Nivel de acabado del microservicio social

1. Social

Pareja: Andrés Martínez Reviriego y Sergio Álvarez Piñón

2. Nivel de acabado

Nivel objetivo: 10 – Se cubren los requisitos del microservicio básico y varias capacidades avanzadas.

MICROSERVICIO BÁSICO QUE GESTIONA 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**.
    - Microservicio básico que gestiona un recurso con API REST con CRUD y códigos adecuados: amistad (requests, aceptar/rechazar, listar, borrar) en `src/routes/friendshipRoutes.js`, mensajería (conversaciones, mensajes, paginación) en `src/routes/messagingRoutes.js`, feed en `src/routes/feedRoutes.js`.
- La API debe tener un mecanismo de autenticación: **REALIZADO**.
    - Middleware JWT + cabeceras del API Gateway (`x-gateway-authenticated`, `x-user-id`) en `src/middlewares/authMiddlewares.js`. Las rutas están abiertas solo para `docs`, `health`, `about` y `version`.
- Debe tener un frontend que permita hacer todas las operaciones de la API: **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/pages/app/social/* y social. Además, en algunos componentes como Feed.jsx y UsersExploreSection.jsx podemos encontrar la inserción de componentes desarrollados por nosotros, o algunos botones como el de enviar solicitud de amistad desde la vista de exploración de usuarios o el feed social integrado.
- Debe estar desplegado y ser accesible desde la nube: **REALIZADO**.
    - Imagen `socialbeats/social` consumida en `docker-compose.yml` (servicio `social-service` con `PORT=3004`, `KAFKA_BROKER=kafka:9092`, Mongo `social-mongodb`).
- La API que gestione el recurso también debe ser accesible en una dirección bien versionada: **REALIZADO**.
    - Prefijo `/api/v1/...` en todas las rutas.
- Se debe tener una documentación de todas las operaciones de la API: **REALIZADO**.
    - OpenAPI autogenerada vía JSDoc en `spec/oas.yaml`; Swagger UI habilitada (véase `src/routes/aboutRoutes.js` para meta).

- Debe tener persistencia utilizando *MongoDB* u otra base de datos no SQL: **REALIZADO**.
    - MongoDB con modelos `Friendship`, `Conversation`, `Message`, `Feed`, `User`, `Beat` en `src/models`.
- Deben validarse los datos antes de almacenarlos en la base de datos: **REALIZADO**.
    - Mongoose schemas con enums y required (ej. `src/models/Friendship.js`); validaciones adicionales en servicios (IDs válidos, longitudes de texto, ownership).
- Debe haber definida una imagen Docker del proyecto: **REALIZADO**.
    - Dockerfile y docker-compose (local, dev, test) en raíz; imágenes publicadas bajo `socialbeats/social`.
- Gestión del código fuente: El código debe estar subido a un repositorio de Github siguiendo Github Flow: **REALIZADO**.
    - Repo con Conventional Commits, lint/prettier, Husky, workflows (lint/tests/releases) en `workflows`.
- Integración continua: El código debe compilarse, probarse y generar la imagen de Docker automáticamente: **REALIZADO**.
    - Workflows `linter.yml` y `run-tests.yml` ejecutan lint y vitest; `create-releases.yml` publica imagen en Docker Hub al taggear.
- Debe haber pruebas de componente implementadas en Javascript para el código del backend utilizando Jest o similar: **REALIZADO**.
    - Vitest unitarios en `tests/unit` (mensajería, feed, socket, entidades) y de integración en `tests/integration`. Scripts `npm test`, `npm run test:integration`, `npm run test:coverage`.

MICROSERVICIO AVANZADO QUE GESTIONE UN RECURSO

- Usar el patrón materialized view para mantener internamente el estado de otros microservicios: **REALIZADO**.
    - Usuarios y beats materializados desde eventos Kafka (`USER_*`, `BEAT_*`) en `src/services/kafkaConsumer.js` para enriquecer feed y validar IDs.
- Implementar cachés o algún mecanismo para optimizar el acceso a datos de otros recursos: **REALIZADO**.
    - Caché en memoria de usuarios para enriquecer peticiones de amistades en `src/utils/cache.js`.
- Consumir alguna API externa a través del backend: **NO REALIZADO**.
    - No se comsume API externa.
- Implementar el patrón “rate limit” al hacer uso de servicios externos: **REALIZADO**.

