Universidad de las Fuerzas Armadas ESPE

Departamento de Ciencias de la Computación
Carrera de Tecnologías de la Información – En línea

Arquitectura de Software

Desarrollo de una aplicación basada en arquitectura de microservicios para la Gestión de Autores y Publicaciones

Estudiantes:
Rodríguez Salas Betty Lizeth
Víctor Hugo Villamarín López

Docente: Ing. Geovanny Cudco

Repositorio:
https://github.com/saoricoder/GestionAutor_Publicacion.git

Fecha Entrega:
04/02/2026

Índice del Documento Técnico: Sistema Editorial

Índice Figuras

Índice Tablas

1. Introducción y Contexto

Objetivo del Proyecto.
Desarrollar una solución funcional basada en una **arquitectura de microservicios** (Autores y Publicaciones) que permita gestionar el ciclo de vida editorial de manera desacoplada. El proyecto integra el modelado y simulación de procesos mediante **BPMN en Camunda**, la aplicación de principios **SOLID** y patrones de diseño, asegurando un despliegue reproducible y orquestado a través de **Docker Compose**.

Descripción del Problema Editorial.
Una editorial digital requiere una infraestructura moderna para administrar autores y publicaciones de forma escalable y mantenible. Actualmente, el proceso editorial presenta desafíos en la gestión de flujos de trabajo que involucran etapas de creación, revisión, aprobación y publicación. Existe además la necesidad técnica de validar estrictamente la existencia de un autor antes de registrar cualquier obra, evitando la inconsistencia de datos y el acoplamiento excesivo entre los dominios de información.

2. Arquitectura del Sistema

2.1. Diagrama General de la Solución (PlantUML).

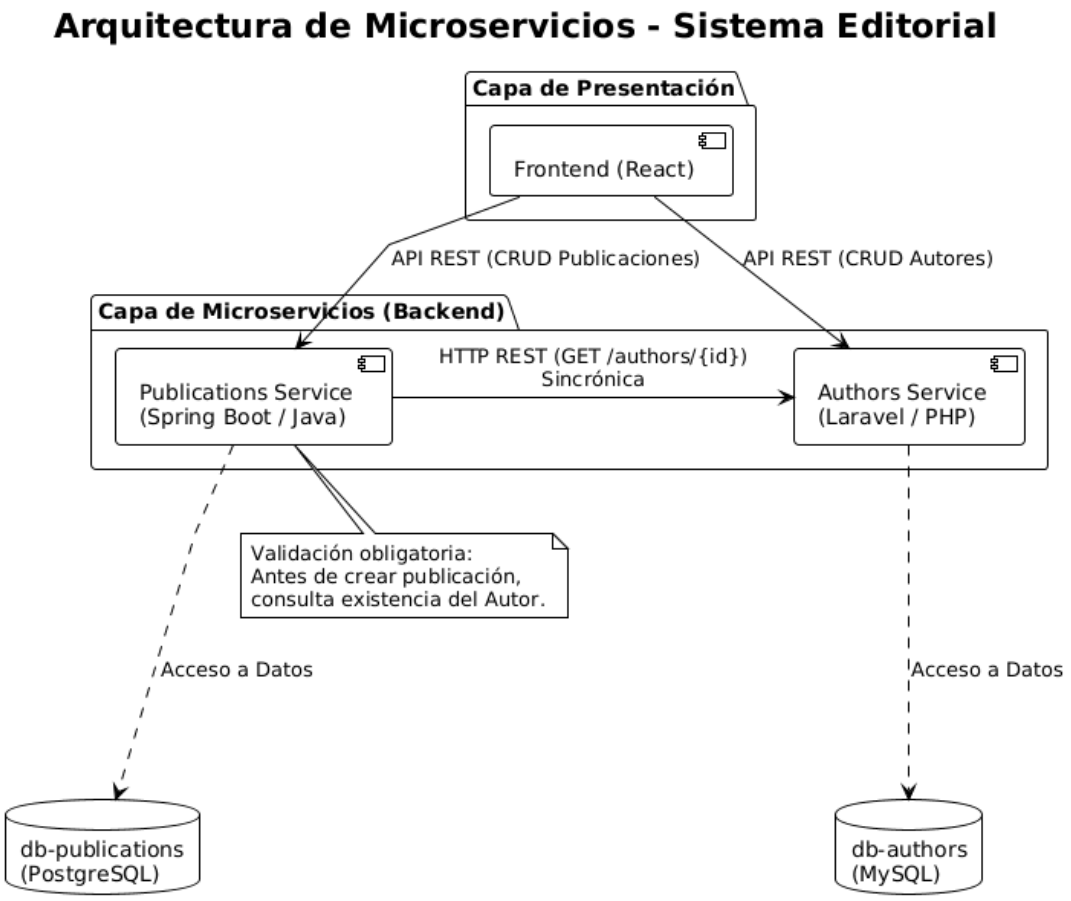

Figura 1.Diagrama de Arquitectura del Sistema

2.2. Descripción de Microservicios: Authors Service y Publications Service.

La solución se basa en una arquitectura de servicios desacoplados para garantizar escalabilidad y mantenimiento independiente.

