Noctua Sapienta - Review Service

FUNDAMENTOS DE INGENIERÍA DEL SOFTWARE PARA SISTEMAS CLOUD

Realizado por

José Calderón Valdivia

Paula Yuan César Aguilar

Profesores

Manuel Resinas Arias de Reyna

Antonio Ruiz Cortés

Máster Universitario en Ingeniería del Software: Cloud, Datos y Gestión TI

Curso 2023/24

Índice general

Índice de figuras

1. Nivel de acabado elegido

Optamos al nivel de acabado **Hasta 9 puntos**, habiendo seleccionado 8 características de microservicio avanzado y las 3 de aplicación basada en microservicios avanzada, es decir, parte del nivel **Hasta 10 puntos**. A continuación se detallan.

1.1. Características de microservicio avanzado que gestione un recurso elegidas

Se han elegido las siguientes características de microservicio avanzado:

1. Implementar un frontend con rutas y navegación.
2. Consumo de algún API externa (distinta de las de los grupos de práctica).
3. Tener el API REST documentado con swagger.
4. Implementación de un mecanismo de autenticación basado en JWT o equivalente.
5. Implementar el patrón "circuit breaker" en las comunicaciones con otros servicios.
6. Implementar un microservicio adicional haciendo uso de una arquitectura serverless (Function-as-a-Service).
7. Implementar mecanismos de gestión de la capacidad como throttling o feature toggles.
8. Implementar paginación en las peticiones a la base de datos.

Además se ha realizado un total de 59 pruebas unitarias y un análisis de la capacidad y un SLA como extensión al Customer Agreement.

1.2. Características de aplicación avanzada que gestione un recurso elegidas

Se han elegido las siguientes características de aplicación avanzada:

1. Tener un front end común que integre los front ends de cada uno de los microservicios. Cada pareja debe ocuparse, al menos, de la parte específica de su microservicio en el front end común.
2. Implementación de un mecanismo de autenticación homogéneo para todos los microservicios.
3. Implementación de un User Acceptance Test (UAT).

2. Descripción de la aplicación

La aplicación desarrollada en este proyecto, llamada *Noctua Sapienta*, presenta una web de venta de libros, en la que se aborda desde el control de los libros en venta por parte de un vendedor hasta la gestión de los pedidos de los usuarios compradores de un libro, pasando por la creación de reseñas o valoraciones para los diferentes libros y vendedores.

Se ha utilizado arquitectura de microservicios para su desarrollo, entre los que podemos distinguir:

- Usuarios
- Libros
- Pedidos
- Reseñas

En nuestro caso, nos encargamos del **microservicio de reseñas**, que se encarga de manejar los dos tipos de reseñas posibles en la aplicación, de libros o de vendedores. A continuación se detallan las principales funcionalidades que puede se pueden llevar a cabo en la aplicación:

- Un vendedor puede consultar las reseñas que los clientes le han puesto
- Un cliente puede consultar todas las reseñas que ha puesto, ya sean de libros o de vendedores
- Un usuario cualquiera puede consultar las reseñas que los clientes le han puesto a un libro o a un vendedor
- Cualquier cliente puede poner una reseña de un libro, sin necesidad de haberlo comprado
- Un cliente sólo puede valorar a un vendedor si le realizado algún pedido con anterioridad

3. Justifiación de la consecución de los requisitos

3.1. Microservicio básico que gestione un recurso

3.1.1. El backend debe ser un 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

La funcionalidad de nuestra API se encuentra repartida entre los archivos /routes/bookReviews.js y /routes/sellerReviews.js de nuestro repositorio.

En cada uno de los endpoint se devuelve un código de estado dependiendo del resultado de la operación.

3.1.2. La API debe tener un mecanismo de autenticación

Se utiliza un mecanismo de autenticación basado en JWT, unificado para todos los microservicios y el frontend. La definición de los métodos que se encargan de la autenticación en las llamadas a la API se encuentra en /middleware/validateJWT.js.

3.1.3. 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)

Este frontend común puede encontrarlo en su repositorio:

https://github.com/Noctua-sapientia/frontend

3.1.4. La API que gestione el recurso debe ser accesible en una dirección bien versionada

Se ha añadido un versionado a la dirección base de la API, como puede verse en el archivo app.js.

