Monitoreo de la API

Table of contents

1 Métricas en memoria **1**

2 Publicador en segundo plano **2**
2.1 Flujo resumido 2
2.2 Configuración 2

3 Cache de respuestas **2**

4 Logs y auditoría **2**

5 Cobertura DIPRES (status page) **3**

6 Próximos pasos sugeridos **3**

1 Métricas en memoria

El middleware `metrics_middleware` (ver `api/main.py`) registra:

- Conteo de códigos HTTP.
- Últimas 50 solicitudes (path, método, timestamp, bytes enviados).
- Metadatos del último payload (`payload_meta`).
- Versión semver + commit (`meta.version`, `meta.git_commit`) cuando se consulta `/status`.

Se expone vía `GET /api/0.6.9/status`.

```
{
  "status_codes": {"200": 42, "404": 3},
  "recent_requests": [
    {"path": "/api/0.6.9/datasets", "status_code": 200, ...}
  ],
  "last_payload": {"collection": "datasets", "resource": "dim_region", "returned": 25, "timesta
}
```

2 Publicador en segundo plano

- `api.metrics.storage.APIMetrics` mantiene la ventana de peticiones recientes y los contadores agregados. El middleware registra cada request con `metrics.record(...)`.
- `api.metrics.publisher.MetricsPublisher` toma esos snapshots y, cada `interval_seconds`, invoca `publish_once` para persistir los percentiles, RPS y tasa de error en `monitor_api_metrics`.
- `start_metrics_publisher(engine, interval_seconds=60, window_seconds=300, max_retries=3, retry_backoff=1.0)` inicializa un hilo `daemon`. Los tiempos de espera entre reintentos siguen el patrón `retry_backoff * intento`.
- Ante errores de base de datos (`SQLAlchemyError`) se reintenta hasta `max_retries` veces y se registra un `warning`; si se agotan los intentos, se continúa el ciclo sin interrumpir la API.
- El método `publish_once` puede usarse en tareas programadas (cron) para correr el mismo flujo en ejecución síncrona.

2.1 Flujo resumido

1. Middleware → `APIMetrics.record`: acumula datos crudos.
2. Hilo `MetricsPublisher` → `collect_route_stats`: agrupa por ruta/método y calcula percentiles (p50, p95, p99) y error rate.
3. `_flush` → INSERT en `monitor_api_metrics` calculando RPS (`sample_size / interval_seconds`).

2.2 Configuración

- Variables de entorno sugeridas: `API_METRICS_INTERVAL_SECONDS`, `API_METRICS_WINDOW_SECONDS`, `API_METRICS_RETRY_BACKOFF`.
- Se recomienda fijar `history_size` de `APIMetrics` según el tráfico estimado (por defecto 50).
- Pruebas unitarias: `tests/unit/test_api_metrics.py` cubre `APIMetrics`, `collect_route_stats` y los reintentos del publicador con un engine simulado.

3 Cache de respuestas

- Implementado en `api/caching.py` (TTL configurable con `API_CACHE_TTL_SECONDS`).
- Decorador `@cached_response` envuelve endpoints idempotentes.
- Limpiar cache: reiniciar el servicio (`systemctl restart illanes00-ep`).

4 Logs y auditoría

- `etl_log` almacena ejecuciones de pipelines y puede usarse para monitoreo a largo plazo.
- Los accesos a la API quedan en `journalctl -u illanes00-ep` y en la métrica en memoria.

5 Cobertura DIPRES (status page)

- El endpoint `/api/0.6.9/status` expone `dipres_coverage` con el inventario de archivos descargados y la lista `pending_downloads` (meses/trimestres faltantes, datasets pendientes).
- El script `PYTHONPATH=. python scripts/check_dipres_coverage.py` imprime el mismo resumen en consola.
- Integración en la landing: sección "Estado de datos" enlaza al JSON de status y muestra los faltantes destacados (hasta 4 años recientes + resumen de pendientes).

6 Próximos pasos sugeridos

1. Exportar métricas a Prometheus (adapter constante).
2. Persistir `status_codes` en Postgres cada X minutos.
3. Alertar si se detecta incremento de `5xx` en la ventana configurable.