Meli Challenge: Clasificador de datos sensibles en bases de datos MySQL

Este repositorio contiene una solución para la prueba técnica "Clasificación de base de datos". La aplicación explora instancias MySQL externas, recorre esquemas y tablas, clasifica columnas según reglas configurables (regex) y persiste resultados y un historial de ejecuciones.

Solución: Clasificación de base de datos

Esta solución explora instancias MySQL externas, recorre esquemas y tablas, y clasifica columnas según reglas configurables (regex) almacenadas en la base de datos. Los resultados y el historial de ejecuciones se persisten para trazabilidad y auditoría.

Diseño

La solución está pensada para entornos donde la trazabilidad, la auditoría y la consistencia son fundamentales. Por eso se eligió una base de datos relacional (MySQL), que permite modelar relaciones entre configuraciones, ejecuciones y resultados de escaneo mediante claves foráneas y consultas estructuradas. Esto facilita la generación de reportes, el seguimiento histórico y la integración con sistemas corporativos de cumplimiento y seguridad.

El sistema es extensible: las reglas de clasificación se gestionan dinámicamente desde la base de datos, permitiendo agregar nuevos tipos de datos sensibles sin modificar el código. La arquitectura sigue principios SOLID, separando responsabilidades en capas (controladores, servicios, repositorios y clasificadores), lo que mejora la mantenibilidad y la testabilidad. Los patrones Repository y Factory aseguran que la lógica de negocio no dependa de detalles de almacenamiento ni de la implementación de los clasificadores.

Además, la solución es robusta ante errores: cada ejecución de escaneo queda registrada con su estado, permitiendo identificar fallos y mantener la trazabilidad. El diseño está alineado con prácticas de desarrollo profesional y preparado para evolucionar, integrarse en pipelines de auditoría y adaptarse a nuevos requisitos de negocio o seguridad.

PROF

Componentes principales

El proyecto está organizado en capas para facilitar la mantenibilidad y la claridad:

- **Controladores**: gestionan las peticiones HTTP y exponen los endpoints de la API.
- **Servicios**: contienen la lógica de negocio, orquestan el flujo de escaneo y clasificación.
- **Repositorios**: manejan el acceso a la base de datos y la persistencia de información.
- **Clasificadores**: implementan la lógica para identificar tipos de datos sensibles usando reglas dinámicas.

Cómo ejecutar el proyecto

1. Crear el archivo `.env` en la raíz del proyecto con las variables de entorno necesarias.
2. Ejecutar los servicios con Docker Compose:

```
docker-compose up --build -d
```

Una vez levantado, se pueden realizar peticiones a los endpoints según el puerto definido en `.env`. Por ejemplo http://localhost:8000.

Autenticación por medio de API key

La aplicación soporta un mecanismo sencillo de API key. Es necesario definir la variable `API_KEY` en el archivo `.env`.

Incluir el header `X-API-Key: <your-key>` en las peticiones a los endpoints protegidos. Si `API_KEY` no está definido, la autenticación se omite (modo desarrollo).

Ejemplo de header:

```
X-API-Key: mysecretkey
```

Endpoints principales

Registrar una base externa

POST /api/v1/database

```
curl -X POST http://localhost:8000/api/v1/database \
  -H "Content-Type: application/json" \
  -H "X-API-Key: mysecretkey" \
  -d '{
    "host": "meli-challenge-target-db",
    "port": 3309,
    "username": "target_user",
    "password": "target_password"
  }'
```

Respuesta esperada:

```
{
  "id": 1
}
```

Lanzar escaneo

PROF

POST /api/v1/database/scan/:id

```
curl -X POST http://localhost:8000/api/v1/database/scan/1 \
  -H "X-API-Key: mysecretkey"
```

Respuesta esperada:

```
{
  "scan_id": 1
}
```

Consultar resultados de escaneo

GET /api/v1/database/scan/:id

```
curl -X GET http://localhost:8000/api/v1/database/scan/1 \
  -H "X-API-Key: mysecretkey"
```

Respuesta ejemplo:

```
{
  "database": [
    {
      "schema_name": "target_sample_db",
      "schema_tables": [
        {
          "table_name": "users",
          "columns": [
            {
              "column_name": "created_at",
              "info_type": "N/A"
            },
            {
              "column_name": "first_name",
              "info_type": "FIRST_NAME"
            },
            {
              "column_name": "id",
              "info_type": "N/A"
            },
            {
              "column_name": "ip_address",
              "info_type": "IP_ADDRESS"
            },
            {
```

```
              "column_name": "last_name",
              "info_type": "LAST_NAME"
            },
            {
              "column_name": "phone",
              "info_type": "PHONE_NUMBER"
            },
            {
              "column_name": "useremail",
              "info_type": "EMAIL_ADDRESS"
            },
            {
              "column_name": "username",
              "info_type": "USERNAME"
            }
          ]
        }
      ]
    }
  ]
}
```

Tests

Los tests unitarios están implementados en Testify y cubren la lógica principal del sistema:

1. Validan la creación y actualización de registros en el historial de escaneos (`scan_history`).
2. Verifican la persistencia y agrupamiento de resultados (`scan_results`).
3. Validan que las reglas dinámicas clasifican correctamente columnas, probando casos positivos y negativos para asegurar que los clasificadores construidos desde la base de datos funcionan como se espera.
4. Utilizan mocks (`sqlmock` y repositorios simulados) para probar la lógica sin requerir una base real ni levantar contenedores.
5. Los tests se pueden ejecutar con el siguiente comando:

PROF

```
go test ./... -v
```

Logging

La aplicación incluye un logger que escribe a stdout. Controla el nivel de detalle con la variable de entorno `LOG_LEVEL` (valores: `DEBUG`, `INFO`, `WARN`, `ERROR`). Por defecto el nivel es `INFO`. Los logs aparecen en la salida estándar y contienen timestamp y nivel, por ejemplo:

```
[INFO] Scanning: target_sample_db.users
[ERROR] SaveResult exec failed for scanID=1: <error>
```

Para ver logs más verbosos, exporta `LOG_LEVEL=DEBUG` antes de levantar los servicios.

Modelo de datos

El modelo de datos y las migraciones están definidos en el archivo `init.sql`.

Resumen del modelo:

- `external_databases`: almacena las conexiones a bases externas que serán escaneadas (host, puerto, usuario, contraseña).
- `scan_history`: registra cada ejecución de escaneo, con referencia a la base, timestamp y estado (`running`, `success`, `failed`).
- `scan_results`: guarda los resultados detallados de cada escaneo, incluyendo el esquema, tabla, columna y tipo de información detectada.
- `classification_rules`: contiene las reglas de clasificación (regex y tipo), permitiendo que el sistema sea extensible y configurable sin modificar el código.

Las relaciones entre tablas permiten trazabilidad completa: cada resultado está vinculado a un escaneo y cada escaneo a una base registrada.