Manual del Programador CopernicusProyect

Contenido

Manual del Programador — *CopernicusProyect*

Este documento está dirigido a desarrolladores que deseen comprender, mantener o extender el proyecto **CopernicusProyect**. Incluye la estructura general del sistema, descripción de los módulos principales, dependencias necesarias, flujo de ejecución y pautas para contribuir correctamente.

1. Estructura del Proyecto

```
CopernicusProyect/
│
├── app.py
├── requirements.txt
├── .gitignore
│
├── Auth/
│   ├── __init__.py
│   └── auth.py
│
├── Constants/
│   ├── __init__.py
│   └── constants.py
│
├── Request/
│   ├── __init__.py
│   └── request.py
│
├── utils/
│   └── tiff_reader.py
│
└── README.md
```

2. Descripción de Módulos

app.py

Archivo principal del proyecto. Controla el flujo de ejecución, gestiona fechas, embalses y satélites, y coordina las peticiones a la API de Copernicus de forma concurrente.

- **Concurrencia:** Utiliza ThreadPoolExecutor para ejecución paralela de tareas.
- **Control de interrupciones:** Captura interrupciones mediante Ctrl+C de forma segura.
- **Funciones principales:**
  - main(): Ejecuta el bucle principal.

- process_request(): Gestiona lógica de petición, reintentos y almacenamiento de resultados o banderas.

Auth/auth.py

Encargado de la autenticación mediante OAuth2 para acceder a la API de Copernicus.

- **Clase principal:** Auth
  - Obtiene y gestiona el token de acceso utilizando las variables de entorno CLIENT_ID y CLIENT_SECRET.

Constants/constants.py

Contiene definiciones globales utilizadas en todo el proyecto.

- **Variables principales:**
  - SATELLITES, SENTINEL2_BANDS, SENTINEL3_BANDS, BANDS
  - BBOX, TIME_FROM, CLOUD_COVERAGE

Request/request.py

Construye y ejecuta las peticiones a la API de Copernicus.

- **Clase principal:** Request
  - Genera el payload y evalscript.
  - Valida la respuesta y comprueba si los archivos TIFF son válidos.
  - Utiliza is_empty_tiff_rasterio() desde tiff_reader.py.

utils/tiff_reader.py

Incluye funciones utilitarias para la gestión de archivos TIFF.

- **Funciones principales:**
  - save_bands_tiff(): Guarda cada banda en un archivo TIFF.
  - open_tiff(): Visualiza archivos TIFF.
  - create_output_path(): Genera la ruta de almacenamiento en función de fecha, embalse y satélite.
  - create_flag_file(): Crea un archivo .flag si no se detectan datos útiles.
  - is_empty_tiff_rasterio(): Determina si un TIFF está vacío o dañado.

3. Flujo de Ejecución

1. **Inicio de ejecución:**
    - Se lanza app.py.
    - Se inicializan las fechas, embalses y satélites.
2. **Iteración por fecha:**
    - Se generan tareas concurrentes para cada combinación de satélite y embalse.
    - Cada tarea invoca process_request():
        - Verifica la existencia previa de archivos TIFF o .flag.
        - Si no existen, se realiza la petición.
        - Si la respuesta es válida, se almacenan los TIFF.
        - En caso contrario, se crea un archivo .flag.
3. **Gestión de errores:**
    - Se implementan reintentos exponenciales ante fallos temporales.
    - Se manejan excepciones de red y errores HTTP.
    - Se soporta la interrupción del programa mediante señal de teclado.

4. Dependencias

Las dependencias del proyecto se encuentran definidas en requirements.txt.

Principales paquetes:

- requests, requests-oauthlib, oauthlib: Gestión de autenticación y peticiones HTTP.
- rasterio, numpy, matplotlib, Pillow: Manipulación y visualización de archivos TIFF.
- python-dotenv: Carga de variables de entorno desde archivos .env.
- Otros: affine, attrs, click, entre otros.

Instalación:

```
pip install -r requirements.txt
```

5. Variables de Entorno

Se requiere un archivo .env con las siguientes variables mínimas:

```
CLIENT_ID=...
CLIENT_SECRET=...
OUTPUT_DIR=output
```

6. Buenas Prácticas

- No modificar las constantes directamente en el código fuente. Utilizar constants.py.
- No almacenar archivos TIFF ni .flag en el repositorio. El archivo .gitignore está configurado para ignorarlos.
- Para añadir nuevos embalses o satélites:
  - Definir sus valores en constants.py.
  - Añadir las bandas necesarias en la estructura correspondiente.
- Para análisis o depuración:
  - Revisar mensajes de consola incluidos en el flujo.
  - Archivos TIFF corruptos se almacenan como debug_corrupt_tiff_S3.bin.
- Para extender la funcionalidad:
  - Nuevas funciones de análisis pueden añadirse en tiff_reader.py.
  - Nuevos productos de Copernicus deben configurarse en constants.py y request.py.

7. Ejecución del Proyecto

El proyecto puede ejecutarse mediante:

```
python app.py
```

8. Posibles Mejoras

A continuación se sugieren algunas mejoras y extensiones que pueden implementarse en el proyecto para aumentar su robustez, eficiencia y funcionalidad:

Gestión avanzada de errores

Implementar un sistema de logs estructurados (por ejemplo, usando la librería logging) para registrar errores, advertencias y eventos importantes en archivos separados.

