Nivel de acabado del microservicio beats-interaction

1. Beats-interaction

Pareja: Daniel Galván Cancio y Jaime Linares Barrera

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 las playlist, comentarios y puntuaciones) y tenemos algunos métodos (`create` y `post`) para la moderación de contenido. Todo el código se encuentra en los archivos de las carpetas `src/routes` y `src/services`.
- La API debe tener un mecanismo de autenticación: **REALIZADO**.
    - La autenticación se encuentra en `src/middlewares/authMiddlewares.js`. 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. La autenticación se hace con JWT.
- 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 referido con nuestro microservicio son: `src/pages/app/beats-interaction/*` y `src/services/beats-interaction/`. Además, en algunos componentes como `src/pages/app/beats/BeatDetailPage.jsx` podemos encontrar la inserción de componentes desarrollados por nosotros, o algunos botones como el de `ver playlists del usuario` de la vista de perfil de usuario.
- 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 clusted 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 todos los archivos de la carpeta `src/routes`. La documentación de la API esta disponible en https://api.socialbeats.es/socialbeats-api/api/v1/docs/, sólo basta con seleccionar el

microservicio de beats-interactions. El OAS también está disponible en https://api.socialbeats.es/socialbeats-api/api/v1/oas/beats-interaction.yaml

- Se debe tener una documentación de todas las operaciones de la API incluyendo las posibles peticiones y las respuestas recibidas: **REALIZADO**.
    - Tenemos el archivo `spec/oas.yaml` donde aparece completamente definida todas las operaciones del microservicio peticiones y todas las posibles respuestas que pueden darse.
- Debe tener persistencia utilizando *MongoDB* u otra base de datos no SQL: **REALIZADO**.
    - Tenemos la conexión a la base de datos en el archivo `src/db.js`.
- Deben validarse los datos antes de almacenarlos en la base de datos (por ejemplo, haciendo uso de *mongoose*): **REALIZADO**.
    - En todos los modelos se hacen validaaciones de la propia entidad y esto puede encontrarse en los archivos de `src/models/` (eso usa mongoose). Además en los servicios también se hacen más validaciones y estás pueden enconttrarsse en los aarchivos de `src/services/`.
- Debe haber definida una imagen Docker del proyecto: **REALIZADO**.
    - La última imagen disponible del microservicio se encuentra en https://hub.docker.com/repository/docker/socialbeats/beats-interaction. La última imagen disponible de todos los micrroservicios se encuentran en https://hub.docker.com/repositories/socialbeats.
- Gestión del código fuente: El código debe estar subido a un repositorio de Github siguiendo Github Flow: **REALIZADO**.
    - El código del microservicio se encuentra accesible en la siguiente ruta: https://github.com/SocialBeats/beats-interaction. La metodología de ramas y de commits se encuentran en https://github.com/SocialBeats/docs/tree/main/work_methodology/work_methodology.md
- 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 lo archivos que se encuentran dentro de la carpeta `github/workflows/` sirven para la integración continua. En concreto son:
        - `conventional-commits.yml`: se encarga de verificar que se sigue conventional commits.
        - `create-releases.yml`: siempre que se hace *push* de una *tag* a la rama *main* se crea y pública una versión del microservicio en Docker. Es el workflow más importante porque integra el ciclo de CD.
        - `linter.yml`: verificar, al crear una pull o hacer push a main o develop, que todos los archivos siguen ESLint. En caso de fallar por incumplimiento de lint, se aplica un `npm run lint:fix` para hacer que todos los archivos sigan con el estilo definido.
        - `run-tests.yml`: ejecuta los tests inproc, al crear una pull o hacer push a main o develop, para comprobar que no se ha roto ninguna funcionalidad.

- 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**.

