

Nivel de acabado del microservicio analytics-and-dashboards

1. Analytics and Dashboards

- **Pareja:** Rafael Pulido Cifuentes y Daniel Ruiz López

2. Nivel de acabado

Nivel objetivo: 10 - Se opta a máxima puntuación porque el microservicio implementa:

MICROSERVICIO BÁSICO QUE GESTIONE UN RECURSO

- El backend debe ser una API REST tal como se ha visto en clase implementando al menos los métodos GET, POST, PUT y DELETE y devolviendo un conjunto de códigos de estado adecuado: **REALIZADO**.
 - Tenemos hechos tres CRUD al completo (para `dashboards`, `widgets` y `beat-metrics`) y tenemos algunos métodos para integraciones externas (`quotable` y `translator`). Todo el código se encuentra en los archivos de las carpetas `app/endpoints` y `app/services`.
- La API debe tener un mecanismo de autenticación: **REALIZADO**.
 - La autenticación se encuentra en `app/middleware/authentication.py`. En la API-Gateway se valida la sesión del usuario y se le introducen unas cabeceras especiales y en nuestro microservicio se hacen comprobaciones sobre esas cabeceras (`x-user-id`, etc). La autenticación se hace con JWT validado en gateway.
- Debe tener un frontend que permita hacer todas las operaciones de la API (este frontend puede ser individual o estar integrado con el resto de frontends): **REALIZADO**.
 - El frontend se encuentra integrado con el resto de microservicios en un repositorio de frontend común. Las carpetas que contienen todos los archivos referidos con nuestro microservicio son: `src/services/analytics/` para la lógica de peticiones y `src/components/Dashboard/` para la visualización de widgets y paneles. Estos componentes se utilizan en las pantallas dedicadas que se encuentran en `src/pages/app/dashboards/`
- Debe estar desplegado y ser accesible desde la nube (ya sea de forma individual o como parte de la aplicación): **REALIZADO**.
 - Se ha desplegado el microservicio en un clúster de Kubernetes de Digital Ocean y se ha usado Ionos para el DNS. Está explicado en el documento de nivel de acabado de la aplicación.
- La API que gestione el recurso también debe ser accesible en una dirección bien versionada: **REALIZADO**.
 - Todos nuestros endpoints están disponibles a través de la api-gateway de nuestra aplicación, y están correctamente versionados bajo el prefijo de `/api/v1`. El prefijo se puede ver en `main.py` y los routers en `app/endpoints/`.

- Se debe tener una documentación de todas las operaciones de la API incluyendo las posibles peticiones y las respuestas recibidas: **REALIZADO**.
 - La aplicación expone `/docs` y `/redoc` generados automáticamente por FastAPI (OpenAPI/Swagger) donde aparecen definidas todas las operaciones, esquemas de Pydantic y respuestas.
- Debe tener persistencia utilizando *MongoDB* u otra base de datos no SQL: **REALIZADO**.
 - Tenemos la conexión a la base de datos en el archivo `app/database/config.py` usando el driver asíncrono `motor.motor_asyncio`.
- Deben validarse los datos antes de almacenarlos en la base de datos (por ejemplo, haciendo uso de *mongoose*): **REALIZADO**.
 - Usamos **Pydantic** para la validación de datos (schemas) en entrada y salida. Los modelos se encuentran en `app/schemas/` y garantizan la integridad antes de llegar a la capa de persistencia o servicio.
- Debe haber definida una imagen Docker del proyecto: **REALIZADO**.
 - La última imagen disponible del microservicio se encuentra en Docker Hub bajo el repositorio `socialbeats/analytics-and-dashboards`. Dockerfile disponible en la raíz.
- Gestión del código fuente: El código debe estar subido a un repositorio de Github siguiendo Github Flow: **REALIZADO**.
 - El código del microservicio se encuentra en el monorepo/repo correspondiente (*analytics-and-dashboards*). Se sigue la metodología de ramas y commits convencional.
- Integración continua: El código debe compilarse, probarse y generar la imagen de Docker automáticamente usando GitHub Actions u otro sistema de integración continua en cada commit: **REALIZADO**.
 - Todos los archivos que se encuentran dentro de la carpeta `.github/workflows/` sirven para la integración continua:
 - `create-releases.yml`: crea y publica la versión en Docker Hub al hacer tags.
 - Linters configurados en `pyproject.toml` (Black, Isort) para asegurar calidad.
- Debe haber pruebas de componente implementadas en Javascript para el código del backend utilizando Jest o similar. Como norma general debe haber tests para todas las funciones del API no triviales de la aplicación. Probando tanto escenarios positivos como negativos. Las pruebas deben ser tanto *in-process* como *out-of-process*: **REALIZADO**.
 - Al ser un microservicio en **Python**, utilizamos **Pytest**.
 - Todos los archivos se encuentran en `tests/`. Pruebas unitarias (`test_*_service.py`) e integración (`test_*_endpoints.py`) cubren la lógica de negocio y los endpoints. Se ejecutan con `pytest`.