Por lo que para acceder a los recursos, bookReview y sellerReview, se accederá mediante las rutas:

- /api/v1/reviews/books en el caso de las valoraciones de libros

- /api/v1/reviews/sellers en el caso de que nos refiramos a las valoraciones de los vendedores

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

Se puede encontrar la documentación en:

https://app.swaggerhub.com/apis/TEAMREVIEW24/ReviewsAPI/1.0.0

3.1.6. Debe tener persistencia utilizando MongoDB u otra base de datos no SQL

Como puede comprobarse en db.js, se emplea una base de datos MongoAtlas.

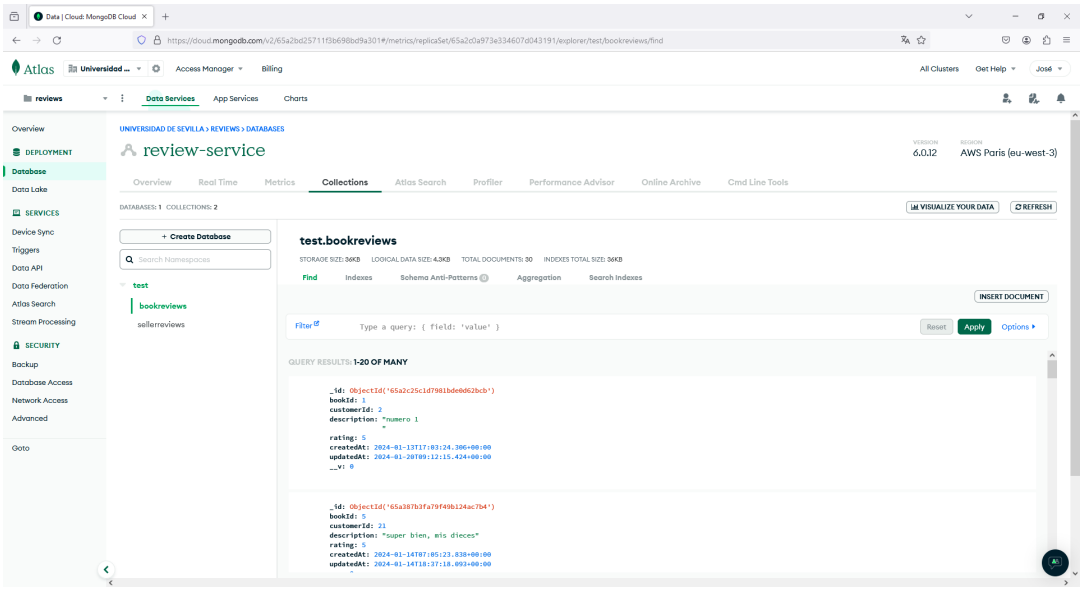

Figura 3.1: Base de datos Mongo Atlas

3.1.7. Deben validarse los datos antes de almacenarlos en la base de datos (por ejemplo, haciendo uso de mongoose)

Se han definido las restricciones de validación en los modelo de la base de datos, bookReview.js y sellerReview.js, a través de moongose.

```
const bookReviewSchema = new mongoose.Schema({
    bookId: {
        type: Number,
        required: true
    },
    customerId: {
        type: Number,
        required: true
    },
    description: {
        type: String,
        required: true
    },
    rating: {
        type: Number,
        required: true
    }
},
{
    // This option assigns "createdAt" and "updatedAt" to the schema:
    // https://stackoverflow.com/questions/12669615/add-created-at-and-updated-at-fields-to-mongoose-schemas
    timestamps: true
});
```

Figura 3.2: Esquema de reviews de libros

```
const sellerReviewSchema = new mongoose.Schema({
    sellerId: {
        type: Number,
        required: true
    },
    customerId: {
        type: Number,
        required: true
    },
    description: {
        type: String,
        required: true
    },
    rating: {
        type: Number,
        required: true
    }
},
{
    // This option assigns "createdAt" and "updatedAt" to the schema:
    // https://stackoverflow.com/questions/12669615/add-created-at-and-updated-at-fields-to-mongoose-schemas
    timestamps: true
});
```

Figura 3.3: Esquema de reviews de vendedores

3.1.8. Se debe utilizar gestión del código fuente y mecanismos de integración continua: El código debe estar subido a un repositorio de Github siguiendo Github flow. El código debe compilarse y probarse automáticamente usando GitHub Actions en cada commit

En el repositorio de GitHub se puede ver cómo se ha seguido GitHub Flow y cómo se ha configurado un Action para ejecutar las pruebas unitarias y de integración del microservicio cada vez que se realice un commit.