    - Todos los archivos que se encuentran dentro de la carpeta `tests/` del proyecto. Los tests in-process se encuentran en `tests/unit` y los test out-of-process se encuentran en `test/integration`. Para la ejecución de los in-process se puede hacer cualquiera de los siguientes comandos: `npm run test`, `npm test`, `npm run test:unit`, `npm run test:inproc`, `npm run test:watch`. Se puede ver la cobertura de pruebas con `npm run test:coverage`. Para los tests out-of-process basta con hacer `npm run test:outproc`. Esto levanta una imagen de la aplicación usando el docker-compose de tests (que no levanta kafka, redis ni el pricing) y ejecuta los tests de las rutas para validar el correcto funcinoamiento de los endpoints. Estos tests no añaden cobertura de pruebas.

MICROSERVICIO AVANZADO QUE GESTIONE UN RECURSO

- Usar el patrón materialized view para mantener internamente el estado de otros microservicios: **REALIZADO**.

    - Se puede encontra dentro del microservicio en los archivos `src/models/BeatMaterialized.js` y `src/models/UserMaterialized.js`. Ahí se definen las propiedades que guardamos de las bases de datos de los microservicios *user-auth* y de *beats-upload*. Con los eventos de Kafka mantenemos actualizada la materialized view y usamos sus datos para enriquecer nuestras respuestas en todos los servicios, ya que referenciamos usuarios y beats constantemente. También usamos el materialized view para validar la existencia de recursos antes de referenciarlos de modo que no existan inconsistencias.

- Implementar cachés o algún mecanismo para optimizar el acceso a datos de otros recursos: **REALIZADO**.

    - Se usa *Redis* para tener cacheadas las llamadas a la API de *OpenRouter* y para emplear el rate-limit y nunca exceder los límites de uso de la misma. El uso de *Redis* se encuentra en `src/cache.js`, y su uso está sobre todo en el archivo `src/utils/moderationEngine.js` pero también en `src/utils/rateLimit.js`. Si el materialized view se considera una caché permanente, entonces también se aplica caché con usuarios y beats tal y como se explica en el punto anterior.

- 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 ha integrado la API de *OpenRouter* para moderar el contendo de los comentarios, las playlists y los ratings. Esto se puede ver en el archivo `src/utils/openRouterClient.js`, que es donde se conecta nuestro microservicio con la API externa.

- Implementar el patrón "rate limit" al hacer uso de servicios externos: **REALIZADO**.

    - Se encuentra en el archivo `src/utils/rateLimit.js`. Se usa la caché de *Redis* para controlar las llamadas a la API de *OpenRouter* y adaptarlas al plan gratuito. En nuestro caso como queremos evitar ser baneados hacemos un uso conservador, es decir, no aprovechamos la

capacidad máxima de la API, estableciendo límites ligeramente por debajo de los reales (18 en vez de 20 y 45 en vez de 50).

- 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*) y éstas se validan en nuestro middleware. Esto se puede encontrar en el archivo `src/middlewares/authMiddlewares.js`.
- Implementar el patrón "circuit breaker" en las comunicaciones con otros servicios: **REALIZADO**.
    - Se aplica *circuit breaker* para reintentar las conexiones tanto con *Redis* como con *Kafka*, de modo que si se desconectan nuestro microservicio no crashea y trata de reconectarse. Además se pueden habilitar y deshabilitar dichas conexiones con las variables de entorno `ENABLE_KAFKA` y `ENABLE_REDIS`. El *circuit breaker* puede observarse en `src/cache.js` para *Redis* y `src/services/kafkaConsumer.js` para *Kafka*
- 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: **REALIZADO**.
    - Se han añadido características adicionales como un logger en `logger.js`, la gestión de githooks con *husky*, que se ve en la carpeta `.husky` y además se autoinstala al instalar las dependencias, el soporte a varios entornos de desarrollo, teniendo varios .env.example y varios Dockerfiles y docker-compose. Además tenemos la posibilidad de activar o desactivar redis, kafka, el pricing con variables de entorno. hay endpoints de salud que usamos para ver el estado de las conexiones con redis, kafka y la base de datos (disponible en `src/routes/healthRoutes.js`), y hay endpoints de versionado y changelog (disponible en `src/routes\/aboutRoutes.js`). Por último, se ha implementado un mecanismo de apagado del contenedor para que se cierren las conexiones, se finalicen las solicitudes que se estén gestionando en ese momento y se apague el contenedor de forma correcta. Todo esto se puede observar en `main.js`.