MICROSERVICIO AVANZADO QUE GESTIONE UN RECURSO

- Implementar un frontend con rutas y navegación. **REALIZADO**.

- No disponemos de un frontend aislado, sino que está integrado en el repositorio común de la aplicación (SPA) junto al resto de microservicios.
 - Cuenta con sus correspondientes rutas y navegación integradas en la aplicación principal, permitiendo acceder a los dashboards y métricas de forma fluida (ej. rutas en `/app/dashboards` gestionadas por React Router).
- Usar el patrón materialized view para mantener internamente el estado de otros microservicios: **NO REALIZADO**.
 - Implementar cachés o algún mecanismo para optimizar el acceso a datos de otros recursos: **REALIZADO**.
 - Se usa **Redis** para cachear las respuestas de APIs externas y reducir la latencia y el consumo de cuota.
 - **Quotable API**: Las frases aleatorias servidas por `QuotableService` se cachean durante 1 hora (`app/services/quotable_service.py`), permitiendo servir la misma frase a múltiples usuarios sin golpear la API externa constantemente.
 - **Azure Translator**: Las traducciones se cachean durante 24 horas en `app/services/translator_service.py`. Esto evita volver a pagar/consumir cuota de Azure para textos que ya han sido traducidos previamente.
 - La conexión a Redis se gestiona globalmente en `app/middleware/rate_limiter.py`.
 - Consumir alguna API externa (distinta de las de los grupos de práctica) a través del backend o algún otro tipo de almacenamiento de datos en cloud como Amazon S3: **REALIZADO**.
 - Se han integrado múltiples APIs externas para enriquecer la funcionalidad:
 - **Quotable API**: Para obtener citas inspiradoras aleatorias relacionadas con la música/creatividad (`app/services/quotable_service.py`).
 - **Azure Translator**: Para ofrecer servicios de traducción de textos en la plataforma (`app/services/translator_service.py`).
 - Implementar el patrón “rate limit” al hacer uso de servicios externos: **REALIZADO**.
 - Se encuentra en el archivo `app/middleware/rate_limit.py`. Se usa la caché de *Redis* para controlar las llamadas a las APIs externas de *Quotable* y *Azure Translator* y adaptarlas a un consumo moderado y evitar que se disparen las llamadas.
 - Implementar un mecanismo de autenticación basado en JWT o equivalente: **REALIZADO**.
 - Se valida el JWT en el API Gateway, donde se introducen unas cabeceras especiales (`x-gateway-authenticated` y `x-user-id`, etc) y éstas se validan en nuestro middleware. Esto se puede encontrar en el archivo `app/middleware/authentication.py`.
 - Implementar el patrón “circuit breaker” en las comunicaciones con otros servicios: **REALIZADO**.
 - Se aplica *circuit breaker* de forma global mediante middleware (`app/middleware/circuit_breaker.py`) para prevenir caídas en cascada ante fallos repetidos.
 - El sistema monitoriza los errores 500 y “abre el circuito” temporalmente si se supera el umbral de fallos.

