Account acc = accountsByType.get(type);

            // Obtener el contacto primario de la cuenta
            Contact primaryContact = [
                SELECT Id
                FROM Contact
                WHERE AccountId = :acc.Id
                LIMIT 1
            ];

            if (primaryContact != null) {
                OpportunityContactRole role = new OpportunityContactRole(
                    OpportunityId = opportunityId,
                    ContactId = primaryContact.Id,
                    Role = mapTypeToRole(type)
                );
                roles.add(role);
            }
        }

        if (!roles.isEmpty()) {
            insert roles;
        }
    }

    // Método para mapear tipo de asegurado a rol de contacto
    private String mapTypeToRole(String type) {
        switch on type {
            when 'Principal' {
                return 'Decision Maker';
            }
            when 'Adicional' {
                return 'Influencer';
            }
            when else {
                return 'Other';
            }
        }
    }
}
```

Consideraciones y Mejores Prácticas

1. **Manejo de Excepciones**: La clase maneja excepciones en algunos métodos pero no en todos. Considerar implementar un manejo consistente de excepciones en todos los métodos.
2. **Métodos Virtuales**: El método `setInsuredPartyToQuoteAndOpp` está marcado como virtual, permitiendo su personalización en clases derivadas. Al extender la clase, asegurarse de mantener la funcionalidad básica o llamar al método base según sea necesario.
3. **DML en Bucles**: El método `setInsuredPartyToQuoteAndOpp` realiza operaciones DML fuera de los bucles, lo cual es una buena práctica para evitar alcanzar límites de gobernador.
4. **Validación de Entradas**: La clase incluye métodos para validar la estructura de los datos de entrada, lo que es una buena práctica para garantizar la integridad de los datos.
5. **Uso de Consultas Dinámicas**: La clase utiliza consultas dinámicas para obtener objetos. Cuando sea posible, considerar el uso de consultas estáticas para aprovechar la verificación en tiempo de compilación.

Notas Adicionales

1. La clase utiliza el operador de navegación segura (`?.`) para evitar excepciones de referencia nula al acceder a valores en mapas.
2. El mensaje de excepción en `updateQuoteState` menciona "invalid opportunity stage" aunque está validando el estado de una cotización, lo que podría indicar un error o reutilización de código.
3. La clase parece formar parte de un sistema más amplio para la gestión de cotizaciones y productos, interactuando con otras clases como `CorePicklistUtils` y `ProductHandlerImplementationException`.
4. El uso de la palabra clave `global` indica que esta clase está diseñada para ser accesible desde paquetes gestionados o diferentes namespaces.

Guías de Implementación - SURA Foundation Framework

Resumen Ejecutivo

Esta guía proporciona instrucciones completas para la implementación, configuración y despliegue de **SURA Foundation Framework**. Incluye desde la instalación inicial hasta la configuración avanzada, mejores prácticas y troubleshooting para garantizar una implementación exitosa.

🎯 Prerrequisitos

Requisitos de Ambiente

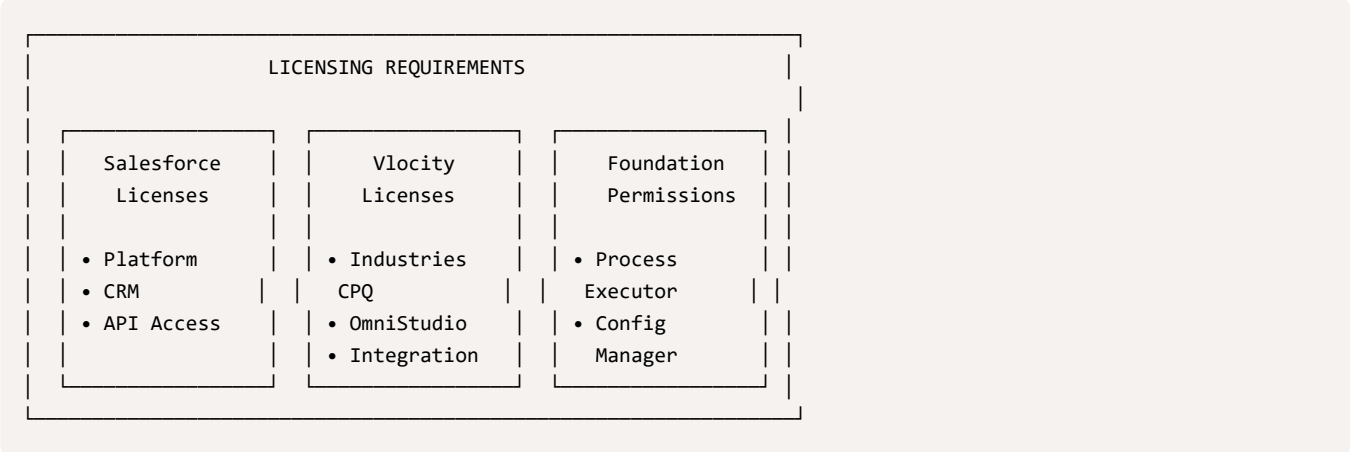
Salesforce Organization

- **Edición**: Enterprise, Unlimited, o Developer Edition
- **API Version**: 58.0 o superior
- **My Domain**: Configurado y desplegado
- **Lightning Experience**: Habilitado

Vlocity/OmniStudio

- **OmniStudio**: Versión 2024.1 o superior
- **Industries CPQ**: Instalado y configurado
- **Integration Procedures**: Disponibles
- **Data Raptors**: Configurados

Permisos y Licencias



Usuarios y Roles

Roles Requeridos:

- **System Administrator**: Instalación y configuración inicial
- **Foundation Administrator**: Gestión de procesos y configuraciones
- **Integration Specialist**: Configuración de servicios externos

- **Developer**: Extensiones y customizaciones

📦 Instalación del Managed Package

Paso 1: Descarga e Instalación

Desde AppExchange (Recomendado)

