Microservicio “users-service” de 0debt

José Ramón Baños Botón

Sonia María Rus Morales

Esta sección describe cómo el microservicio users-service cumple todos los requisitos exigidos para un microservicio básico, además de cómo se implementan las funcionalidades avanzadas que permiten optar a la máxima calificación para un microservicio avanzado, indicando evidencias concretas en el código según la estructura real del repositorio.

MICROSERVICIO BÁSICO QUE GESTIONE UN RECURSO:

1. **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.**

El microservicio expone una API REST versionada que gestiona autenticación y usuarios, utilizando correctamente los métodos HTTP. Los métodos definidos para un CRUD de usuario los podemos encontrar de la siguiente manera:

Lo podemos ver en la clase: src/routes/auth.ts, en los métodos: POST /auth/register (201,409) y POST /auth/login (200,401,429). Y en la clase src/routes/users.ts en los métodos: GET /users/me (200,401,404), PATCH /users/:id (200,400,401,403,404), DELETE /users/:id (200,400,401,403,404)

2. **La API debe tener un mecanismo de autenticación:**

El microservicio implementa autenticación basada en JWT. Los tokens se generan y firman en el users-service y posteriormente son validados por el API Gateway.

Lo podemos comprobar en la siguiente clase: src/utils/jwt.ts → generación y verificación de JWT. En esta clase: src/routes/auth.ts → emisión del token en login. Y en esta clase: src/middleware/auth.ts → protección de rutas autenticadas

El payload del token incluye sub (userId), email y plan.

3. **Debe tener un frontend que permita hacer todas las operaciones de la API:**

El frontend común a todos los microservicios se encuentra en el siguiente repositorio: https://github.com/0debt/frontend

4. **Debe estar desplegado y ser accesible desde la nube:**

El despliegue se realiza sobre un VPS proporcionado por Hetzner, lo que garantiza un entorno cloud estable y persistente. Sobre este VPS se ha instalado Coolify, que actúa como PaaS auto-hospedada, encargándose de la orquestación de contenedores Docker, la gestión del ciclo de vida del servicio y la exposición segura del microservicio a Internet. (Despliegue en: https://www.0debt.xyz/)

El proceso de despliegue del users-service sigue un flujo automatizado y reproducible:

- El código fuente del microservicio se aloja en un repositorio GitHub independiente, siguiendo buenas prácticas de gestión del código.
- En cada push a la rama principal: Se ejecuta el pipeline de integración continua. Se construye la imagen Docker del microservicio.
- GitHub notifica a Coolify mediante un webhook.
- Coolify: Descarga la nueva versión del código o imagen. Inyecta las variables de entorno necesarias. Levanta el contenedor Docker del users-service. Publica el servicio mediante HTTPS.

5. **La API que gestione el recurso también debe ser accesible en una dirección bien versionada:**

El microservicio implementa un versionado explícito de la API mediante URL, utilizando el prefijo /api/v1. Este enfoque sigue una de las prácticas más comunes y recomendadas en el diseño de APIs REST, especialmente en arquitecturas de microservicios. El archivo clave para contemplar este versionado es el src/index.ts.

6. **Se debe tener una documentación de todas las operaciones de la API incluyendo las posibles peticiones y las respuestas recibidas:**

Se utiliza OpenAPI (Swagger) para proporcionar una documentación formal, estructurada y actualizada automáticamente a partir de la definición centralizada de la API. Se construye a partir de un núcleo OpenAPI centralizado (src/docs/openapi.ts) y la descripción de cada endpoint en src/routes/auth.ts y src/routes/users.ts,

documentando el tipo de método HTTP, su ruta versionada y posibles respuestas y errores.

7. **Debe tener persistencia utilizando MongoDB u otra base de datos no SQL:**

Se utiliza MongoDB Atlas como base de datos principal del microservicio.

En la carpeta src/db/mongo.ts se realiza la conexión a MongoDB.