3.1.9. Debe haber definida una imagen Docker del proyecto

Se encuentra definida en el archivo Dockerfile del proyecto:

Code Blame 21 lines (15 loc) · 292 Bytes

```
1   FROM node:14-alpine
2
3   WORKDIR /app
4
5   COPY package.json .
6   COPY package-lock.json .
7
8   RUN npm install
9
10  COPY bin/ ./bin
11  COPY public ./public
12  COPY routes/ ./routes
13  COPY services/ ./services
14  COPY models/ ./models
15  COPY middlewares/ ./middlewares
16  COPY app.js .
17  COPY db.js .
18
19  EXPOSE 4004
20
21  CMD npm start
```

Figura 3.4: Dockerfile

3.1.10. Debe haber pruebas unitarias implementadas en Javascript para el código del backend utilizando Jest (el usado en los ejercicios) o Mocha y Chai o similar. Como norma general debe haber tests para todas las funciones no triviales de la aplicación. Probando tanto escenarios positivos como negativos

Se han implementado 59 tests unitarios, como puede comprobarse en el log de la ejecución de Actions:

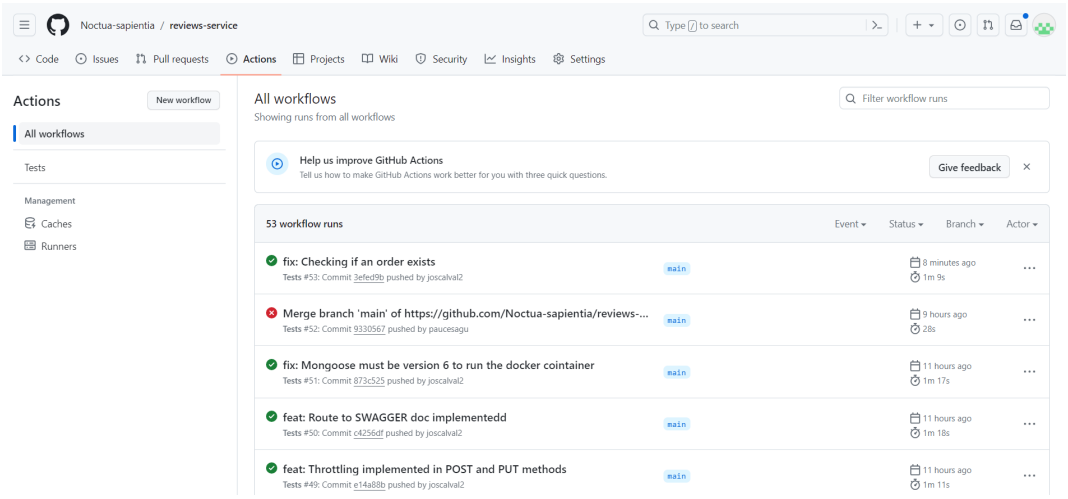

Figura 3.5: Actions Github

En cuanto a los test unitarios, podemos ver, por ejemplo, el del caso de éxito de POST en la base de datos, donde vemos cómo se mockean algunos métodos al principio, se realiza la llamada y se realizan las distintas comprobaciones posteriores:

```
it("Should add a new book review if everything is fine", () => {
    dbExists.mockImplementation(async () => Promise.resolve(false));
    dbSave.mockImplementation(async () => Promise.resolve(true));
    existsBook.mockImplementation(async () => Promise.resolve(true));
    dbAggregate.mockImplementation(async () => Promise.resolve([{ _id: null, averageRating: 4.2 }]));
    updateRatingBook.mockImplementation(async () => Promise.resolve(true));
    containsInsult.mockImplementation(async () => Promise.resolve("False"));

    return request(app).post("/api/v1/reviews/books").send(bookReviewJSON).set('Authorization', jwtToken).then((response) => {
        expect(response.statusCode).toBe(201);
        expect(dbExists).toBeCalled();
        expect(dbSave).toBeCalled();
        expect(existsBook).toBeCalled();
        expect(dbAggregate).toBeCalled();
        expect(updateRatingBook).toBeCalled();
        expect(response.body.customerId).toBe(bookReviewJSON.customerId);
        expect(response.body.description).toBe(bookReviewJSON.description);
        expect(response.body.rating).toBe(bookReviewJSON.rating);
        expect(response.body.bookId).toBe(bookReviewJSON.bookId);