```
1. Acceder a AppExchange
2. Buscar "SURA Foundation Framework"
3. Hacer clic en "Get It Now"
4. Seleccionar ambiente de destino
5. Confirmar instalación
```

Instalación Manual (Sandbox)

```
# URL de instalación directa
https://login.salesforce.com/packaging/installPackage.apexp?p0=04t...

# Para Sandbox
https://test.salesforce.com/packaging/installPackage.apexp?p0=04t...
```

Package ID por Versión

```
Versión 1.0-1: 04tXXXXXXXXXXX1
Versión 1.0-3: 04tXXXXXXXXXXX3
Versión 1.0-4: 04tXXXXXXXXXXX4
Versión 1.0-5: 04tXXXXXXXXXXX5
Versión 1.0-6: 04tXXXXXXXXXXX6 (Actual)
```

Paso 2: Configuración de Instalación

Opciones de Instalación

```
┌─────────────────────────────────────────────────────────┐
│                INSTALLATION OPTIONS                     │
│                                                         │
│  [x] Install for All Users                              │
│  [ ] Install for Admins Only                            │
│  [ ] Install for Specific Profiles                      │
│                                                         │
│  Security Settings:                                     │
│  [x] Grant access to all data                           │
│  [x] Enable Apex class access                           │
│  [x] Enable custom tabs                                 │
│                                                         │
│  Advanced Options:                                      │
│  [x] Compile all Apex classes                           │
│  [x] Enable push notifications                          │
│  [x] Upgrade existing components                        │
└─────────────────────────────────────────────────────────┘
```

Paso 3: Validación de Instalación

Script de Validación

```
// Ejecutar en Developer Console
System.debug('=== SURA Foundation Installation Validation ===');
```

```
// Verificar clases instaladas
Type coreClass = Type.forName('SFSura', 'Core');
System.debug('Core class installed: ' + (coreClass != null));

// Verificar objetos personalizados
Schema.DescribeSObjectResult eventLogDesc =
    SFFoundation_EventLog__c.SObjectType.getDescribe();
System.debug('EventLog object installed: ' + eventLogDesc.isCreateable());

// Verificar Custom Metadata Types
List<SFFoundation_ProcessConfiguration__mdt> configs =
    [SELECT Id FROM SFFoundation_ProcessConfiguration__mdt LIMIT 1];
System.debug('Process configurations available: ' + !configs.isEmpty());

System.debug('=== Installation validation complete ===');
```

⚙️ Configuración Inicial

Paso 1: Configuración de Metadatos

Instalar Configuraciones Básicas

```
// Ejecutar desde el Panel de Administración
// Navegación: Setup > Apps > App Manager > SURA Foundation

1. Hacer clic en "Instalar metadatos y configuraciones iniciales"
2. Seleccionar productos a configurar:
   [x] Seguros de Motos
   [x] Seguros de Autos
   [ ] Seguros de Viajes (Beta)
   [ ] Arrendamiento (Beta)
3. Confirmar instalación
```

Configuración Manual de Procesos