8. **Deben validarse los datos antes de almacenarlos en la base de datos:**

Los datos de entrada se validan antes de persistirse y se gestionan errores de forma centralizada. Esto se realiza mediante esquemas Zod integrados con OpenAPIHono. Lo podemos observar en el código de la clase src/routes/auth.ts, donde se realizan los endpoints de registro y login. Los esquemas RegisterSchema y LoginSchema definen de forma explícita la estructura y restricciones del request body. OpenAPIHono valida automáticamente la petición entrante contra estos esquemas antes de ejecutar la lógica del endpoint. Si la validación falla, el handler no se ejecuta y se devuelve un error 400 Bad Request, garantizando que ningún dato inválido llegue a la lógica de negocio.

9. **Debe haber definida una imagen Docker del proyecto:**

El microservicio dispone de una imagen Docker propia, definida mediante un Dockerfile en la raíz del proyecto, que permite empaquetar el microservicio junto con su runtime, dependencias y código fuente. La imagen se construye a partir de la imagen oficial oven/bun:latest, garantizando compatibilidad total con el entorno de desarrollo y producción. El Dockerfile define un directorio de trabajo aislado, instala las dependencias utilizando bun.lock para asegurar reproducibilidad, copia el código del microservicio y expone el puerto 3000, desde el cual el servicio atiende peticiones HTTP. La imagen es utilizada directamente en producción por la plataforma de despliegue (Coolify), permitiendo ejecutar el microservicio de forma aislada, reproducible y portable en distintos entornos.

Además, en el Dockerfile se define el estándar de cómo debe construirse, probarse y desplegarse el código. Establece un entorno de ejecución homogéneo que puede ser reutilizado tanto en integración continua como en producción, permitiendo que los pipelines de CI/CD ejecuten la instalación de dependencias, las pruebas automáticas y el arranque del servicio en condiciones idénticas. De este modo, se garantiza que el código validado en los procesos de testing es exactamente el mismo que se despliega

posteriormente, evitando discrepancias entre entornos y reforzando la fiabilidad del ciclo de desarrollo.

10. **Gestión del código fuente: El código debe estar subido a un repositorio de Github siguiendo Github Flow:**

La gestión del código fuente del microservicio se realiza mediante un repositorio independiente al resto de microservicios del proyecto en GitHub, siguiendo el modelo de trabajo GitHub Flow, lo que permite un desarrollo incremental, controlado y fácilmente trazable.

El microservicio se desarrolla sobre una rama principal estable (main), a la que se van integrando cambios de otras ramas de forma progresiva mediante commits frecuentes y descriptivos, facilitando la comprensión de la evolución del código y la identificación de modificaciones concretas. El repositorio está diseñado para ser autocontenible, incluyendo todo lo necesario para construir, ejecutar y desplegar el microservicio, lo que refuerza su independencia dentro de la arquitectura de microservicios. Asimismo, se mantiene una separación clara entre código y configuración, proporcionando un archivo .env.example con las variables necesarias para la ejecución del servicio, mientras que los valores reales y sensibles quedan excluidos del control de versiones mediante .gitignore.

Esta gestión del código permite una integración directa con los procesos de CI/CD, ya que cada cambio versionado puede ser probado, empaquetado en una imagen Docker y desplegado automáticamente.

11. **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:**

La integración continua del microservicio se implementa mediante GitHub Actions, permitiendo que el código se compile y se pruebe automáticamente en cada cambio realizado sobre el repositorio. Este mecanismo garantiza que cada modificación del código pasa por un proceso de validación automática antes de considerarse estable, reduciendo errores y asegurando la calidad del microservicio. El pipeline de integración continua está configurado para ejecutar de forma automática la instalación de dependencias y las pruebas de componente utilizando Bun, el mismo runtime empleado en desarrollo y producción, lo que asegura coherencia entre entornos. Además, este

proceso se integra de forma natural con la generación de la imagen Docker del microservicio, de manera que únicamente el código que supera correctamente las pruebas puede ser empaquetado y desplegado.

