API Documentation - Simple Smells Detector

Esta documentación describe la API interna del plugin Simple Smells Detector y cómo extender su funcionalidad.

Tabla de Contenidos

Arquitectura de la API

Flujo de Comunicación

```
UI (ui.html) ↔ Plugin Core (code.js) ↔ Analysis Engine
```

Mensajes de Comunicación

UI → Plugin Core

```
// Mensaje de análisis completo
parent.postMessage({
  pluginMessage: {
    type: 'analyze-all',
    scope: 'current-page' | 'entire-file',
    settings: { /* configuraciones personalizadas */ }
  }
}, '*');

// Mensaje de análisis específico
parent.postMessage({
  pluginMessage: {
    type: 'analyze-specific',
    detector: 'sizeDetector',
    scope: 'current-page',
    settings: {}
  }
}, '*');

// Mensaje de configuración
```

```
parent.postMessage({
  pluginMessage: {
    type: 'save-settings',
    settings: { UMBRAL_CAMPOS_FORMULARIO: 10 }
  }
}, '*');
```

Plugin Core → UI

```
// Resultados de análisis
figma.ui.postMessage({
  type: 'analysis-results',
  results: findings,
  metadata: {
    totalNodes: 1234,
    analysisTime: 1500,
    detectors: ['size', 'consistency']
  }
});

// Progreso de análisis
figma.ui.postMessage({
  type: 'analysis-progress',
  progress: 65,
  currentDetector: 'consistencyDetector'
});

// Error
figma.ui.postMessage({
  type: 'error',
  message: 'Error durante el análisis',
  details: { /* ... */ }
});
```

Core APIs

Main Analysis Function

```
/**
 * Función principal de análisis
 * @param {string} type - Tipo de análisis ('all' | 'specific')
 * @param {Object} options - Opciones de configuración
 * @returns {Promise<Array<Finding>>} Resultados del análisis
 */
async function runAnalysis(type, options = {}) {
  const {
    scope = 'current-page',
```

```
    detectors = 'all',
    settings = {}
  } = options;

  // Obtener nodos según scope
  const nodes = await getNodesForAnalysis(scope);

  // Ejecutar detectores
  const findings = await executeDetectors(nodes, detectors, settings);

  // Enriquecer con metadatos
  return findings.map(finding => withFrameInfo(finding));
}
```

Node Extraction

```
/**
 * Obtiene nodos para análisis según el scope
 * @param {string} scope - Alcance del análisis
 * @returns {Promise<Array<SceneNode>>} Nodos filtrados
 */
async function getNodesForAnalysis(scope) {
  let nodes = [];

  switch (scope) {
    case 'current-page':
      nodes = figma.currentPage.findAll();
      break;
    case 'selection':
      nodes = figma.currentPage.selection;
      break;
    case 'entire-file':
      nodes = figma.root.findAll();
      break;
  }

  return filterRelevantNodes(nodes);
}
```

Finding Structure

```
/**
 * Estructura estándar de un hallazgo
 */
interface Finding {
  // Identificación
  id: string;
  type: SmellType;
```

```
  detector: string;

  // Nodo afectado
  node: SceneNode;
  nodeId: string;
  nodeName: string;

  // Clasificación
  severity: 'low' | 'medium' | 'high';
  category: 'semantic' | 'consistency' | 'format' | 'structure';

  // Descripción
  message: string;
  suggestion?: string;

  // Contexto
  frameInfo?: FrameInfo;
  metadata?: Object;

  // Timestamp
  timestamp: number;
}
```