    - La API Gateway utiliza un mecanismo de rate limit.
- Implementar un mecanismo de autenticación basado en JWT o equivalente: **REALIZADO**.
    - Validación de cabeceras gateway + JWT en `src/middlewares/authMiddlewares.js`.
- Implementar el patrón “circuit breaker” en las comunicaciones con otros servicios: **REALIZADO**.
    - Reconexión con backoff a Kafka en `src/services/kafkaProducer.js`; health expone estado de Kafka en `src/routes/healthRoutes.js`.
- Implementar un microservicio adicional haciendo uso de una arquitectura serverless: **NO REALIZADO**.
- Implementar mecanismos de gestión de la capacidad como throttling o feature toggles: **NO REALIZADO**.
- Cualquier otra extensión al microservicio básico acordada previamente: **REALIZADO**.
    - WebSockets en `src/services/socketService.js` para notificar mensajes en tiempo real.

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.
- 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.
    - El video está accesible aquí.
- 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.
- 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. Tets para comprobar entidades, rutas y servicios. Los tests son accesibles en `test/`.
- 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

Funcional

Gestiona la red social interna:

- **Amistades**: solicitudes de amistad (enviar, listar, aceptar/rechazar, eliminar).
- **Mensajería**: directa entre amigos con paginación y WebSockets.
- **Feed Social**: generado desde eventos de otros microservicios (beats, ratings, comments) y de amigos.

Flujo principal

1. **API Gateway** valida JWT y añade cabeceras `x-gateway-authenticated`/`x-user-id`.
2. **Social** valida cabeceras y permisos.
3. Operación de dominio (amistad, conversación, mensaje, feed) con validaciones de negocio.
4. Persistencia en **Mongo**.
5. **Kafka**: consume eventos externos (users, beats, comments, ratings) y publica eventos `FEED_*` a amigos.
6. **WebSocket**: entrega en tiempo real nuevos mensajes al receptor y emisor.

---

4. Arquitectura y componentes relevantes

Arquitectura de archivos clave

```
social/
├── .dockerignore                                  # Archivos excluidos
del Docker build
├── .env                                           # Variables de entorno
(local)
├── .env.docker-compose.example                   # Template para docker-
compose
├── .env.docker.example                           # Template para Docker
híbrido
├── .env.example                                  # Template para dev local
├── .eslintrc.json                                # Configuración ESLint
├── .gitignore                                    # Archivos excluidos del
repositorio
├── .github/
│   ├── ISSUE_TEMPLATE/
│   │   ├── bug-report.yml                       # Template para bug
reports
│   │   ├── issue-template.yml                   # Template genérico de
issues
│   │   └── security-vulnerability.yml           # Template para
vulnerabilidades
```

```
│   └── workflows/
│       ├── conventional-commits.yml                    # Valida conventional
commits
│       ├── create-releases.yml                         # Crea releases
automáticas
│       ├── linter.yml                                  # Ejecuta linter (ESLint,
Prettier)
│       └── run-tests.yml                               # Ejecuta tests (unit +
integration)
├── .husky/
│   ├── commit-msg                                      # Hook pre-commit message
(commitlint)
│   └── pre-commit                                      # Hook pre-commit (lint +
format)
├── .markdownlint.json                                  # Configuración para
Markdown linting
├── .prettierignore                                     # Archivos excluidos de
Prettier
├── .prettierrc.json                                    # Configuración Prettier
├── .version                                            # Archivo de versión del
app (v0.0.6)
├── .vscode/
│   └── settings.json                                   # Configuración de VS Code
├── CHANGELOG.md                                        # Historial de cambios y
releases
├── Dockerfile                                          # Multi-stage build para
producción
├── Dockerfile-dev                                      # Build para desarrollo
local
├── LICENSE                                             # Licencia del proyecto
├── README.md                                           # Documentación principal
├── commitlint.config.cjs                               # Configuración de
Conventional Commits
├── docker-compose-dev.yml                              # Orquestación Docker
(desarrollo)
├── docker-compose-test.yml                             # Orquestación Docker
(testing)
├── docker-compose.yml                                  # Orquestación Docker
(producción)
├── logger.js                                           # Logger estructurado
(Winston/Pino)
├── main.js                                             # Punto de entrada Express +
Socket.IO
├── package-lock.json                                   # Lock file npm
(dependencias exactas)
├── package.json                                        # Definición de proyecto y
scripts
├── scripts/
│   └── copyEnv.cjs                                     # Script de
copia/inicialización de .env
├── spec/
│   └── oas.yaml                                        # Especificación OpenAPI 3.0
(autogenerada)
├── src/
```

```
│   ├── controllers/
│   │   └── messagingController.js                    # Factory pattern para
controlador mensajería
│   ├── db.js                                         # Función de conexión a
MongoDB
│   ├── middlewares/
│   │   ├── authMiddlewares.js                        # Middleware de
autenticación JWT (futuro)
│   │   └── fakeAuth.js                               # Mock de autenticación para
dev
│   ├── models/
│   │   ├── Beat.js                                   # Schema Beat (materializado
Kafka)
│   │   ├── Conversation.js                           # Schema Conversación
directa
│   │   ├── Feed.js                                   # Schema Evento del feed
│   │   ├── Friendship.js                             # Schema Relación de amistad
│   │   ├── Message.js                                # Schema Mensaje en
conversación
│   │   ├── OASSchemas.js                             # Definiciones OpenAPI
reutilizables (14+ esquemas)
│   │   ├── User.js                                   # Schema Usuario
(materializado Kafka)
│   │   └── models.js                                 # Exportador central de
modelos
│   ├── routes/
│   │   ├── aboutRoutes.js                            # Meta rutas + swagger-jsdoc
config
│   │   ├── feedRoutes.js                             # GET /api/v1/feed (con
JSDoc)
│   │   ├── friendshipRoutes.js                       # 6 endpoints amistad (con
JSDoc)
│   │   ├── healthRoutes.js                           # Health checks públicos
│   │   └── messagingRoutes.js                        # Mensajería + Socket.IO
(JSDoc completo)
│   ├── services/
│   │   ├── feedService.js                            # Lógica: obtener feed
social paginado
│   │   ├── friendHelper.js                           # Helper: validar amistad
entre usuarios
│   │   ├── friendshipService.js                      # Lógica: crear, responder,
listar amistades
│   │   ├── indexes.js                                # Función: crear índices
MongoDB
│   │   ├── kafkaAdmin.js                             # Función: crear topics
Kafka
│   │   ├── kafkaConsumer.js                          # Consumidor Kafka +
materialización + feed
│   │   ├── kafkaProducer.js                          # Productor eventos con
circuit breaker
│   │   └── socketService.js                          # Servidor WebSocket
Socket.IO
│   ├── utils/
│   │   ├── cache.js                                  # Caché en memoria con TTL
```

```
│   │   └── versionUtils.js                          # Helper: leer versión desde
.version
│   └── logger.js                                    # Instancia logger
importable
├── tests/
│   ├── integration/
│   │   ├── .gitkeep                                 # Placeholder directorio
│   │   ├── health.test.js                           # Test endpoints /health
│   │   └── messaging.integration.test.js            # Test flujo completo
mensajería
│   ├── setup/
│   │   ├── integration.setup.js                     # Setup para tests
integración
│   │   └── setup.js                                 # Setup común para tests
│   └── unit/
│       ├── entities.feed.test.js                    # Test modelo Feed
│       ├── entities.friendship.test.js              # Test modelo Friendship
│       ├── messaging.controller.test.js             # Test controlador
mensajería
│       ├── messaging.routes.test.js                 # Test rutas mensajería
│       └── socket.service.test.js                   # Test Socket.IO
├── vitest.config.js                                 # Configuración tests
unitarios
└── vitest.integration.config.js                     # Configuración tests
integración
```