12. **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:**

El microservicio incluye un conjunto de pruebas de componente automáticas que permiten verificar su correcto funcionamiento de forma aislada. Estas pruebas se ejecutan utilizando el test runner de Bun, el mismo runtime empleado en el desarrollo y la ejecución del servicio, lo que garantiza coherencia entre los entornos de prueba y producción. Los tests se encuentran organizados dentro de la carpeta src/tests/ y están diseñados para cubrir tanto escenarios positivos como negativos. Los escenarios positivos validan los flujos normales de uso del sistema, como el registro y login correctos, el acceso autenticado a endpoints protegidos, el correcto funcionamiento del health check y la comunicación exitosa con servicios externos. Los escenarios negativos comprueban el comportamiento del sistema ante situaciones de error, incluyendo credenciales inválidas, accesos no autorizados, abuso del endpoint de login mediante throttling, fallos en servicios externos y activación del Circuit Breaker.

MICROSERVICIO AVANZADO QUE GESTIONE UN RECURSO:

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

Se utiliza Redis como sistema de caché para optimizar el acceso a los datos internos de usuario que son consumidos de forma frecuente por otros microservicios del sistema. Esta caché se aplica específicamente en el endpoint interno GET /api/v1/internal/users/:id de la clase src/routes/users.ts, cuyo objetivo es proporcionar información básica del usuario (identificador, nombre, email, avatar y plan) sin exponer datos sensibles como la contraseña o los add-ons al microservicio de groups-service. Al tratarse de un endpoint interno y de alto consumo, se ha priorizado su rendimiento mediante el uso de un mecanismo de caché distribuida.

La implementación sigue el patrón cache-aside, donde el microservicio consulta primero Redis antes de acceder a la base de datos. Para cada petición, se genera una clave de caché con el formato user:{id}. Si Redis está disponible y la clave existe, el microservicio devuelve directamente la respuesta almacenada en caché, evitando así una consulta innecesaria a MongoDB. En caso de que el dato no esté cacheado, el servicio accede a la base de datos, construye la respuesta filtrando los campos sensibles y almacena el resultado en Redis con un tiempo de vida de 60 segundos antes de devolverlo al consumidor.

Este diseño garantiza una mejora significativa del rendimiento en escenarios de alta lectura, reduce la carga sobre MongoDB y mantiene la coherencia de los datos al tratarse de información de lectura frecuente y bajo nivel de mutación. Además, el uso de Redis es opcional y tolerante a fallos, ya que el microservicio comprueba previamente si la conexión a Redis está disponible. En caso de indisponibilidad, el endpoint continúa funcionando correctamente accediendo directamente a la base de datos, evitando dependencias rígidas y fallos en cascada.

2. **Consumir alguna API externa a través del backend o algún otro tipo de almacenamiento de datos en cloud como Amazon S3:**

Se utiliza Supabase como servicio de almacenamiento cloud para la actualización y almacenamiento del avatar del usuario, desacoplando completamente este tipo de contenido del propio microservicio y de la base de datos MongoDB. La interacción con Supabase se realiza exclusivamente desde el backend del users-service, encapsulada en un cliente dedicado, lo que garantiza que las credenciales y la lógica de acceso al servicio externo no queden expuestas al frontend. Cuando un usuario modifica su avatar, el microservicio gestiona la subida de la imagen a Supabase y almacena únicamente la URL resultante en la base de datos del usuario.

Además del uso de Supabase como sistema de almacenamiento cloud, el microservicio consume una API externa pública para la generación automática del avatar del usuario en el momento de su creación. Concretamente, se utiliza la API de DiceBear (https://api.dicebear.com), un servicio externo que permite generar avatares dinámicos a partir de un identificador o semilla.