- También incluye lógica de reintento/resiliencia en las conexiones a infraestructuras críticas como Kafka ([app/services/kafka_consumer.py](#)) y Redis ([app/middleware/rate_limiter.py](#)), manejando excepciones de conexión y reestableciendo el servicio automáticamente.
- Implementar un microservicio adicional haciendo uso de una arquitectura serverless (Functions-as-a-Service): **NO REALIZADO**.
- Implementar mecanismos de gestión de la capacidad como throttling o feature toggles para rendimiento: **NO REALIZADO**.
- Cualquier otra extensión al microservicio básico acordada previamente con el profesor: **NO REALIZADO**.

NIVEL HASTA 5 PUNTOS

- Microservicio básico completamente implementado. **REALIZADO**.
 - Explicado anteriormente.
- Diseño de un customer agreement para la aplicación en su conjunto con, al menos, tres planes de precios que consideren características funcionales y extrafuncionales. **REALIZADO**.
 - Tarea grupal, explicado en el documento de nivel de acabado de la aplicación.
- Ficha técnica normalizada del modelo de consumo de las APIs externas utilizadas en la aplicación y que debe incluir al menos algún servicio externo de envío de correos electrónicos con un plan de precios múltiple como SendGrid. **REALIZADO**.
 - Tarea grupal, explicado en el documento de nivel de acabado de la aplicación. Nuestra parte de la ficha técnica es la de *Quotable* y *Azure Translator*.
- Documento incluido en el repositorio del microservicio (o en el wiki del repositorio en Github) por cada pareja **REALIZADO**.
 - Es este mismo documento. Además está explicado en el documento de nivel de acabado de la aplicación.
- Vídeo de demostración del microservicio o aplicación funcionando. **REALIZADO**.
 - Es el video de nuestro microservicio y el video demo de la presentación. El video está accesible en [Youtube](#)
- Presentación preparada para ser presentada en 30 minutos por cada equipo de 8/10 personas. **REALIZADO**.
 - Tarea grupal, explicado en el documento de nivel de acabado de la aplicación.

NIVEL HASTA 7 PUNTOS

- Debe incluir todos los requisitos del nivel hasta 5 puntos: **REALIZADO**.
 - Explicado anteriormente.

- Aplicación basada en microservicios básica implementada: **REALIZADO**.
 - Tarea grupal, explicado en el documento de nivel de acabado de la aplicación.
- Análisis justificativo de la suscripción óptima de las APIs del proyecto: **REALIZADO**.
 - Tarea grupal, explicado en el documento de nivel de acabado de la aplicación. Nuestra parte es el análisis de la suscripción óptima de *Azure Translator*.
- Al menos 3 de las características del microservicio avanzado implementados: **REALIZADO**.
 - Explicado anteriormente

NIVEL HASTA 9 PUNTOS

- Un mínimo de 20 pruebas de componente implementadas incluyendo escenarios positivos y negativos: **REALIZADO**.
 - Se han implementado suites exhaustivas de pruebas unitarias y de integración para todos los servicios principales (`beat_metrics`, `dashboards`, `widgets`).
 - Existen hasta 200 casos de prueba documentados en archivos como `tests/test_beat_metrics_service.py` o `tests/test_dashboard_endpoints.py`, cubriendo escenarios de éxito (creación, lectura, actualización) y manejo de errores (IDs inválidos, recursos no encontrados, errores de base de datos).
 - Utilizamos **pytest** y **mocks** (`unittest.mock`) para aislar los componentes y verificar la lógica de negocio sin depender de servicios externos en los tests unitarios.
- Tener el API REST documentado con swagger (OpenAPI): **REALIZADO**.
 - Al utilizar **FastAPI**, la documentación OpenAPI (Swagger) se genera de forma automática basándose en los modelos de **Pydantic** (`app/schemas/`) y las definiciones de los endpoints (`app/endpoints/`).
 - La especificación OpenAPI está disponible en `/openapi.json`.
 - El microservicio expone una interfaz interactiva Swagger UI en la ruta `/docs` y una documentación alternativa con ReDoc en `/redoc`, accesibles directamente desde el navegador al desplegar el servicio.
- Al menos 5 de las características del microservicio avanzado implementados: **REALIZADO**.
 - Explicado anteriormente
- Al menos 3 de las características de la aplicación basada en microservicios avanzada implementados: **REALIZADO**.
 - Tarea grupal, explicado en el documento de nivel de acabado de la aplicación.