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 *OpenRouter*.

- 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 de nuestro microservicio está accesible en Youtube, y el de la aplicación en https://youtu.be/gb9y5TR-ftw.

- 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 *OpenRouter* y también nos encargamos de hacer la plantilla de ese documento.

- 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**.

    - Sí. Se especifica en el documento de uso de IA, pero hemos hecho un test suite muy rígido. Existen pruebas para todos los campos y todas las validaciones de errores controlados en nuestras entidades, rutas y servicios. Todo esto se puede ver fácilmente en los archivos de `test/`. Por ejemplo, sólo para el método de crear una playlit, existen 13 casos de pruebas (incluyendo negativos). Esto se ve en el describe llamado `create playlist test` del archivo `tests/unit/services.playlist.test.js`. Como se ha comentado, usamos IA generativa para validar todos los escenarios posibles.

- Tener el API REST documentado con swagger (OpenAPI): **REALIZADO**.

    - Se puede ver en el `spec/oas.yaml`. Ese archivo se autogenera gracias al *js-doc* de los archivos en `src/routes`. Ese contenido lo utiliza la librería *swagger-jsdoc* para crear el oas.yaml. Además tenemos definidos los schemas de nuestras entidades en `src/models/OASSchemas.js`. El microservicio dispone de una UI de Swagger que se renderiza gracias lo anterior y a la librería *swagger-ui-express*.

- 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 del microservicio en la aplicación

Resumen funcional

El microservicio **beats-interaction** gestiona las **interacciones sociales** de los usuarios sobre beats y playlists dentro de la aplicación:

- **Comentarios** sobre beats y playlists.
- **Ratings** (puntuación + comentario opcional) sobre beats y playlists.
- **Playlists**: creación, edición, visibilidad (pública/privada), colaboradores, e items (beats añadidos).
- **Moderación**:
    - Creación de **Moderation Reports** contra comentarios/ratings/playlists.
    - Estado del reporte: `Checking` | `Rejected` | `Accepted`.
    - Integración con moderación automática vía IA.
    - Healthcheck del subsistema de moderación (Redis/Cron/pending reports).

Además, para minimizar latencia entre microservicios y evitar lecturas directas a servicios externos, utiliza Materialized Views de:

- `users` (desde user-auth)
- `beats` (desde beats-upload)

Flujo principal (alto nivel)

1. El usuario realiza una acción (crear playlist / comentar / puntuar / reportar).
2. El microservicio valida autenticación (bearerAuth JWT).
3. Se validan reglas de dominio (p.ej. no duplicar rating por beat/playlist, límites de longitud, playlist pública para rating, etc.).
4. Se aplica moderación:
    - Moderación síncrona si el SLA del rate-limit lo permite.
    - Si no, el contenido pasa a cola/pendiente y un CronJob lo revisa después.
5. Se persiste en MongoDB la entidad correspondiente.
6. Opcional: se publican/consumen eventos Kafka para mantener Materialized Views y consistencia entre microservicios.