Esta integración se realiza íntegramente desde el backend durante el proceso de registro del usuario, dentro del endpoint POST /api/v1/auth/register, implementado en el archivo src/routes/auth.ts. Cuando se crea un nuevo usuario, el microservicio construye una URL de avatar utilizando la API de DiceBear, empleando como semilla el nombre del

usuario o, en su defecto, su dirección de correo electrónico. Esta URL apunta a un recurso SVG generado dinámicamente por el servicio externo y se asigna directamente al campo avatar del usuario.

3. **Implementar un mecanismo de autenticación basado en JWT o equivalente:**

El microservicio implementa un mecanismo de autenticación basado en JWT, actuando como el emisor oficial de identidad del sistema dentro de la arquitectura de microservicios. Este microservicio es responsable de generar y firmar los tokens JWT tras un proceso de autenticación exitoso, concretamente en el endpoint de login. La lógica de generación y firma del token se encuentra centralizada en el archivo src/utils/jwt.ts.

El token JWT generado incluye en su payload información relevante del usuario, como su identificador (sub), su correo electrónico y el plan de suscripción asociado (FREE, PRO o ENTERPRISE). Esta información se incorpora directamente en el token para permitir que otros componentes del sistema, como el API Gateway o los propios microservicios de negocio, puedan tomar decisiones de autorización sin necesidad de realizar consultas adicionales al users-service. Los endpoints protegidos del microservicio validan la presencia y validez del token mediante un middleware de autenticación definido en src/middleware/auth.ts, que se encarga de verificar la firma, la estructura del token y su vigencia antes de permitir el acceso a la lógica de negocio.

4. **Implementar el patrón "circuit breaker" en las comunicaciones con otros servicios:**

Se implementa el patrón de Circuit Breaker con el objetivo de proteger la comunicación con servicios externos y evitar fallos en cascada dentro de la arquitectura de microservicios. En concreto, este patrón se aplica a la interacción con el microservicio de notifications-service, que es invocado durante el proceso de registro de usuarios para inicializar sus preferencias de notificación. Dado que este servicio externo no forma parte del flujo crítico de autenticación, su indisponibilidad no debe comprometer la creación de nuevos usuarios ni la disponibilidad general del sistema.

La implementación del Circuit Breaker se encuentra encapsulada en el archivo src/lib/circuitBreaker.ts, donde se define la lógica encargada de monitorizar los fallos consecutivos en las llamadas externas y de cambiar dinámicamente el estado del circuito (CLOSED, OPEN). Este mecanismo permite que, ante un número elevado de

errores o tiempos de respuesta anómalos, el circuito se abra y se eviten nuevas llamadas al servicio externo durante un periodo de tiempo determinado. De este modo, el users-service deja de consumir recursos en peticiones destinadas a fallar y protege su propio rendimiento y estabilidad.

El Circuit Breaker se utiliza de forma efectiva en src/lib/notificationClient.ts, que actúa como cliente HTTP del notifications-service. Cuando el circuito está cerrado y el servicio externo responde correctamente, las llamadas se realizan con normalidad. En cambio, cuando el circuito se encuentra abierto, el cliente ejecuta un fallback seguro, permitiendo que el flujo principal continúe sin bloquear el registro del usuario. Esta estrategia garantiza que la creación de cuentas no dependa de la disponibilidad de servicios auxiliares.

5. **Implementar mecanismos de gestión de la capacidad como throttling para rendimiento:**

Se implementa un mecanismo de gestión de la capacidad mediante throttling con el objetivo de proteger el sistema frente a abusos y mejorar su rendimiento y seguridad, especialmente en uno de los endpoints más críticos de la aplicación: en el endpoint POST /auth/login, dentro del archivo src/routes/auth.ts, utilizando Redis como contador distribuido. En cada intento de login, el microservicio incrementa un contador asociado al email del usuario (login_attempts:{email}) y establece un tiempo de expiración de 60 segundos en el primer intento. Si el número de intentos supera el umbral definido (más de 5 intentos en un minuto), el microservicio bloquea temporalmente el acceso y devuelve una respuesta 429 Too Many Requests, impidiendo nuevos intentos de autenticación hasta que el contador expira.