    });
});
```

Figura 3.6: Test método POST

Mención especial merece también la manera de mockear los métodos de conexión con la base de datos de Mongoose, ya que debido a la implementación de la paginación y la adaptación para manejo de Big Data requerían una manera de utilización de Jest que se escapaba a la documentación. Dicho mecanismo consiste en utilizar cadenas:

```
describe("GET /reviews/sellers", () => {
    it("Should return all seller reviews", () => {

        const mockFindChain = {
            sort: jest.fn().mockReturnThis(),
            limit: jest.fn().mockReturnThis(),
            skip: jest.fn().mockImplementation(() => Promise.resolve(sellerReviews)),
        };

        dbFind = jest.spyOn(SellerReview, 'find').mockImplementation(() => mockFindChain);

        return request(app).get("/api/v1/reviews/sellers").set('Authorization', jwtToken).then((response) => {
            expect(response.statusCode).toBe(200);
            expect(response.body).toBeArrayOfSize(sellerReviews.length);
            expect(response.body[0].id).toBe(sellerReviews[0].id);
            expect(response.body[0].customerId).toBe(sellerReviews[0].customerId);
            expect(response.body[0].description).toBe(sellerReviews[0].description);
            expect(response.body[0].rating).toBe(sellerReviews[0].rating);
            expect(response.body[0].sellerId).toBe(sellerReviews[0].sellerId);
            expect(dbFind).toBeCalled();
        });
    });
```

Figura 3.7: Test método GET con cadenas

3.1.11. Debe haber pruebas de integración con la base de datos

Se ha realizado diversas pruebas que verifica la correcta integración con la base de datos mongo. Concretamente han sido 14 en las que se ha verificado que:

- Se almacenan reviews
- Se obtiene una review por ID
- Se obtienen las reviews con un determinado bookId
- Se obtienen las reviews con un determinado sellerId
- Se obtienen las reviews con un determinado customerId
- Se obtienen todas las reviews
- Se actualiza una determinada review
- Se delimita una determinada review

El código puede consultarse en https://github.com/Noctua-sapientia/reviews-service/blob/main/tests/integration/db-integration.test.js

3.2. Microservicio avanzado que gestione un recurso

3.2.1. Implementar un frontend con rutas y navegación

Puede comprobarse de forma práctica en el despliegue del frontend. Puede comprobar que existen rutas y la navegación es obvia con los botones de los menús:

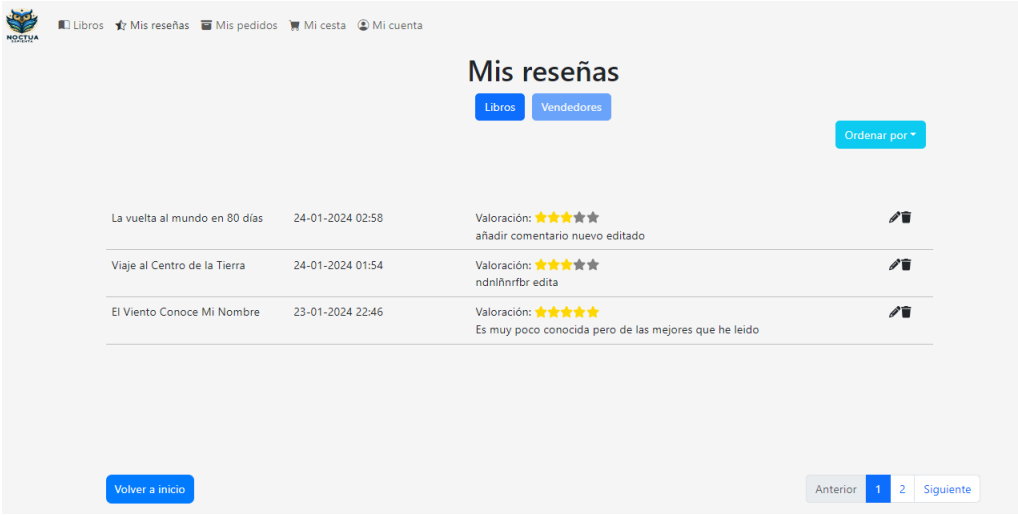

Figura 3.8: Pantalla de sección "Mis reseñas"

El código puede encontrarse en: https://github.com/Noctua-sapientia/frontend

3.2.2. Consumo de algún API externa (distinta de las de los grupos de práctica)

Se ha utilizado la API de Mailgun para enviar un correo a un usuario cuando realiza una reseña que contenga algún insulto. Dicho código puede consultarse en services/emailService.js.

En la parte del servidor se realiza el envío del correo cuando la reseña contiene algún insulto y además le aparece un aviso en la pantalla al usuario:

Figura 3.9: Comentario no válido

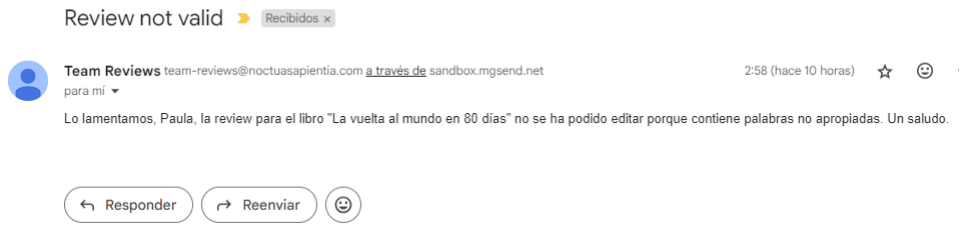

Figura 3.10: Correo explicando detalles de la inadecuación del comentario

3.2.3. Tener el API REST documentado con swagger

Se puede encontrar la documentación en SWAGGER en:

https://app.swaggerhub.com/apis/TEAMREVIEW24/ReviewsAPI/1.0.0

3.2.4. Implementación de un mecanismo de autenticación basado en JWT o equivalente

Se ha implementado JWT de forma homogénea en todos los microservicios y frontend, explicado la Sección 3.1.2 de este documento.

3.2.5. Implementar el patrón "circuit breaker" en las comunicaciones con otros servicios

Se ha implementado el patrón circuit breaker en la comunicación con el microservicio de libros. Cuando se añade una reseña de un libro con su valoración, se calcula la valoración del mismo haciendo una media de todas las asociadas a él. De esta forma en el caso de que se supere el umbral de errores al hacer el método PUT, no sobrecargaremos el servicio de libros. Puede consultar el código en middlewares/circuitBreakerPattern.js

3.2.6. Implementar un microservicio adicional haciendo uso de una arquitectura serverless (Function-as-a-Service)

Se ha implementado un microservicio con arquitectura FaaS para comprobar si una reseña contiene un insulto. Para ello se ha utilizado Azure Functions donde hemos creado una función que dada una lista de palabras ofensivas nos devuelva si el comentario recibido contiene alguna de esas palabras:

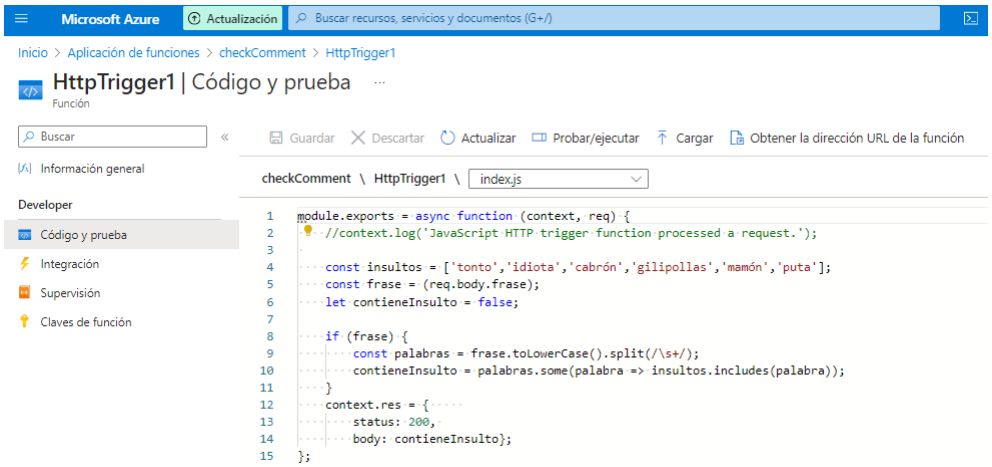

Figura 3.11: Código de implementación de la función

El código de la conexión con Azure se puede consultar en services/checkComment.js

3.2.7. Implementar mecanismos de gestión de la capacidad como throttling o feature toggles

Se ha implementado un mecanismo de trhottling para evitar que un usuario pueda realizar o editar más de 50 reseñas por minuto, debido a que estas llamadas a la API generan bastantes llamadas hacia otros microservicios y así se reduce la posibilidad de colapso del sistema.

El código de la implementación del throttling puede consultarse en middlewares/ratelimit.js

3.2.8. Implementar paginación en las peticiones a la base de datos, apta para Big Data

En las peticiones GET se han implementado las opciones de paginación, tales como elegir el número de elementos para traer de la base de datos, el orden de los mismos o el criterio de ordenación.

La capacidad de tratamiento para Big Data radica en que se han utilizado todas las funciones de Mongoose que permiten realizarlo, de modo que es la base de datos Mongo la que responde con los datos que se le piden.

```
router.get('/', validateJWT, async function(req, res, next) {
  try {

    if ( !validateSortField(req.query.sort) ) { return res.status(400).send("Invalid sort field. It must be 'rating' or 'date'."); }
    let sortat;
    if (req.query.sort) {
      sortat = req.query.sort === 'date' ? 'createdAt' : 'rating';
    } else {
      sortat = 'createdAt';
    }

    if ( !validateOrderField(req.query.order) ) { return res.status(400).send("Invalid order field. It must be 'asc' or 'desc'."); }
    let order = req.query.order !== undefined ? req.query.order : 'desc';

    let filters = {};

    if (req.query.bookId !== undefined) { filters["bookId"] = req.query.bookId; }

    if (req.query.customerId !== undefined) { filters["customerId"] = req.query.customerId; }

    if ( !validateLimit(req.query.limit) ) { return res.status(400).send("Limit must be a number greater than 0.") }
    let limit = req.query.limit === undefined ? null : req.query.limit;

    if ( !validateOffset(req.query.offset) ) { return res.status(400).send("Offset must be a non-negative number") }
    let offset = req.query.offset === undefined ? null : req.query.offset;

    const result = await BookReview.find(filters).sort([[sortat, order]]).limit(limit).skip(offset);
    if (result.length > 0) {
      res.status(200).send(result.map((r) => r.cleanup()));
    } else {
      res.status(404).send({error: `Review not found.`});
    }
  } catch(e) {
    debug('DB  problem', e);
    res.sendStatus(500);
  }
});
```

Figura 3.12: Consulta GET optimizadas con Mongoose

3.3. Aplicación basada en microservicios básica

3.3.1. Interacción completa entre todos los microservicios de la aplicación integrando información

Se puede comprobar esta integración desplegando el frontend y utilizando la aplicación en su conjunto.

3.4. Aplicación basada en microservicios avanzada

3.4.1. Tener un front end común que integre los front ends de cada uno de los microservicios. Cada pareja debe ocuparse, al menos, de la parte específica de su microservicio en el front end común

Se puede comprobar esta integración desplegando el frontend o a su repositorio y utilizando la aplicación en su conjunto. Por ejemplo, en la siguiente imagen podemos ver como se han integrado los distintos microservicios, el libro con su detalle, las valoraciones asociadas al mismo, la opción de añadir al carrito para crear un pedido y los vendedores posibles.