4. Arquitectura y componentes relevantes

Arquitectura

```
.
│
├── .github                              # Plantillas de issues y workflows de
Github
│   ├── ISSUE_TEMPLATE
│   │   ├── bug-report.yml
│   │   ├── issue-template.yml
│   │   └── security-vulnerability.yml
│   └── workflows
│       ├── conventional-commits.yml
│       ├── create-releases.yml
│       ├── linter.yml
│       └── run-tests.yml
│
├── .husky                               # Husky para la gestión de githooks
│   ├── commit-msg
│   ├── pre-commit
│   └── _
│       ├── .gitignore
│       ├── applypatch-msg
│       ├── commit-msg
│       ├── h
│       ├── husky.sh
│       ├── post-applypatch
│       ├── post-checkout
│       ├── post-commit
│       ├── post-merge
│       ├── post-rewrite
│       ├── pre-applypatch
│       ├── pre-auto-gc
│       ├── pre-commit
```

```
│       ├── pre-merge-commit
│       ├── pre-push
│       ├── pre-rebase
│       └── prepare-commit-msg
│
├── .vscode                          # Configuración del entorno de desarrollo
│   └── settings.json
│
├── scripts                          # Scripts para preparar el entorno de
desarrollo
│   └── copyEnv.cjs
│
├── spec                             # Especificación de la API
│   └── oas.yaml
│
├── src
│   ├── cache.js                     # Conexión con Redis
│   ├── db.js                        # Conexión con Mongo
│   ├── middlewares                  # Midlewares para la autenticación
│   │   └── authMiddlewares.js
│   ├── models                       # Modelo de datos
│   │   ├── BeatMaterialized.js
│   │   ├── Comment.js
│   │   ├── ModerationReport.js
│   │   ├── Playlist.js
│   │   ├── Rating.js
│   │   ├── UserMaterialized.js
│   │   ├── OASSchemas.js
│   │   └── models.js
│   ├── routes                       # Documentación de las rutas y definición
de endpoints
│   │   ├── aboutRoutes.js
│   │   ├── commentRoutes.js
│   │   ├── healthRoutes.js
│   │   ├── moderationReportRoutes.js
│   │   ├── playlistRoutes.js
│   │   └── ratingRoutes.js
│   ├── services                     # Lógica de gestión de peticiones
│   │   ├── commentService.js
│   │   ├── kafkaConsumer.js
│   │   ├── moderationReportService.js
│   │   ├── playlistService.js
│   │   └── ratingService.js
│   └── utils                        # Métodos útiles y conexiones a OpenRouter
│       ├── moderationCron.js
│       ├── moderationEngine.js
│       ├── moderationWorker.js
│       ├── openRouterClient.js
│       ├── rateLimit.js
│       ├── spaceConnection.js
│       └── versionUtils.js
│
├── tests
│   ├── integration                  # Tests outproc
```

```
│   │   ├── integration.comment.test.js
│   │   ├── integration.health.test.js
│   │   ├── integration.playlist.test.js
│   │   └── integration.rating.test.js
│   ├── setup                          # Configuración de tests inproc y outproc
│   │   ├── setup-integration.js
│   │   └── setup.js
│   └── unit                           # Tests unitarios
│       ├── entities.comment.test.js
│       ├── entities.ModerationReport.test.js
│       ├── entities.playlist.test.js
│       ├── entities.rating.test.js
│       ├── routes.comment.test.js
│       ├── routes.health.test.js
│       ├── routes.moderationReport.test.js
│       ├── routes.playlist.test.js
│       ├── routes.rating.test.js
│       ├── services.comment.test.js
│       ├── services.kafkaConsumer.test.js
│       ├── services.moderationReport.test.js
│       ├── services.playlist.test.js
│       └── services.rating.test.js
│
├── .dockerignore
├── .env
├── .env.development.example
├── .env.docker-compose.example
├── .env.docker.example
├── .env.example
├── .eslintrc.json
├── .gitignore
├── .markdownlint.json
├── .prettierrc.json
├── .version
├── CHANGELOG.md
├── commitlint.config.cjs
├── docker-compose-dev.yml
├── docker-compose-test.yml
├── docker-compose.yml
├── Dockerfile
├── Dockerfile-dev
├── LICENSE
├── logger.js
├── main.js
├── package-lock.json
├── package.json
├── README.md
├── vitest.config.js
└── vitest.integration.config.js
```

Persistencia (MongoDB)

Colecciones principales:

- `comments`
- `ratings`
- `playlists`
- `moderationReports`
- `users_materialized`
- `beats_materialized`