Detector Interface

Base Detector Class

```
/**
 * Clase base para todos los detectores
 */
class BaseDetector {
  constructor(settings = {}) {
    this.settings = this.mergeSettings(settings);
    this.metadata = this.getMetadata();
  }

  /**
   * Método principal de análisis - DEBE ser implementado
   * @param {Array<SceneNode>} nodes - Nodos a analizar
   * @param {Object} context - Contexto adicional
   * @returns {Promise<Array<Finding>>} Hallazgos detectados
   */
  async analyze(nodes, context = {}) {
    throw new Error('analyze() method must be implemented');
  }

  /**
   * Metadatos del detector - DEBE ser implementado
   * @returns {DetectorMetadata} Información del detector
   */
```

```
  getMetadata() {
    throw new Error('getMetadata() method must be implemented');
  }

  /**
   * Configuración por defecto
   * @returns {Object} Settings por defecto
   */
  getDefaultSettings() {
    return {};
  }

  /**
   * Merge de configuraciones
   * @param {Object} userSettings - Configuración del usuario
   * @returns {Object} Configuración final
   */
  mergeSettings(userSettings) {
    return {
      ...this.getDefaultSettings(),
      ...userSettings
    };
  }
}
```

Detector Metadata

```
interface DetectorMetadata {
  // Identificación
  id: string;
  name: string;
  version: string;

  // Descripción
  description: string;
  category: DetectorCategory;

  // Configuración
  smellType: SmellType;
  defaultSettings: Object;

  // Información adicional
  author?: string;
  documentation?: string;
  examples?: Array<string>;
}
```

Ejemplo de Implementación

```
/**
 * Detector de tamaños inadecuados de campos
 */
class SizeDetector extends BaseDetector {
  getDefaultSettings() {
    return {
      dataTypes: TIPO_DE_DATOS,
      strictMode: false
    };
  }

  async analyze(nodes, context = {}) {
    const findings = [];
    const inputNodes = this.filterInputNodes(nodes);

    for (const node of inputNodes) {
      const finding = await this.analyzeNode(node, context);
      if (finding) {
        findings.push(finding);
      }
    }

    return findings;
  }

  async analyzeNode(node, context) {
    // Encontrar texto asociado
    const associatedText = encontrarTextoAsociado(node);
    if (!associatedText) return null;

    // Identificar tipo de dato
    const dataType = identificarTipoDeDato(associatedText);
    if (!dataType) return null;

    // Verificar tamaño
    const isInappropriate = this.checkSizeAppropriateness(node, dataType);
    if (!isInappropriate) return null;

    return {
      id: generateFindingId(),
      type: 'INAPPROPRIATE_SIZE',
      detector: 'sizeDetector',
      node,
      nodeId: node.id,
      nodeName: node.name,
      severity: this.calculateSeverity(node, dataType),
      category: 'semantic',
      message: `Campo de ${dataType.name} con ancho ${node.width}px
${dataType.mensaje}`,
      suggestion: `Considera ajustar el ancho entre
${dataType.minWidth}-${dataType.maxWidth}px`,
      metadata: {
```

```
        currentWidth: node.width,
        suggestedRange: [dataType.minWidth, dataType.maxWidth],
        dataType: dataType.name
      },
      timestamp: Date.now()
    };
  }

  getMetadata() {
    return {
      id: 'size-detector',
      name: 'Size Detector',
      version: '1.0.0',
      description: 'Detecta campos con ancho inadecuado para su tipo de
contenido',
      category: 'semantic',
      smellType: 'INAPPROPRIATE_SIZE',
      author: 'Simple Smells Detector Team'
    };
  }
}
```

Utilidades

Geometry Utils

```
/**
 * Utilidades de cálculos geométricos
 */
const GeometryUtils = {
  /**
   * Calcula la distancia entre dos nodos
   * @param {SceneNode} node1
   * @param {SceneNode} node2
   * @returns {number} Distancia euclidiana
   */
  distance(node1, node2) {
    const dx = node1.x - node2.x;
    const dy = node1.y - node2.y;
    return Math.sqrt(dx * dx + dy * dy);
  },

  /**
   * Verifica si dos nodos están en la misma fila horizontal
   * @param {SceneNode} node1
   * @param {SceneNode} node2
   * @param {number} tolerance
   * @returns {boolean}
   */
  areInSameRow(node1, node2, tolerance = 10) {
```

```
    return Math.abs(node1.y - node2.y) <= tolerance;
  },

  /**
   * Agrupa nodos por proximidad
   * @param {Array<SceneNode>} nodes
   * @param {number} maxDistance
   * @returns {Array<Array<SceneNode>>} Grupos de nodos
   */
  groupByProximity(nodes, maxDistance = 100) {
    // Implementación de clustering por proximidad
  }
};
```