NIVEL HASTA 10 PUNTOS

- Al menos 6 características del microservicio avanzado implementados: **REALIZADO**.
 - Explicado anteriormente

- Al menos 4 características de la aplicación basada en microservicios avanzada implementados:
REALIZADO.
 - Tarea grupal, explicado en el documento de nivel de acabado de la aplicación.
- Documento de uso de IA: **REALIZADO**
 - Incluimos aquí esta sección porque al no aparecer en el documento **Proyecto.pdf** con las instrucciones del trabajo creemos que era opcional. De igual modo se ha contribuido a ese documento con las explicaciones de las herramientas de IA utilizadas y del uso que se les dió por nuestra parte.

3. Descripción de el microservicio en la aplicación

- Resumen funcional (qué hace, para quién va dirigido):
 - Este microservicio se encarga del cálculo y almacenamiento de métricas (beat metrics) para los "beats" que suben los usuarios a la aplicación, y de exponer esas métricas mediante endpoints REST para su visualización en dashboards y widgets en el frontend.
 - Flujo principal:
 1. El usuario sube un beat (archivo).
 2. El endpoint **POST /analytics/beat-metrics** recibe la petición y valida que el usuario tiene permiso sobre el beat (ver **app/endpoints/beat_metrics.py**).
 3. El audio se guarda/descarga mediante **AudioFileHandler** y se analiza con el componente de audio (**app/services/audio_analyzer.py**). El análisis extrae **coreMetrics** y **extraMetrics**.
 4. El resultado se persiste en la colección **beat_metrics** de MongoDB (**app/services/beat_metrics_service.py**).
 5. Tras completar el cálculo, el servicio notifica al microservicio de beats (PUT a la URL configurada en **settings.BEATS_SERVICE_URL**) para marcar el beat como con métricas calculadas.
 6. El frontend consulta los endpoints de **dashboards** y **beat-metrics** para mostrar widgets y paneles (**app/endpoints/dashboards.py**, **app/services/dashboard_service.py**).
- Requisitos clave atendidos (resumen):
 - API REST completa con endpoints de consulta y gestión de métricas y dashboards (GET/POST/PUT/DELETE).
 - Autenticación/autorización sobre endpoints mediante middleware (**app/middleware/authentication.py**).
 - Persistencia en MongoDB para **beat_metrics** y **dashboards**.
 - Procesamiento de audio y cálculo de métricas (análisis por **audio_analyzer**).
 - Comunicación con otros microservicios (notificación al microservicio de beats) usando **httpx**.
 - Tests de componente y de endpoints incluidos en **tests/**.

4. Descripción del API REST del microservicio

Provee una descripción por endpoint (método, ruta, request, response, códigos HTTP, ejemplos).

Dashboards

- GET /analytics/dashboards
 - Descripción: Lista dashboards. Usuarios normales ven solo sus dashboards; los administradores ven todos.
 - Parámetros: `skip` (int, opcional), `limit` (int, opcional)
 - Respuesta (200): Lista de objetos `DashboardResponse`.
- GET /analytics/dashboards/{dashboard_id}
 - Descripción: Obtiene un dashboard por su identificador.
 - Respuesta (200): `DashboardResponse`; 404 si no existe.
- POST /analytics/dashboards
 - Descripción: Crea un nuevo dashboard. `owner_id` se asigna desde el contexto del usuario.
 - Body (JSON): `DashboardCreate` — ejemplo: `{ "name": "Mi panel", "beatId": "beat_12345" }`
 - Respuesta (201): `DashboardResponse`.
- PUT /analytics/dashboards/{dashboard_id}
 - Descripción: Actualiza un dashboard (solo nombre permitidos en la API pública).
 - Body (JSON): `DashboardUpdate` — ejemplo: `{ "name": "Panel actualizado" }`
 - Respuesta (200): `DashboardResponse`.
- DELETE /analytics/dashboards/{dashboard_id}
 - Descripción: Elimina un dashboard.
 - Respuesta (200): Mensaje de éxito.