- **Authors Service (Laravel / PHP):** Su responsabilidad principal es la administración

del ciclo de vida de los autores, incluyendo su registro, consulta y almacenamiento de metadatos como afiliación y biografía.

- **Publications Service (Spring Boot / Java):** Gestiona las publicaciones y su estado editorial. Implementa la lógica de negocio necesaria para transicionar entre estados (DRAFT, APPROVED, etc.) y depende del servicio de autores para garantizar la integridad de los datos.

2.3. Estrategia de Persistencia Independiente (MySQL y PostgreSQL).

Se ha implementado el patrón **Database per Service** para asegurar el aislamiento total de los datos y evitar el acoplamiento excesivo.

Tabla 1.Gestor de Base de Datos

Microservicio	Motor de Base de Datos	Justificación
Authors Service	**MySQL**	Ideal para la gestión estructurada y relacional de perfiles de usuario y metadatos de autores.
Publications Service	**PostgreSQL**	Provee robustez y soporte avanzado para tipos de datos complejos y estados editoriales concurrentes.

Esta estrategia permite que cada servicio escale sus recursos de almacenamiento de manera independiente según su demanda específica.

3. Modelado y Simulación de Procesos (BPMN)

3.1. Definición del Proceso Editorial.

El proceso editorial se define como un flujo de trabajo sistemático y desacoplado diseñado para gestionar el ciclo de vida de una publicación digital. Este proceso comienza con la creación de un borrador y transita por diversas etapas de validación técnica y editorial hasta alcanzar un estado final.
El flujo implementado contempla los siguientes estados obligatorios para garantizar la integridad de la información:

- **DRAFT (Borrador):** Etapa inicial de creación por parte del autor.
- **IN_REVIEW (En Revisión):** Fase de evaluación donde el contenido es analizado por el comité.
- **APPROVED / REJECTED(Aprobado):** Estados resultantes de la decisión editorial tras la revisión.
- **PUBLISHED(Publicado):** Estado final donde la publicación es liberada tras su aprobación.

Roles y Participantes (Lanes): Autor, Editor y Revisor.
Para la correcta orquestación del proceso en el modelo BPMN 2.0, se han definido tres roles principales, cada uno representado por un carril (lane) específico dentro del pool editorial:

- **Autor:** Es el responsable de iniciar el proceso mediante la creación del borrador y, en caso de ser necesario, realizar los ajustes o retrabajos solicitados por el comité de revisión.
- **Revisor (o Comité de Revisión):** Encargado de ejecutar la tarea humana de revisión técnica y editorial. Su función crítica es operar el gateway exclusivo (XOR) para decidir si una obra es aprobada, rechazada o devuelta para cambios.
- **Editor:** Responsable de la gestión final de la obra, asegurando que las publicaciones aprobadas sean preparadas y publicadas correctamente en el sistema.

3.2. Evidencia de Simulación de Escenarios (Token Simulation):

Escenario 1: Aprobación directa.