```
-- Custom Metadata Type: SFFoundation_ProcessConfiguration__mdt

INSERT SFFoundation_ProcessConfiguration__mdt (
    DeveloperName = 'Lead_Motos_Conocimiento',
    MasterLabel = 'Lead Motos - Conocimiento',
    ProductFamily__c = 'Seguros de Motos',
    Stage__c = 'Conocimiento',
    HandlerClass__c = 'MT_Kn_123',
    Order__c = 0,
    IsActive__c = true,
    ApiName__c = 'leadMotosKnowing'
);
```

Paso 2: Feature Flags

Configuración Básica de Features

```
{
  "featureFlags": [
    {
      "developerName": "EnableFoundationLogging",
      "isEnabled": true,
```

```
      "environment": "All",
      "scope": "Global",
      "description": "Habilita logging del framework"
    },
    {
      "developerName": "EnableAdvancedValidations",
      "isEnabled": false,
      "environment": "Production",
      "scope": "Global",
      "description": "Validaciones avanzadas de negocio"
    },
    {
      "developerName": "EnableRUNTIntegration",
      "isEnabled": true,
      "environment": "All",
      "scope": "Product",
      "productFamily": "Seguros de Motos",
      "description": "Integración con RUNT para motos"
    }
  ]
}
```

Paso 3: Configuración de Integraciones

RUNT Integration

```
// Custom Metadata: SFFoundation_IntegrationConfiguration__mdt

ServiceName__c: 'RUNT_VehicleConsultation'
BaseURL__c: 'https://api.runt.gov.co/v2'
AuthMethod__c: 'API_Key'
TimeoutMs__c: 30000
MaxRetries__c: 3
RetryPattern__c: 'Fibonacci'
RetryIntervalMs__c: 1000
RetryOnHttpCodes__c: '500,502,503,504'
IsActive__c: true
```

FASECOLDA Integration

```
ServiceName__c: 'FASECOLDA_VehicleData'
BaseURL__c: 'https://api.fasecolda.com/v1'
AuthMethod__c: 'Bearer_Token'
TimeoutMs__c: 20000
MaxRetries__c: 2
RetryPattern__c: 'Incremental'
RetryIntervalMs__c: 500
RetryOnHttpCodes__c: '429,500,502,503'
IsActive__c: true
```

🔧 Configuración por Ambiente

Desarrollo (Sandbox)

Configuraciones Específicas

```
{
  "environment": "Development",
```

```
  "settings": {
    "logging": {
      "level": "DEBUG",
      "retentionDays": 7
    },
    "integrations": {
      "timeout": 60000,
      "enableMocks": true,
      "bypassValidations": true
    },
    "features": {
      "enableAllFeatures": true,
      "debugMode": true
    }
  }
}
```

Mock Services

```
// Habilitar servicios simulados para desarrollo
SFFoundation_GlobalSettings__c devSettings = new SFFoundation_GlobalSettings__c();
devSettings.Name = 'Development';
devSettings.IsEnabled__c = true;
devSettings.LogLevel__c = 'DEBUG';
devSettings.DefaultTimeout__c = 60000;
insert devSettings;
```

Testing (UAT)

Configuraciones de Prueba

```
{
  "environment": "Testing",
  "settings": {
    "logging": {
      "level": "INFO",
      "retentionDays": 30
    },
    "integrations": {
      "timeout": 30000,
      "enableMocks": false,
      "useTestEndpoints": true
    },
    "features": {
      "gradualRollout": true,
      "rolloutPercentage": 50
    }
  }
}
```

Producción

Configuraciones de Producción

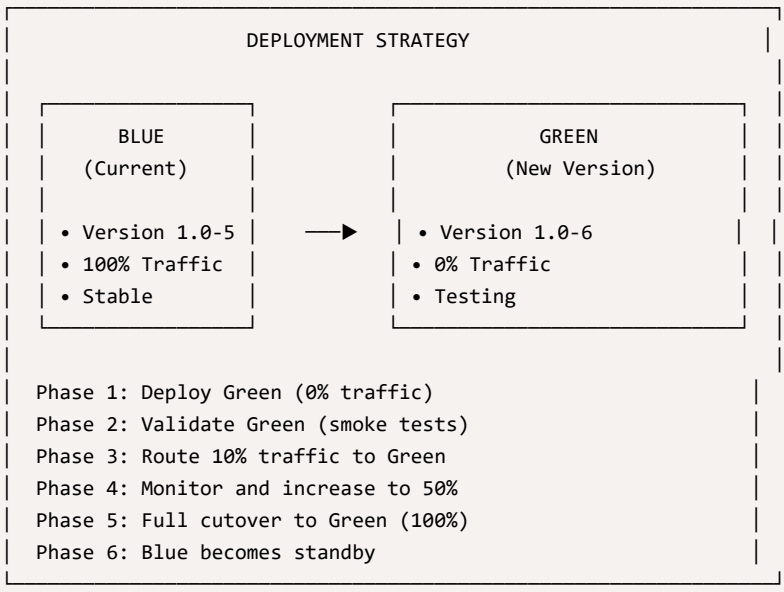
```
{
  "environment": "Production",
  "settings": {
    "logging": {
```