La implementación está diseñada de forma tolerante a fallos, ya que el uso de Redis es opcional y no constituye una dependencia rígida. En caso de que Redis no esté disponible o se produzca un error en la comunicación, el endpoint de login continúa funcionando con normalidad, priorizando la disponibilidad del servicio sobre el mecanismo de protección. Este diseño evita fallos en cascada y garantiza que el throttling actúe como una capa adicional de seguridad y rendimiento, pero nunca como un punto único de fallo.

6. **Implementar mecanismos de gestión de la capacidad como feature toggles para rendimiento:**

Se implementa el mecanismo de feature toggles en el backend para habilitar o restringir funcionalidades en función del plan de suscripción del usuario, permitiendo gestionar la capacidad y el acceso a características avanzadas sin necesidad de redesplegar el microservicio. Este enfoque se aplica de forma explícita en la funcionalidad de subida y actualización del avatar del usuario, que está disponible únicamente para usuarios con planes PRO o ENTERPRISE.

La restricción de esta funcionalidad se realiza mediante el middleware requirePlan, aplicado directamente a la ruta PATCH /api/v1/users/:id/avatar. Este middleware evalúa el plan del usuario extraído del JWT previamente validado, y bloquea el acceso a la ruta si el plan no se encuentra dentro de los permitidos. De este modo, un mismo endpoint puede existir en producción para todos los usuarios, pero su ejecución queda condicionada dinámicamente al nivel de suscripción, actuando como un feature toggle controlado desde el backend. Los usuarios con plan FREE reciben una respuesta de tipo 403 Forbidden, mientras que los usuarios con planes superiores pueden acceder a la funcionalidad sin cambios adicionales en el código o en la infraestructura.

Este diseño permite que la funcionalidad de subida de avatar se active o desactive de forma inmediata mediante cambios en el plan del usuario, sin necesidad de modificar el microservicio ni desplegar nuevas versiones. Además, la lógica del feature toggle se integra de forma transparente con el resto del flujo del endpoint, que incluye validaciones de seguridad adicionales como la comprobación de identidad del usuario, la validación del tipo y tamaño del archivo y el almacenamiento del avatar en Supabase, guardando únicamente la URL pública resultante en MongoDB.

---

* **Gestión de add-ons del usuario (implementado, pero no utilizado en la aplicación final)**

Dentro del microservicio users-service se ha implementado el endpoint PATCH /api/v1/users/:id/addons, cuyo objetivo es permitir la asignación y modificación de add-ons asociados a un usuario, extendiendo el modelo de suscripción más allá del plan base (FREE, PRO, ENTERPRISE). El endpoint cuenta con autenticación basada en JWT, validación de datos de entrada mediante esquemas, control de permisos y persistencia en MongoDB, siguiendo los mismos criterios de calidad y seguridad que el resto de la API. La lógica se encuentra integrada en el fichero src/routes/users.ts, donde se valida la identidad del usuario y se actualiza el conjunto de add-ons asociados a su perfil.

No obstante, aunque esta funcionalidad está completamente implementada y operativa a nivel de microservicio, finalmente no se ha utilizado ni explotado en la aplicación grupal 0debt. Esta decisión se tomó durante la fase de integración global de la aplicación, priorizando otras características avanzadas y manteniendo el modelo de suscripción únicamente basado en planes, sin consumo efectivo de add-ons desde el frontend común. Aun así, el endpoint permanece disponible y documentado como una extensión preparada del sistema, demostrando la capacidad del microservicio para soportar este requisito avanzado, aunque no haya sido activado en la aplicación final.