Semantic Utils

```
/**
 * Utilidades de análisis semántico
 */
const SemanticUtils = {
  /**
   * Normaliza texto para análisis
   * @param {string} text
   * @returns {string} Texto normalizado
   */
  normalizeText(text) {
    return text
      .toLowerCase()
      .normalize('NFD')
      .replace(/[\u0300-\u036f]/g, '')
      .trim();
  },

  /**
   * Encuentra el texto asociado a un nodo
   * @param {SceneNode} node
   * @param {number} searchRadius
   * @returns {string|null} Texto encontrado
   */
  findAssociatedText(node, searchRadius = 100) {
    // Lógica de búsqueda de texto asociado
  },

  /**
   * Clasifica el tipo de dato basándose en el texto
   * @param {string} text
   * @returns {DataType|null} Tipo de dato identificado
   */
  identifyDataType(text) {
    const normalizedText = this.normalizeText(text);
```

```
    for (const [typeName, typeConfig] of Object.entries(TIPO_DE_DATOS)) {
      for (const keyword of typeConfig.keywords) {
        if (normalizedText.includes(keyword)) {
          return { name: typeName, ...typeConfig };
        }
      }
    }

    return null;
  }
};
```

Node Filters

```
/**
 * Filtros de nodos especializados
 */
const NodeFilters = {
  /**
   * Filtra nodos que pueden ser campos de entrada
   * @param {Array<SceneNode>} nodes
   * @returns {Array<SceneNode>} Nodos de entrada
   */
  getInputNodes(nodes) {
    return nodes.filter(node => {
      // Filtros por tipo, nombre, y características
      if (node.type === 'RECTANGLE' || node.type === 'FRAME') {
        return this.looksLikeInput(node);
      }
      return false;
    });
  },

  /**
   * Verifica si un nodo parece un campo de entrada
   * @param {SceneNode} node
   * @returns {boolean}
   */
  looksLikeInput(node) {
    // Heurísticas para identificar inputs
    const hasInputKeywords = /input|field|text|entrada|campo/i.test(node.name);
    const hasAppropriateSize = node.width > 50 && node.height > 20 &&
node.height < 80;
    const isVisible = node.visible && node.opacity > 0;

    return hasInputKeywords && hasAppropriateSize && isVisible;
  },

  /**
```

```
   * Filtra nodos de texto
   * @param {Array<SceneNode>} nodes
   * @returns {Array<TextNode>} Nodos de texto
   */
  getTextNodes(nodes) {
    return nodes.filter(node => node.type === 'TEXT');
  }
};
```

Sistema de Configuración

Settings Management

```
/**
 * Gestor de configuraciones
 */
class SettingsManager {
  constructor() {
    this.defaultSettings = DEFAULT_SETTINGS;
    this.currentSettings = {};
  }

  /**
   * Carga configuraciones desde clientStorage
   * @returns {Promise<Object>} Configuraciones cargadas
   */
  async loadSettings() {
    try {
      const stored = await figma.clientStorage.getAsync('settings');
      this.currentSettings = { ...this.defaultSettings, ...stored };
      return this.currentSettings;
    } catch (error) {
      console.warn('Error loading settings:', error);
      this.currentSettings = { ...this.defaultSettings };
      return this.currentSettings;
    }
  }

  /**
   * Guarda configuraciones en clientStorage
   * @param {Object} settings - Configuraciones a guardar
   * @returns {Promise<void>}
   */
  async saveSettings(settings) {
    this.currentSettings = { ...this.currentSettings, ...settings };
    await figma.clientStorage.setAsync('settings', this.currentSettings);
  }

  /**
   * Obtiene configuración específica
```

```
   * @param {string} key - Clave de configuración
   * @param {*} defaultValue - Valor por defecto
   * @returns {*} Valor de configuración
   */
  get(key, defaultValue = null) {
    return this.currentSettings[key] ?? this.defaultSettings[key] ??
defaultValue;
  }

  /**
   * Resetea configuraciones a valores por defecto
   * @returns {Promise<void>}
   */
  async resetToDefaults() {
    this.currentSettings = { ...this.defaultSettings };
    await this.saveSettings(this.currentSettings);
  }
}
```