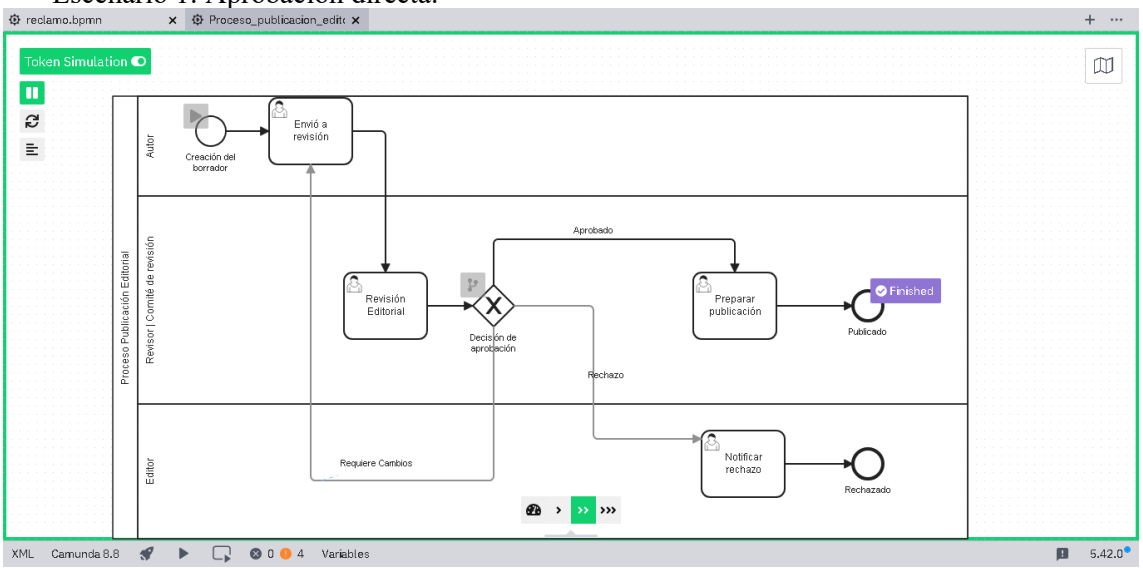

Figura 2.Simulación aprobada

Escenario 2: Rechazo.

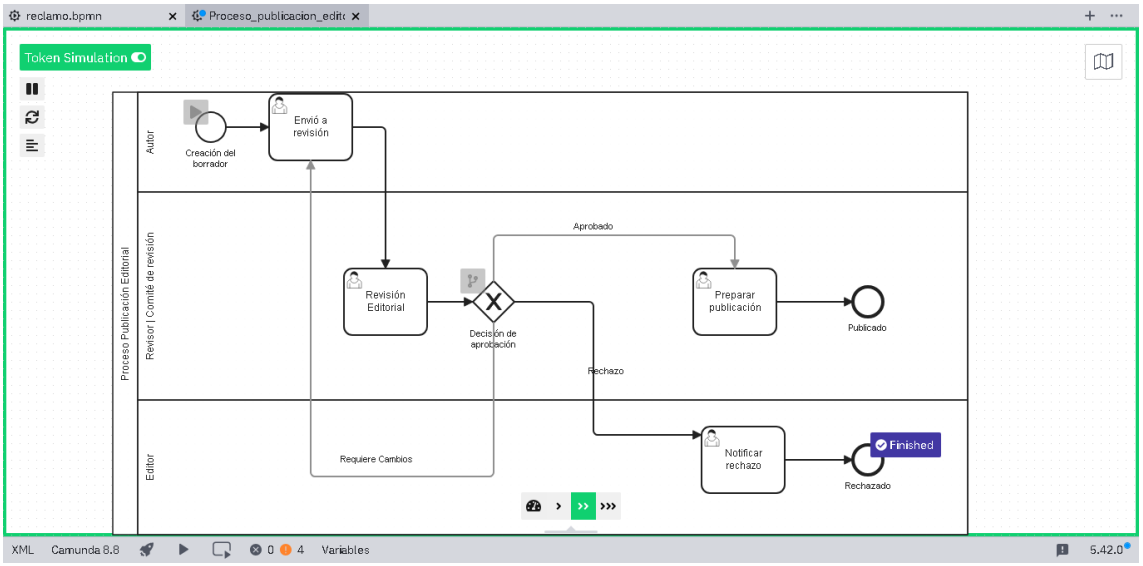

Figura 3.Simulación rechazada

Escenario 3: Reenvió por cambios y aprobación.

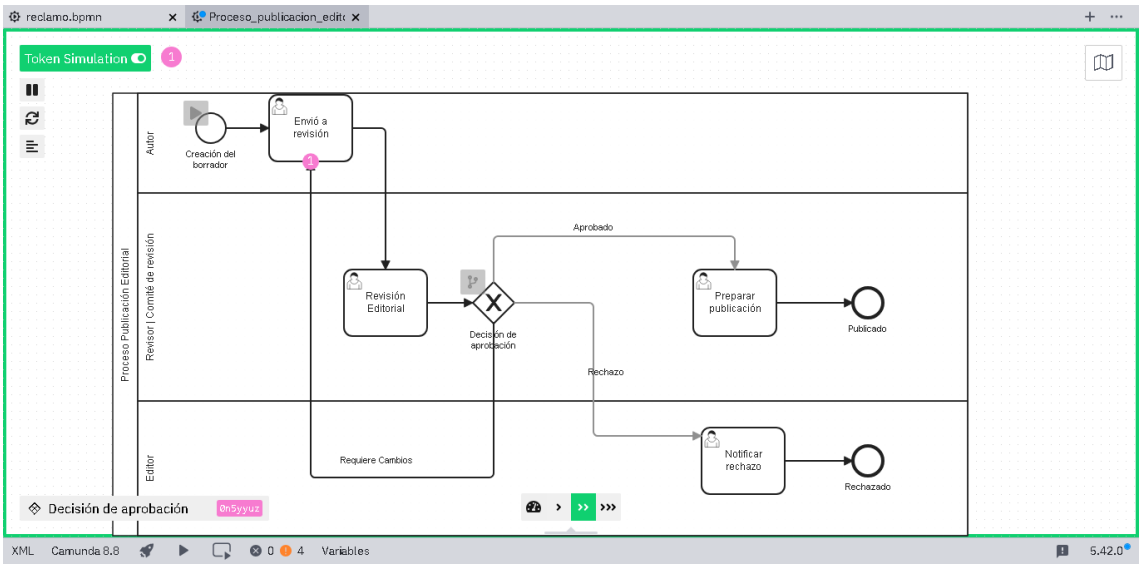

Figura 4.Simulación envió a revisión

4. Patrones de Diseño Aplicados

1. Patrón Repository (Capa de Persistencia)

El patrón Repository actúa como un mediador entre las capas de lógica de negocio y las capas de acceso a datos, abstrayendo la complejidad de las consultas.