Referencias de implementación:

- `app/endpoints/dashboards.py`
- `app/services/dashboard_service.py`

Widgets

- GET /analytics/widgets
 - Descripción: Lista widgets del sistema; permite filtrar por `dashboardId`.
 - Parámetros: `dashboardId` (opcional), `skip`, `limit`.
 - Respuesta (200): Lista de `WidgetResponse`.
- GET /analytics/widgets/{widget_id}
 - Descripción: Recupera un widget por su identificador.
 - Respuesta (200): `WidgetResponse`.
- POST /analytics/widgets
 - Descripción: Crea un widget asociado a un dashboard.
 - Body (JSON): `WidgetCreate` — ejemplo: `{ "dashboardId": "id_del_dashboard", "metricType": "BPM" }`

- Respuesta (201): `WidgetResponse`.
- PUT /analytics/widgets/{widget_id}
 - Descripción: Actualiza un widget.
 - Body (JSON): `WidgetUpdate`.
 - Respuesta (200): `WidgetResponse`.
- DELETE /analytics/widgets/{widget_id}
 - Descripción: Elimina un widget.
 - Respuesta (200): Mensaje de éxito.
- GET /analytics/dashboards/{dashboard_id}/widgets
 - Descripción: Lista widgets de un dashboard específico.
 - Respuesta (200): Lista de `WidgetResponse`.

Referencias de implementación:

- `app/endpoints/widgets.py`
- `app/services/widget_service.py`

Beat Metrics

- GET /analytics/beat-metrics
 - Descripción: Lista métricas de beats; permite filtrar por `beatId`.
 - Parámetros: `beatId` (opcional), `skip`, `limit`.
 - Respuesta (200): Lista de `BeatMetricsResponse`.
- GET /analytics/beat-metrics/{beat_metrics_id}
 - Descripción: Recupera métricas por su identificador.
 - Respuesta (200): `BeatMetricsResponse`.
- POST /analytics/beat-metrics
 - Descripción: Crea una entrada de métricas analizando un audio.
 - Envío: `multipart/form-data` para `audioFile` o formulario con `audioUrl`.
 - Campos: `beatId` (required), `audioFile` (file, opcional), `audioUrl` (string, opcional).
 - Respuesta (201): `BeatMetricsResponse`.
- PUT /analytics/beat-metrics/{beat_metrics_id}
 - Descripción: Actualiza una entrada de métricas.
 - Body (JSON): `BeatMetricsUpdate`.
 - Respuesta (200): `BeatMetricsResponse`.
- DELETE /analytics/beat-metrics/{beat_metrics_id}
 - Descripción: Elimina una entrada de métricas.
 - Respuesta (200): Mensaje de éxito.

Referencias de implementación:

- `app/endpoints/beat_metrics.py`
- `app/services/beat_metrics_service.py`

5. Arquitectura del microservicio

Este microservicio se distingue del resto de la arquitectura (construida mayoritariamente en Node.js) por estar desarrollado en **Python** utilizando el framework **FastAPI**. Esta decisión arquitectónica se fundamenta en la amplia disponibilidad de librerías para el procesamiento de datos y análisis de audio (como `librosa` y `numpy`), lo que lo hace idóneo para el dominio de "Analytics".