Custom Data Types

```
/**
 * Gestor de tipos de datos personalizados
 */
class DataTypeManager {
  constructor() {
    this.customTypes = new Map();
  }

  /**
   * Registra un nuevo tipo de dato
   * @param {string} id - Identificador del tipo
   * @param {DataTypeConfig} config - Configuración del tipo
   */
  registerDataType(id, config) {
    const dataType = {
      id,
      ...config,
      isCustom: true,
      createdAt: Date.now()
    };

    this.customTypes.set(id, dataType);
    this.saveCustomTypes();
  }

  /**
   * Obtiene todos los tipos (built-in + custom)
   * @returns {Object} Todos los tipos de datos
   */
```

```
  getAllDataTypes() {
    const allTypes = { ...TIPO_DE_DATOS };

    for (const [id, type] of this.customTypes) {
      allTypes[id] = type;
    }

    return allTypes;
  }

  /**
   * Elimina un tipo personalizado
   * @param {string} id - ID del tipo a eliminar
   */
  removeCustomDataType(id) {
    this.customTypes.delete(id);
    this.saveCustomTypes();
  }

  async saveCustomTypes() {
    const types = Object.fromEntries(this.customTypes);
    await figma.clientStorage.setAsync('customDataTypes', types);
  }

  async loadCustomTypes() {
    try {
      const stored = await figma.clientStorage.getAsync('customDataTypes');
      if (stored) {
        this.customTypes = new Map(Object.entries(stored));
      }
    } catch (error) {
      console.warn('Error loading custom data types:', error);
    }
  }
}
```

Extensión del Sistema

Registrar Nuevos Detectores

```
/**
 * Registro de detectores
 */
class DetectorRegistry {
  constructor() {
    this.detectors = new Map();
    this.loadBuiltInDetectors();
  }

  /**
```

```
   * Registra un nuevo detector
   * @param {BaseDetector} detector - Instancia del detector
   */
  register(detector) {
    const metadata = detector.getMetadata();
    this.detectors.set(metadata.id, detector);
  }

  /**
   * Obtiene un detector por ID
   * @param {string} id - ID del detector
   * @returns {BaseDetector|null} Detector encontrado
   */
  get(id) {
    return this.detectors.get(id) || null;
  }

  /**
   * Obtiene todos los detectores registrados
   * @returns {Array<BaseDetector>} Lista de detectores
   */
  getAll() {
    return Array.from(this.detectors.values());
  }

  /**
   * Filtra detectores por categoría
   * @param {string} category - Categoría a filtrar
   * @returns {Array<BaseDetector>} Detectores filtrados
   */
  getByCategory(category) {
    return this.getAll().filter(detector => {
      const metadata = detector.getMetadata();
      return metadata.category === category;
    });
  }

  loadBuiltInDetectors() {
    // Cargar detectores incorporados
    this.register(new SizeDetector());
    this.register(new ConsistencyDetector());
    this.register(new FormatDetector());
    // ... otros detectores
  }
}
```