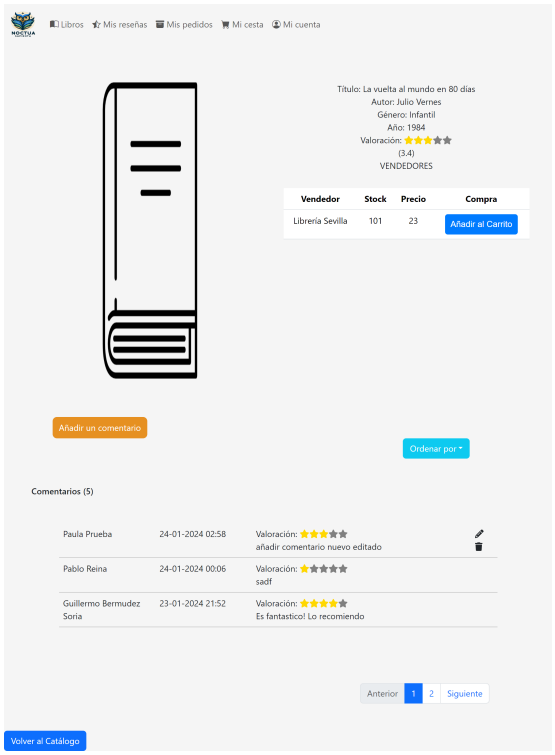

Figura 3.13: Detalle del libro con componentes de distintos microservicios

Se han implementado los archivos que se encuentran dentro de la carpeta src/-components/Review del proyecto del frontend.

También, hemos implementado los archivos src/api/ReviewsApi.js, que permite la conexión del frontend con el microservicio de reseñas y el src/setupProxy.js que permite que el frontend pueda enviar peticiones a distintos microservicios.

3.4.2. Definición de un customer agreement para la aplicación en su conjunto

Se ha definido un acuerdo a nivel de servicio global para la aplicación. Puede consultarse enç Customer Agreement. También se pueden consultar un análisis de la capacidad y un SLA.

3.4.3. Implementación de un mecanismo de autenticación homogéneo para todos los microservicios

De nuevo, se ha implementado JWT de forma homogénea en todos los microservicios y frontend, explicado en la Sección 3.1.2 de este documento.

3.4.4. Realización de un User Acceptance Test (UAT)

Se ha realizado un User Acceptance Test (UAT) para validar que el sistema cumple con los requisitos y necesidades del usuario final. Este proceso es crucial para asegurar que el software o producto es funcional y usable en un entorno de producción real. Durante el UAT, los usuarios finales o representantes de los usuarios prueban el sistema para verificar que todas las especificaciones y funcionalidades acordadas están presentes y trabajando correctamente. Este paso es uno de los últimos antes de la implementación final y ayuda a identificar cualquier problema o deficiencia que pueda haber pasado desapercibida durante las fases de desarrollo y prueba anteriores.

Puede consultarse en UAT.

4. Análisis de los esfuerzos

A continuación se detallan en una tabla el tiempo aproximado dedicado por cada miembro del grupo a la realización del proyecto.

Alumno	Desarrollo	Documentación	Demo	Total
José Calderón Valdivia	85	20	4	109
Paula Yuan César Aguilar	96	12	4	111

Cuadro 4.1: Tiempo empleado en horas

En estos tiempos se excluye el dedicado al seguimiento de los vídeos del profesor, al igual que el tiempo dedicado las reuniones del equipo se incluye en **Desarrollo**.