Persistencia (MongoDB)

Colecciones principales:

- `friendships` - Relaciones de amistad (pending, accepted, rejected)
- `conversations` - Conversaciones directas entre usuarios (one-to-one)
- `messages` - Mensajes dentro de conversaciones
- `feeds` - Eventos del feed social por usuario
- `users_materialized` - Vista materializada de usuarios (desde Kafka)
- `beats_materialized` - Vista materializada de beats (desde Kafka)

Seguridad

- **Autenticación**: JWT vía `Authorization: Bearer <token>`
- **Autorización** (ejemplos):
  - Solo el recipient puede aceptar/rechazar una solicitud de amistad.
  - Solo miembros de una conversación pueden ver y enviar mensajes.
  - Se requiere amistad previa para crear conversación.
  - Solo el propietario de una amistad puede eliminarla.
  - Dos usuarios no pueden tener relación de amistad duplicada.

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 con datos de usuarios y beats en feed.
        - Validación de usuarios en solicitudes de amistad.
- **Eventos creados**: Enviamos eventos al microservicio de beats-interaction para que actualice el feed social de usuarios.
    - `FRIENDSHIP_REQUEST_SENT`: Cuando se envía una solicitud de amistad
    - `FRIENDSHIP_ACCEPTED`: Cuando se acepta una solicitud de amistad

---

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 `spec/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.

- `GET /api/v1/docs`

  Interfaz Swagger UI para explorar la especificación OpenAPI interactivamente.

5.2 Health

- `GET /api/v1/health`

  Healthcheck general (API, MongoDB, Kafka, uptime, entorno).

- `GET /api/v1/health/readiness`

  Estado de readiness para Kubernetes (listo para recibir requests).

- `GET /api/v1/health/liveness`

Estado de liveness para Kubernetes (proceso vivo).

5.3 Friendships (Amistades)

Solicitudes de amistad

- `POST /api/v1/friendships`

  Crea solicitud de amistad (requester infiere del token, recipient vía `recipientId`).