Plugin API para Terceros

```
/**
 * API pública para extensiones de terceros
```

```
 */
const SimpleSmellsAPI = {
  /**
   * Versión de la API
   */
  version: '1.0.0',

  /**
   * Registra un detector personalizado
   * @param {BaseDetector} detector - Detector a registrar
   */
  registerDetector(detector) {
    if (!(detector instanceof BaseDetector)) {
      throw new Error('Detector must extend BaseDetector');
    }

    detectorRegistry.register(detector);
  },

  /**
   * Registra un tipo de dato personalizado
   * @param {string} id - ID del tipo
   * @param {DataTypeConfig} config - Configuración
   */
  registerDataType(id, config) {
    dataTypeManager.registerDataType(id, config);
  },

  /**
   * Acceso a utilidades
   */
  utils: {
    geometry: GeometryUtils,
    semantic: SemanticUtils,
    filters: NodeFilters
  },

  /**
   * Ejecuta análisis programático
   * @param {Object} options - Opciones de análisis
   * @returns {Promise<Array<Finding>>} Resultados
   */
  async runAnalysis(options = {}) {
    return await runAnalysis('all', options);
  }
};

// Exponer API globalmente para plugins
if (typeof globalThis !== 'undefined') {
  globalThis.SimpleSmellsAPI = SimpleSmellsAPI;
}
```

Ejemplos de Uso

Crear un Detector Personalizado

```
/**
 * Ejemplo: Detector de botones sin etiquetas descriptivas
 */
class ButtonLabelDetector extends BaseDetector {
  getDefaultSettings() {
    return {
      minLabelLength: 3,
      bannedWords: ['click', 'button', 'btn']
    };
  }

  async analyze(nodes, context = {}) {
    const findings = [];
    const buttonNodes = this.filterButtonNodes(nodes);

    for (const node of buttonNodes) {
      const label = this.extractButtonLabel(node);

      if (this.hasProblematicLabel(label)) {
        findings.push({
          id: `button-label-${node.id}`,
          type: 'UNCLEAR_BUTTON_LABEL',
          detector: 'buttonLabelDetector',
          node,
          nodeId: node.id,
          nodeName: node.name,
          severity: 'medium',
          category: 'semantic',
          message: `Botón con etiqueta poco descriptiva: "${label}"`,
          suggestion: 'Usa etiquetas que describan claramente la acción del
botón',
          metadata: { label },
          timestamp: Date.now()
        });
      }
    }

    return findings;
  }

  filterButtonNodes(nodes) {
    return nodes.filter(node =>
      /button|btn|boton/i.test(node.name) ||
      this.looksLikeButton(node)
    );
  }
```

```
  hasProblematicLabel(label) {
    if (!label || label.length < this.settings.minLabelLength) {
      return true;
    }

    return this.settings.bannedWords.some(word =>
      label.toLowerCase().includes(word)
    );
  }

  getMetadata() {
    return {
      id: 'button-label-detector',
      name: 'Button Label Detector',
      version: '1.0.0',
      description: 'Detecta botones con etiquetas poco descriptivas',
      category: 'semantic',
      smellType: 'UNCLEAR_BUTTON_LABEL'
    };
  }
}

// Registrar el detector
SimpleSmellsAPI.registerDetector(new ButtonLabelDetector());
```

Crear Tipo de Dato Personalizado

```
// Registrar un nuevo tipo de dato
SimpleSmellsAPI.registerDataType('IBAN', {
  keywords: ['iban', 'cuenta bancaria', 'número de cuenta'],
  minWidth: 250,
  maxWidth: 400,
  mensaje: 'Ancho inadecuado para un IBAN',
  requiresComponent: true,
  componentSmellMessage: 'Este campo IBAN debería usar validación específica',
  validation: {
    pattern: /^[A-Z]{2}[0-9]{22}$/,
    example: 'ES9121000418450200051332'
  }
});
```

Análisis Programático

```
// Ejecutar análisis específico
const findings = await SimpleSmellsAPI.runAnalysis({
  scope: 'current-page',
  detectors: ['size-detector', 'button-label-detector'],
  settings: {
```

```
    UMBRAL_CAMPOS_FORMULARIO: 6
  }
});

console.log(`Encontrados ${findings.length} problemas de usabilidad`);
```

Ver también:

- Arquitectura - Visión general del sistema
- Detectores - Especificación de detectores
- Desarrollo - Guía de desarrollo