Estructura del Proyecto

A continuación se presenta la estructura de carpetas y ficheros principales, siguiendo una arquitectura limpia y modular:

```
analytics-and-dashboards/
├── .github/workflows/           # CI/CD pipelines (Tests, Release)
├── app/
│   ├── core/                   # Configuración global (Vars de entorno, Logging)
│   │   ├── config.py
│   │   └── logging.py
│   ├── database/               # Capa de Persistencia
│   │   └── config.py           # Conexión asíncrona con Motor (MongoDB)
│   ├── endpoints/              # Controladores REST (Routers)
│   │   ├── beat_metrics.py
│   │   ├── dashboards.py
│   │   └── widgets.py
│   ├── middleware/             # Middlewares Globales
│   │   ├── authentication.py  # Validación de identidad (Gateway)
│   │   ├── circuit_breaker.py # Resiliencia HTTP
│   │   └── rate_limiter.py   # Control de flujo con Redis
│   ├── models/                 # Modelos de BBDD (Estructura interna)
│   ├── schemas/                # Schemas Pydantic (Validación y Serialización)
│   ├── services/               # Lógica de Negocio y comunicación externa
│   │   ├── audio_analyzer.py  # Lógica "Heavy" de análisis
│   │   ├── kafka_consumer.py # Consumidor de eventos
│   │   ├── quotable_service.py # Integración API Externa 1
│   │   └── translator_service.py # Integración API Externa 2
│   └── utils/                  # Utilidades transversales
└── docs/                      # Documentación extendida
```

```

└── tests/          # Testing con Pytest
    └── conftest.py  # Fixtures y configuración de tests
        ...
    ...
└── main.py        # Entrypoint de la aplicación FastAPI
└── docker-compose.yml # Orquestación local
└── Dockerfile      # Empaquetado de producción
└── pyproject.toml  # Dependencias y configuración (Poetry/Pip)
└── pytest.ini      # Configuración de Testing

```

Componentes Clave

Seguridad (Gateway Offloading)

La seguridad sigue el patrón de **Gateway Offloading**. El microservicio no emite ni firma tokens JWT, sino que confía en las cabeceras inyectadas por el API Gateway ([api-gateway](#)).

- **Mecanismo:** El middleware de autenticación ([app/middleware/authentication.py](#)) intercepta cada petición e inspecciona las cabeceras `x-gateway-authenticated` y `x-user-id`.
- **Validación:** Si las cabeceras no están presentes o son inválidas, se rechaza la petición con un [401 Unauthorized](#) antes de llegar a la lógica de negocio.
- **Contexto:** Los datos del usuario se inyectan en el estado de la petición (`request.state.user`), haciéndolos accesibles para audit logs y validación de propiedad en los endpoints.

Persistencia en MongoDB (Motor)

Para la persistencia de datos se utiliza **MongoDB**, aprovechando su flexibilidad para almacenar estructuras JSON complejas como las métricas musicales y configuraciones de dashboards dinámicos.

- **Driver Asíncrono:** Se utiliza `motor` (Motor: Async IO for MongoDB) para no bloquear el Event Loop de Python, permitiendo manejar una alta concurrencia.
- **Validación de Datos:** Aunque MongoDB es *schema-less*, la integridad de los datos se aplica estrictamente en la capa de aplicación mediante modelos de **Pydantic** ([app/schemas/](#)). Nada entra ni sale de la base de datos sin ser validado previamente.
- **Colecciones:**
 - `beat_metrics`: Almacena el resultado del análisis de audio.
 - `dashboards`: Configuraciones de los paneles de usuario.
 - `widgets`: Configuración individual de cada widget visual.

KAFKA (Event Driven)

El microservicio se integra en la coreografía de eventos del sistema mediante **Apache Kafka**, utilizando la librería `aiokafka`.

- **Consumer:** Escucha eventos de dominio (como `beat.uploaded` o `user.created`) en [app/services/kafka_consumer.py](#) para reaccionar de forma asíncrona. Por ejemplo, iniciar el cálculo de métricas en segundo plano cuando un usuario sube una nueva canción, desacoplando la subida del archivo del procesamiento pesado.

- **Resiliencia:** La conexión con Kafka maneja reconexiones automáticas y está supervisada por los endpoints de Health Check.

REDIS (Caching & Rate Limiting)

Redis actúa como una pieza fundamental para el rendimiento y la estabilidad:

- **Rate Limiting:** El middleware `app/middleware/rate_limiter.py` utiliza Redis para mantener contadores atómicos de peticiones, protegiendo los endpoints públicos de abusos y ataques de denegación de servicio (DDoS).
- **Caché de APIs Externas:** Para optimizar costes y latencia, las respuestas de servicios de terceros (Quotable API, Azure Translator) se almacenan temporalmente en Redis. Si un dato ya fue solicitado recientemente, se sirve desde la memoria caché instantáneamente, evitando llamadas de red innecesarias y preservando las cuotas de uso de las APIs externas.