Un gráfico en el que puede verse la media de horas invertidas en el proyecto a lo largo del tiempo es el siguiente:

Figura 4.1: Horas invertidas

También podemos visualizar diagramas de tarta del esfuerzo empleado por cada miembro del equipo de desarrollo en el backend y testing o en el frontend:

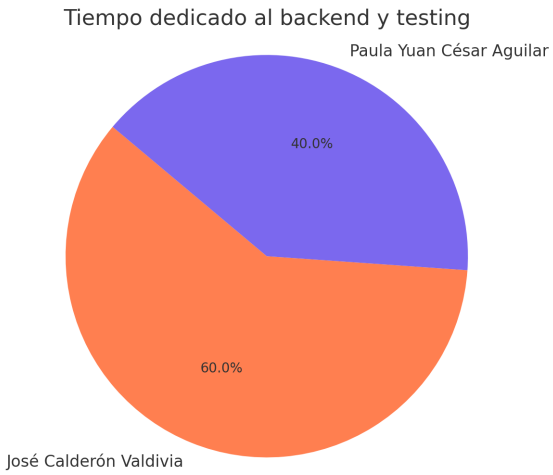

Figura 4.2: Porcentaje desarrollo backend

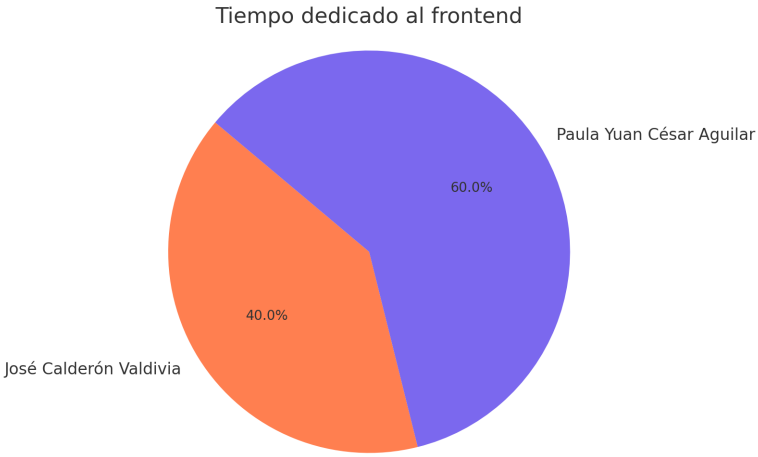

Figura 4.3: Porcentaje desarrollo frontend

5. Conclusiones

Durante el proyecto, hemos abordado el desafío de desarrollar un microservicio de reseñas, utilizando tecnologías de vanguardia como Node.js, Express, React y MongoDB. El objetivo principal fue integrar este microservicio con los otros desarrollados por el resto del equipo para lograr un sistema completo y eficiente.

Se ha podido comprobar la importancia de una arquitectura de microservicios para facilitar la escalabilidad, la independencia y la eficiencia en el desarrollo y despliegue de aplicaciones. La elección de Node.js y Express para el backend permitió una implementación rápida y eficiente de API RESTful, mientras que React en el frontend ofreció una experiencia de usuario dinámica y atractiva. Además, hay que destacar que el uso de contenedores Docker ha permitido encapsular nuestro microservicio, junto con sus dependencias y configuraciones, en entornos autocontenidos. Simplificando así, el proceso de despliegue y garantizando que se pueda ejecutar en cualquier entorno.

La base de datos MongoDB se seleccionó por su capacidad de manejar datos no estructurados y grandes volúmenes de estos, lo que resulta fundamental en un sistema de reseñas donde la información puede variar en formato y longitud. La flexibilidad de MongoDB se alinea bien con la naturaleza evolutiva de los microservicios.

La integración de este microservicio con otros desarrollados por diferentes equipos presentó desafíos de comunicación y estandarización de datos. Se implementaron prácticas como la documentación detallada de la API en SWAGGER para garantizar la interoperabilidad entre los distintos microservicios.

También cabe mencionar la exhaustividad de los tests realizados, que nos ha permitido compreneder cómo se realiza el proceso de testing cuando hay interación con servicios externos y con la base de datos, con los cuáles hemos tenido que tratar casuísticas muy diversas.

Como resultado, hemos logrado implementar con éxito un microservicio de reseñas funcional que cumple con los requisitos establecidos.

En conclusión, este proyecto no solo ha fortalecido nuestra comprensión de las tecnologías seleccionadas, sino que también ha resaltado la importancia de los microservicios como una arquitectura poderosa y flexible, permitiendo una evolución continua y un desarrollo ágil en equipos diversos.