Añadir notificaciones automáticas (correo, Slack, etc.) en caso de fallos críticos o repetidos.

Optimización del rendimiento

Permitir la configuración dinámica del número de hilos o procesos para aprovechar mejor los recursos del sistema.

Implementar procesamiento asíncrono usando asyncio para reducir tiempos de espera en operaciones de red.

Interfaz de usuario

Desarrollar una interfaz gráfica (GUI) o una interfaz web sencilla para seleccionar fechas, embalses y satélites, y visualizar el progreso de las descargas.

Añadir una barra de progreso en consola para mejorar la experiencia del usuario.

Validación y calidad de datos

Implementar validaciones adicionales sobre los archivos TIFF descargados (por ejemplo, comprobación de valores esperados en las bandas).

Añadir un sistema de checksum o hash para verificar la integridad de los archivos descargados.

Automatización y despliegue

Crear scripts para la ejecución automática periódica (por ejemplo, usando cron o tareas programadas de Windows).

Preparar un contenedor Docker para facilitar la instalación y despliegue en diferentes entornos.

Soporte para nuevos productos y fuentes

Añadir soporte para otros productos de Copernicus o fuentes de datos satelitales.

Permitir la configuración de nuevas bandas o parámetros desde archivos externos (por ejemplo, JSON o YAML).

Documentación y pruebas

Ampliar la documentación técnica y de usuario.

Añadir pruebas unitarias y de integración para los módulos principales, usando frameworks como pytest.

Escalabilidad

Adaptar el sistema para funcionar en la nube (por ejemplo, AWS Lambda, Azure Functions) y procesar grandes volúmenes de datos de forma distribuida.

9. Recursos Útiles

Documentación oficial de Copernicus
Documentación de rasterio
OAuth2 para Python

10. Errores Conocidos

1.Problemas de autenticación
Si las variables de entorno CLIENT_ID o CLIENT_SECRET son incorrectas o están ausentes, la autenticación fallará y no se podrán realizar peticiones a la API de Copernicus.
En ocasiones, el token puede expirar antes de lo esperado, requiriendo una nueva autenticación manual.

2. Errores de red o conexión
Si la conexión a internet es inestable, pueden producirse errores de timeout o desconexión durante las descargas.
La API de Copernicus puede estar temporalmente no disponible, lo que genera errores HTTP 5xx.

3. Archivos TIFF corruptos o vacíos
En algunos casos, la API puede devolver archivos TIFF vacíos o corruptos, especialmente cuando no hay datos disponibles para la consulta solicitada.
Estos archivos se detectan y se marcan con un archivo .flag, pero pueden quedar archivos binarios de depuración en la carpeta de salida.

4. Problemas de compatibilidad en Windows
Algunas dependencias como rasterio pueden requerir instalación adicional de librerías del sistema (por ejemplo, GDAL). Si faltan, pueden aparecer errores al importar o manipular archivos TIFF.

5. Limitaciones de concurrencia
Si se lanzan demasiadas tareas concurrentes, se puede alcanzar el límite de conexiones del sistema o de la API, provocando errores de "Too Many Requests" (HTTP 429) o bloqueos temporales. Sin embargo esto ya está controlado y si devuelve un 429 este volverá a hacer una serie de intentos hasta que se realice con éxito.

6. Errores en la configuración de rutas
Si la variable OUTPUT_DIR no está bien definida o la ruta no existe, el programa puede fallar al intentar guardar los archivos descargados.

7. Falta de manejo de excepciones específicas
Algunos errores poco frecuentes pueden no estar completamente gestionados y provocar la detención del programa. Se recomienda revisar y ampliar el manejo de excepciones según se detecten nuevos casos.

11. Preguntas Frecuentes (FAQ)

¿Qué hago si la autenticación falla?
Verifica que las variables de entorno CLIENT_ID y CLIENT_SECRET estén correctamente configuradas en el archivo .env. Si el problema persiste, revisa la conexión a internet y que las credenciales no hayan expirado.

¿Por qué aparecen archivos .flag en las carpetas de salida?
Los archivos .flag indican que para esa combinación de fecha, embalse y satélite no se encontraron datos útiles o la descarga falló repetidamente. Esto evita que el sistema intente descargar los mismos datos vacíos en futuras ejecuciones.

¿Cómo agrego un nuevo embalse o satélite?
Edita el archivo constants.py y añade el nuevo embalse (con su bounding box) o el nuevo satélite en las listas correspondientes. Asegúrate de incluir las bandas necesarias si el satélite es nuevo.

¿Dónde se guardan los archivos descargados?
Los archivos TIFF y .flag se guardan en la carpeta definida por la variable de entorno OUTPUT_DIR, organizada por fecha, embalse y satélite.

¿Cómo puedo visualizar los archivos TIFF descargados?
Puedes usar la función open_tiff() del módulo tiff_reader.py o cualquier visor de imágenes geoespaciales compatible con TIFF, como QGIS.

¿Qué hago si el programa se detiene inesperadamente?
Revisa los mensajes de error en la consola. Si fue por una interrupción manual (Ctrl+C), simplemente vuelve a ejecutar el programa; las tareas ya completadas no se repetirán gracias a los archivos .flag y TIFF existentes.

¿Puedo ejecutar el programa en otro sistema operativo?
Sí, el proyecto es multiplataforma siempre que tengas Python y las dependencias instaladas. Si usas Windows, asegúrate de instalar correctamente las librerías de geoprocesamiento como rasterio.