```
      "level": "ERROR",
      "retentionDays": 90
    },
    "integrations": {
      "timeout": 30000,
      "enableMocks": false,
      "useProductionEndpoints": true
    },
    "features": {
      "conservativeRollout": true,
      "rolloutPercentage": 10
    }
  }
}
```

🚀 Despliegue y Migración

Estrategia de Despliegue

Blue-Green Deployment



Rolling Deployment por Productos

```
Week 1: Seguros de Motos (Pilot)
  - Deploy to 10% of users
  - Monitor for 48 hours
  - Scale to 50% if successful

Week 2: Seguros de Autos
  - Same progressive rollout
  - Parallel monitoring

Week 3: Full Production
  - Complete rollout
  - Legacy system sunset
```

Checklist de Pre-Despliegue

Validaciones Técnicas

```
□ Package installation successful
□ All Apex classes compiled
□ Custom objects accessible
□ Metadata types populated
□ Platform events configured
□ Integration endpoints tested
□ Feature flags configured
□ Permissions assigned
□ Test coverage > 75%
□ Performance benchmarks met
```

Validaciones de Negocio

```
□ Process configurations validated
□ Business rules tested
□ Integration data validated
□ User acceptance tests passed
□ Training materials updated
□ Support team briefed
□ Rollback plan prepared
□ Communication plan executed
```

👥 Configuración de Usuarios y Permisos

Permission Sets

SFFoundation_ProcessExecutor

```
// Permisos para ejecutar procesos
Custom Permissions:
- SFFoundation_ProcessExecutor: true

Object Permissions:
- SFFoundation_ProcessExecution__c: Create, Read
- SFFoundation_EventLog__c: Create, Read
- Lead: Read, Edit
- Opportunity: Read, Edit
- Quote: Read, Edit

Apex Classes:
- SFSura.Core: Enabled
- SFSura.ProcessManager: Enabled
- SFSura.LeadHandler: Enabled
```

SFFoundation_ConfigurationManager

```
// Permisos para gestionar configuraciones
Custom Permissions:
- SFFoundation_ConfigurationManager: true

Object Permissions:
- SFFoundation_ProcessConfiguration__mdt: Read
- SFFoundation_FeatureFlag__mdt: Read, Edit
- SFFoundation_IntegrationConfiguration__mdt: Read
```

```
Apps:
- SURA Foundation Admin: Visible
```

SFFoundation_IntegrationUser

```
// Permisos para integraciones
Custom Permissions:
- SFFoundation_IntegrationCaller: true

Object Permissions:
- SFFoundation_IntegrationLog__c: Create, Read
- SFFoundation_EventLog__c: Create, Read

Apex Classes:
- SFSura.ExternalServiceHandler: Enabled
- SFSura.IntegrationManager: Enabled
```

Asignación de Permisos

Por Rol

```
// System Administrator
- SFFoundation_ProcessExecutor
- SFFoundation_ConfigurationManager
- SFFoundation_IntegrationUser

// Sales User
- SFFoundation_ProcessExecutor

// Integration User
- SFFoundation_IntegrationUser

// Support User
- SFFoundation_ProcessExecutor (Read-only)
```

🔗 Configuración de Integraciones

Named Credentials

RUNT Integration

```
Name: RUNT_API_Credential
URL: https://api.runt.gov.co/v2
Identity Type: Named Principal
Authentication Protocol: Password Authentication
Username: ${system_username}
Password: ${system_password}

Custom Headers:
- X-API-Key: ${api_key}
- Content-Type: application/json
```

FASECOLDA Integration

```
Name: FASECOLDA_API_Credential
URL: https://api.fasecolda.com/v1
Identity Type: Named Principal
Authentication Protocol: OAuth 2.0
```