  - Response: `Friendship` con status `pending` o `accepted` (si auto-aceptación).
  - Validaciones: requester ≠ recipient, usuario existe.

- `GET /api/v1/friendships/received`

  Lista solicitudes de amistad pendientes recibidas por el usuario autenticado.

  - Response: Array de `Friendship` con status `pending`.

- `GET /api/v1/friendships/sent`

  Lista solicitudes de amistad pendientes enviadas por el usuario autenticado.

  - Response: Array de `Friendship` con status `pending`.

Operaciones sobre solicitudes

- `PATCH /api/v1/friendships/{id}/respond`

  Responde a solicitud de amistad (accept/reject).

  - Request body: `{action: "accept" | "reject"}`
  - Response: `Friendship` con status actualizado.
  - Validación: Solo recipient puede responder.

Amistades aceptadas

- `GET /api/v1/friends`

  Lista amigos aceptados del usuario autenticado.

  - Response: `FriendsResponse` con array de `FriendSummary` (id, username, avatar).

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

  Elimina amistad (uno o ambos sentidos según implementación).

  - Response: `{message: "Friendship removed"}`
  - Validación: Usuario debe ser una de las dos partes.

5.4 Feed Social

- `GET /api/v1/feed?page=0&limit=20&userId=...`

Obtiene feed paginado con eventos sociales del usuario autenticado.

- Query params:
  - `userId`: ObjectId opcional (si no, usa authenticated user)
  - `page`: Número de página (default 0)
  - `limit`: Items por página 1-100 (default 20)
- Response: `PaginatedFeed` con items y metadata (page, limit, count, total).
- Evento types: `friendship`, `FEED_BEAT_CREATED`, `FEED_COMMENT_CREATED`, `FEED_RATING_CREATED`.

5.5 Messaging (Mensajería)

Conversaciones

- `POST /api/v1/social/conversations/direct`

  Crea o recupera conversación directa entre dos usuarios.

  - Request body: `{recipientId: "...", initialMessage?: "..."}`
  - Response: `Conversation` + primer `Message` si se proporciona.
  - Validación: Requiere amistad previa, auto-enriquecimiento con Socket.IO.

- `GET /api/v1/social/conversations`

  Lista conversaciones del usuario autenticado con paginación cursor-based.

  - Query params: `cursor`, `limit`.
  - Response: Array de `ConversationListItem` (incluye `otherUserId` para UI).

Mensajes

- `GET /api/v1/social/conversations/{id}/messages`

  Lista mensajes de conversación con paginación backward (últimos primero).

  - Query params: `cursor`, `limit`.
  - Response: Array de `Message` ordenados por createdAt descendente.
  - Validación: Usuario debe ser miembro de conversación.

- `POST /api/v1/social/conversations/{id}/messages`

  Envía mensaje en conversación.

  - Request body: `{text: "..."}`
  - Response: `Message` recién creado.
  - Side effects:
    - Actualiza `Conversation.lastMessageAt` y `lastMessageText`.
    - Emite vía Socket.IO a ambos usuarios en tiempo real.
  - Validación: Usuario debe ser miembro, conversación debe existir.

---

6. Gestión de errores y validación

Códigos HTTP estándar devueltos por los endpoints:

- **200 OK**: Operación exitosa (GET, PUT, PATCH, DELETE idempotentes).
- **201 Created**: Recurso creado exitosamente (POST).
- **400 Bad Request**:
  - Validaciones fallidas (parámetros inválidos, campos requeridos faltantes).
  - Ejemplos: `userId` inválido, `limit` fuera de rango (1-100), `action` no es "accept"/"reject".
- **401 Unauthorized**:
  - Token JWT missing/invalid.
  - Usuario no autenticado en endpoint protegido.
- **403 Forbidden**:
  - Acceso denegado (p.ej. solo recipient puede responder solicitud).
  - Usuario no es miembro de conversación.
  - No hay amistad previa para crear conversación.
- **404 Not Found**:
  - Recurso inexistente: Friendship, Conversation, Message, User (si Kafka enabled).
  - Beat/User materializado no existe en caché.
- **409 Conflict**:
  - Relación de amistad duplicada (requester + recipient ya existen).
  - Solicitud ya procesada (status ≠ pending).
- **422 Unprocessable Entity**:
  - Validaciones de negocio: requester = recipient (auto-amistad).
  - Kafka enabled y `userId` no existe en `UserMaterialized`.
  - Conversación requiere amistad previa.
- **500 Internal Server Error**:
  - Errores inesperados en procesamiento (DB, Kafka, etc.).
- **503 Service Unavailable**:
  - MongoDB offline.
  - Kafka offline (si `KAFKA_ENABLED=true`).
  - Microservicio en proceso de arranque o shutdown.