Seguridad

- **Autenticación**: JWT vía `Authorization: Bearer <token>`
- **Autorización** (ejemplos):
  - Solo el autor puede editar/eliminar su comment.
  - Solo el autor del rating puede editar/eliminar su rating.
  - Solo el owner puede editar/eliminar playlist (colaboradores pueden añadir/quitar beats si está permitido).
  - Reportes: un usuario no puede reportar su propio contenido.
- **Comunicación con otros microservicios**: Existe una variable de entorno llama INTERNAL_API_KEY que sive para comunicarse con el servicio de usuarios de forma síncrona (para realizar un command que solicita la eliminación de un usuario tras 5 denuncias aceptadas)

Kafka + Materialized View

- **Eventos consumidos**: Nuestro objetivo es tener datos “cacheados”/materializados para validar y enriquecer respuestas sin depender de llamadas runtime a otros microservicios. Este "caché" es persistente, por lo cual se actualiza con los eventos de eliminado y actualización de los recursos.
  - `USER_CREATED` / `USER_UPDATED` / `USER_DELETED`: actualizar `UserMaterialized`
  - `BEAT_CREATED` / `BEAT_UPDATED` / `BEAT_DELETED`: actualizar `BeatMaterialized`
  - Uso en endpoints:
    - Comprobación de existencia de `userId` cuando Kafka/materialized está habilitado.
    - Respuestas enriquecidas como `DetailedPlaylist` (incluye `collaboratorsData` y `beatsData`).

- **Eventos creados**: Nos comunicamos con el microservicio de social y enviar eventos sociales para el feed de la aplicación.
  - `COMMENT_CREATED`: Creación de un comentario
  - `RATING_CREATED`: Creación de un rating

5. Descripción del API REST del microservicio

La especificación OpenAPI (OAS) define rutas, request/response y ejemplos. Aquí se resume por grupos funcionales aunque recomendamos revisar el archivo `oas.yaml`.

5.1 Documentation & meta

- GET `/api/v1/about`

Devuelve el README renderizado en HTML.

- GET `/api/v1/version`

  Devuelve versión del API desde .version.

- GET `/api/v1/changelog`

  Devuelve changelog en HTML con filtros por versiones/rango.

5.2 Health

- GET `/api/v1/health`

  Healthcheck general (API, DB, uptime, entorno).

- GET `/api/v1/kafka/health`

  Estado de conectividad con Kafka.

- GET `/api/v1/moderation/health`

  Estado del subsistema de moderación (Redis, cron status, pending reports, etc.).

5.3 Comments

Comentarios en beats

- POST `/api/v1/beats/{beatId}/comments`

  Crea comentario asociado al beat autenticado.

- GET `/api/v1/beats/{beatId}/comments`

  Lista paginada de comentarios del beat (page, limit).

Comentarios en playlists

- POST `/api/v1/playlists/{playlistId}/comments`

  Crea comentario asociado a playlist.

- GET `/api/v1/playlists/{playlistId}/comments`

  Lista paginada de comentarios de playlist.

Operaciones sobre comment

- GET `/api/v1/comments/{commentId}`

  Obtiene un comment por id (útil para moderación).

- PUT/PATCH `/api/v1/comments/{commentId}`

  Edita el texto (solo autor).

- DELETE `/api/v1/comments/{commentId}`

  Borra comment (idempotente en vuestro diseño: devuelve éxito aunque no exista o sea inválido, según OAS).

5.4 Ratings

Ratings en beats

- POST `/api/v1/beats/{beatId}/ratings`

  Crea rating (solo uno por usuario y beat).

- GET `/api/v1/beats/{beatId}/ratings`

  Lista paginada + promedio (average) + contador (count).

- GET `/api/v1/beats/{beatId}/ratings/me`

  Rating del usuario autenticado para ese beat.

Ratings en playlists

- POST `/api/v1/playlists/{playlistId}/ratings`

  Crea rating (playlist debe existir y ser pública).

- GET `/api/v1/playlists/{playlistId}/ratings`

  Lista paginada + average + count.