```
Client ID: ${client_id}
Client Secret: ${client_secret}
Scope: vehicle_data

Custom Headers:
- Accept: application/json
- X-Client-Version: 1.0
```

Remote Site Settings

```
// RUNT
Name: RUNT_API
Remote Site URL: https://api.runt.gov.co
Active: true
Description: RUNT vehicle consultation service

// FASECOLDA
Name: FASECOLDA_API
Remote Site URL: https://api.fasecolda.com
Active: true
Description: FASECOLDA vehicle data service

// Payment Gateway
Name: Payment_Gateway
Remote Site URL: https://payments.sura.com
Active: true
Description: SURA payment processing gateway
```

📊 Monitoreo y Mantenimiento

Dashboard de Monitoreo

Configuración de Dashboard

```
Dashboard Name: SURA Foundation Monitoring

Components:
1. Process Execution Summary (Last 24h)
   - Total executions
   - Success rate
   - Average duration
   - Error distribution

2. Integration Health
   - Service availability
   - Response times
   - Error rates
   - Retry statistics

3. System Performance
   - CPU usage
   - Memory usage
   - API limits
   - Storage usage

4. Business Metrics
   - Leads processed
```

```
   - Quotes generated
   - Policies issued
   - Revenue impact
```

Reports de Monitoreo

```sql
-- Process Performance Report
SELECT
  ProcessName__c,
  COUNT(Id) as TotalExecutions,
  AVG(Duration__c) as AvgDuration,
  COUNT(CASE WHEN Status__c = 'Failed' THEN 1 END) as FailedExecutions
FROM SFFoundation_ProcessExecution__c
WHERE CreatedDate = LAST_N_DAYS:7
GROUP BY ProcessName__c
ORDER BY TotalExecutions DESC

-- Integration Health Report
SELECT
  ServiceName__c,
  AVG(Duration__c) as AvgResponseTime,
  COUNT(CASE WHEN IsSuccess__c = true THEN 1 END) as SuccessCount,
  COUNT(CASE WHEN IsSuccess__c = false THEN 1 END) as ErrorCount
FROM SFFoundation_IntegrationLog__c
WHERE CreatedDate = TODAY
GROUP BY ServiceName__c
```

Alertas Automáticas

Email Alerts

```
// Configurar en Workflow Rules o Process Builder

Alert: High Error Rate
Condition: Error rate > 5% in last hour
Recipients: sysadmin@sura.com, devops@sura.com

Alert: Integration Failure
Condition: Service unavailable > 5 minutes
Recipients: integration-team@sura.com

Alert: Performance Degradation
Condition: Avg response time > 5 seconds
Recipients: performance-team@sura.com
```

Mantenimiento Regular

Tareas Semanales

```
□ Review error logs
□ Check integration health
□ Monitor performance metrics
□ Update feature flag configurations
□ Review capacity utilization
□ Backup configuration metadata
```

Tareas Mensuales

```
□ Archive old event logs
□ Review and optimize queries
□ Update integration credentials
□ Performance testing
□ Security review
□ Documentation updates
```

🔧 Configuración Avanzada

Custom Settings Avanzadas

Performance Tuning

```
SFFoundation_GlobalSettings__c perfSettings = new SFFoundation_GlobalSettings__c();
perfSettings.Name = 'PerformanceSettings';
perfSettings.DefaultTimeout__c = 30000;
perfSettings.MaxRetries__c = 3;
perfSettings.CleanupBatchSize__c = 200;
perfSettings.LogLevel__c = 'INFO';

insert perfSettings;
```

Configuración de Cache

```
{
  "cacheSettings": {
    "processConfigurations": {
      "ttl": 3600,
      "maxSize": 1000,
      "enabled": true
    },
    "featureFlags": {
      "ttl": 1800,
      "maxSize": 500,
      "enabled": true
    },
    "integrationConfigs": {
      "ttl": 7200,
      "maxSize": 100,
      "enabled": true
    }
  }
}
```

Configuración de Batch Jobs

Event Log Cleanup

```
// Schedule cleanup job
SFFoundation_EventLogCleanupBatch cleanup = new SFFoundation_EventLogCleanupBatch();
String cronExpression = '0 0 2 * * ?'; // Daily at 2 AM
System.schedule('Foundation Event Log Cleanup', cronExpression, cleanup);
```

Integration Health Check

```
// Schedule health check job
SFFoundation_HealthCheckBatch healthCheck = new SFFoundation_HealthCheckBatch();
```

```
String cronExpression = '0 */15 * * * ?'; // Every 15 minutes
System.schedule('Foundation Health Check', cronExpression, healthCheck);
```

🦎 Troubleshooting

Problemas Comunes

1. Package Installation Failures

```
Error: "Package installation failed due to missing dependencies"

Solución:
1. Verificar que Vlocity/OmniStudio esté instalado
2. Confirmar versión de API compatible
3. Revisar permisos de usuario instalador
4. Verificar límites de storage
```

2. Process Execution Errors

```
Error: "Handler class not found: MT_Kn_123"

Solución:
1. Verificar configuración en ProcessConfiguration__mdt
2. Confirmar que la clase existe y es accesible
3. Revisar permisos de Apex class access
4. Validar naming convention
```

3. Integration Timeouts

```
Error: "Integration timeout after 30000ms"

Solución:
1. Aumentar timeout en IntegrationConfiguration__mdt
2. Verificar conectividad de red
3. Revisar Named Credentials
4. Contactar proveedor del servicio
```

4. Performance Issues

```
Error: "Process execution taking too long"

Solución:
1. Revisar governor limits usage
2. Optimizar queries SOQL
3. Implementar async processing
4. Revisar bulk operations
```

Logs de Debug

Habilitar Debug Logging

```
// Configurar trace flags para debugging
1. Setup > Debug Logs
2. New Trace Flag
3. Traced Entity Type: User
4. Start Date: Today
5. Expiration Date: Tomorrow
6. Debug Level: Foundation_Debug
```

```
// Custom Debug Level
Apex Code: DEBUG
Apex Profiling: INFO
Callout: DEBUG
Database: DEBUG
System: DEBUG
Validation: INFO
Visualforce: INFO
Workflow: INFO
```

Analizar Logs

```
// Buscar patrones en logs
System.debug('=== Foundation Process Start ===');
System.debug('Process: ' + processName);
System.debug('Duration: ' + duration + 'ms');
System.debug('=== Foundation Process End ===');

// Filtrar en Developer Console:
// USER_DEBUG|Foundation
```

📋 Checklist de Go-Live

Pre-Go-Live (1 semana antes)

Validaciones Técnicas

```
□ All environments synchronized
□ Production data validated
□ Performance benchmarks met
□ Security scan completed
□ Backup procedures tested
□ Rollback plan validated
□ Monitoring alerts configured
□ Load testing completed
```

Validaciones de Negocio

```
□ User acceptance testing passed
□ Business process validation
□ Data migration completed
□ Training sessions conducted
□ Support documentation ready
□ Communication plan executed
□ Change management approved
□ Go/No-Go decision made
```

Go-Live Day

Deployment Sequence

```
Hour 0: Deploy to production
Hour 1: Smoke tests
Hour 2: Enable for pilot users (10%)
Hour 4: Monitor and validate
Hour 6: Scale to 50% if successful
```

```
Hour 12: Full deployment if stable
Hour 24: Post-deployment review
```

Monitoring Checklist

```
□ System health dashboard green
□ All integrations responding
□ Error rates within acceptable limits
□ Performance metrics normal
□ User feedback positive
□ No critical issues reported
□ Support team ready
□ Escalation procedures active
```

Post-Go-Live (1 semana después)

Validation Activities

```
□ Review all metrics and KPIs
□ Collect user feedback
□ Document lessons learned
□ Optimize based on real usage
□ Plan next iteration
□ Update documentation
□ Schedule regular reviews
□ Celebrate success! 🎉
```

📚 Recursos Adicionales

Documentación Técnica

- **Arquitectura General**: Diagramas y patrones implementados
- **Modelo de Datos**: Objetos, relaciones y estructura
- **APIs y Endpoints**: Documentación completa de servicios
- **Catálogo de Clases**: Todas las clases documentadas
- **Alcance y Limitaciones**: Cobertura del framework

Herramientas de Desarrollo

```
- Salesforce CLI (sfdx)
- VS Code con Salesforce Extensions
- Postman Collection para APIs
- Workbench para administración
- Data Loader para migraciones
```

Contactos de Soporte

```
Arquitectura: Soulberto Lorenzo <soulberto@cloudblue.us>
Desarrollo: Jean Carlos Melendez <jean@cloudblue.us>
Soporte: foundation-support@sura.com
Documentación: docs@sura-foundation.com
```

Fecha de Actualización: Mayo 2025**Versión de Implementación:** 1.0**Mantenido por:** Equipo de Implementación SURA Foundation**Estado:** Documento Vivo - Actualización Continua