- GET `/api/v1/playlists/{playlistId}/ratings/me`

  Rating del usuario autenticado para esa playlist.

Operaciones sobre rating

- GET `/api/v1/ratings/{ratingId}`

  Obtiene rating por id.

- PUT/PATCH `/api/v1/ratings/{ratingId}`

  Edita score y comment opcional (solo autor).

- DELETE `/api/v1/ratings/{ratingId}`

  Borrado idempotente con deleted: true/false (y 401 si no es dueño).

5.5 Playlists

- POST `/api/v1/playlists`

  Crea playlist (name, description, isPublic, collaborators, items).

- GET `/api/v1/playlists/me`

  Playlists del usuario autenticado (owner o collaborator).

- GET `/api/v1/playlists/user/{userId}`

  Playlists públicas del usuario objetivo + las compartidas con el requester.

- GET `/api/v1/playlists/public`

  Listado público con filtros + paginación (name, ownerId, page, limit).

- GET `/api/v1/playlists/{id}`

  Obtiene playlist si: pública o requester es owner/collaborator. Responde normalmente con DetailedPlaylist (enriquecida con materialized view).

- PUT/PATCH `/api/v1/playlists/{id}`

  Actualiza playlist (solo owner): name/description/isPublic/collaborators/items.

- DELETE `/api/v1/playlists/{id}`

  Elimina playlist (solo owner).

- POST `/api/v1/playlists/{id}/items`

  Añade beat (beatId) si owner/collaborator. Sin duplicados. Máx 250.

- DELETE `/api/v1/playlists/{id}/items/{beatId}`

  Elimina beat de la playlist si owner/collaborator.

5.6 Moderation Reports

Creación de reportes (el `authorId` se infiere según el recurso reportado):

- POST `/api/v1/comments/{commentId}/moderationReports`
- POST `/api/v1/ratings/{ratingId}/moderationReports`
- POST `/api/v1/playlists/{playlistId}/moderationReports`

Consultas:

- `GET /api/v1/moderationReports`

  Lista completa de reports (sin paginación, orden desc).

- `GET /api/v1/moderationReports/{moderationReportId}`

  Report por id.

- `GET /api/v1/moderationReports/me`

  Reports donde el usuario autenticado es `authorId` (reportado).

- `GET /api/v1/moderationReports/users/{userId}`

Reports donde el `authorId` es el usuario indicado.

6. Moderación con IA (OpenRouter) + estrategia SLA

Qué se modera

Actualmente se modera:

- Título y descripción de playlists
- Comentarios
- Texto de ratings

API externa integrada

- **Proveedor**: OpenRouter
- **Modelo**: `meta-llama/llama-3.2-3b-instruct:free`
- **Motivo**: opción gratuita suficiente para moderación básica y rápida.

Rate limit aplicado en beats-interaction (capa interna)

Para evitar bloqueos del provider y mantener estabilidad:

- **Máx. 18 requests/minuto**
- **Máx. 45 requests/día**
- Si se excede, el contenido queda pendiente y lo procesa un **CronJob** cuando “haya hueco”.

7. Gestión de errores y validación (criterios comunes)

Ejemplos:

- **401 Unauthorized**: token missing/invalid o no-owner en acciones sensibles (editar/borrar).
- **402 Payment Required**: cuando se excede el SLA de nuestra API.
- **403 Forbidden**: acceso denegado (p.ej. playlist privada sin permisos).
- **404 Not found**: beat/playlist/comment/rating inexistente.
- **409 Conflict**: ya existe moderation report en revisión.
- **422 Unprocessable Entity**:
  - validaciones (longitud, rango score)
  - reglas de negocio (no reportar propio contenido)
  - kafka enabled y `userId` no existe en materialized store
- **500 Internal Server Error**: errores inesperados.
- **503 Service unavailable**: cuando el microservicio está caído, en proceso de actualización o creandose y el API-Gateway no puede comunicarse con él.