- **Implementación:**
  - **Authors Service (Laravel):** Se utiliza para encapsular las consultas de Eloquent. Esto permite que los controladores no dependan de la implementación específica de la base de datos MySQL.
  - **Publications Service (Spring Boot):** Implementado mediante la interfaz publicationRepository. Al extender de JpaRepository, Spring genera automáticamente la implementación, permitiendo que el servicio se centre en la lógica editorial sin preocuparse por la persistencia en PostgreSQL.
- **Beneficio:** Facilita las pruebas unitarias al permitir el uso de "mocks" del repositorio y permite cambiar el motor de base de datos con un impacto mínimo en el código de negocio.

```
publicationRepository.java 1A, U
publications-service > src > main > java > as > publications_service > repository > publicationRepository.java
1  package as.publications_service.repository;
2
3  import as.publications_service.entities.digitalPublication;
4
5  import org.springframework.data.jpa.repository.JpaRepository;
6  import org.springframework.stereotype.Repository;
7
8  import java.util.List;
9  import java.util.UUID;
10
11 @Repository
12 public interface publicationRepository extends JpaRepository<digitalPublication, UUID> {
13     List<digitalPublication> findByAuthorId(String authorId);
14 }
```

Figura 5. Patrón Repository Publicacion

2. Patrón DTO - Data Transfer Object (Intercambio de Datos)

Los DTOs son objetos que transportan datos entre procesos, reduciendo el acoplamiento entre la estructura interna de la base de datos y la interfaz pública (API).

- **Implementación:**
  - **Microservicios:** Ambos servicios definen DTOs (ej. authorDTO en Java y clases en app/DTOs en Laravel) para estructurar las respuestas JSON.
  - **Uso:** Cuando el Publications Service consulta al Authors Service, los datos recibidos se mapean a un authorDTO. Esto evita que cambios en la tabla de autores de la base de datos rompan directamente el servicio de publicaciones.
- **Beneficio:** Mejora la seguridad (al no exponer campos sensibles como contraseñas o timestamps internos) y permite versionar la API de forma independiente a la base de datos.

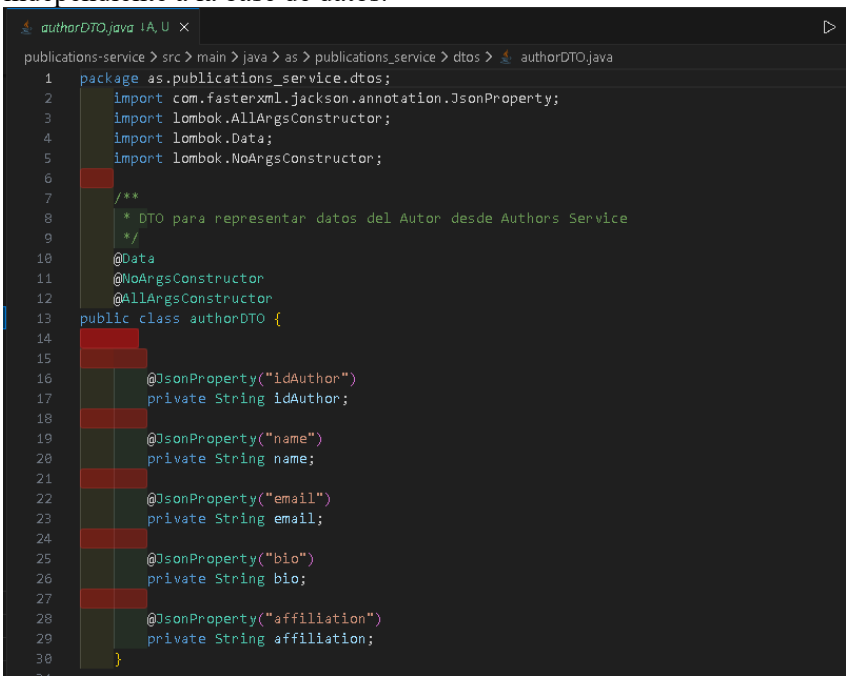

Figura 6.Patrón DTO Publications

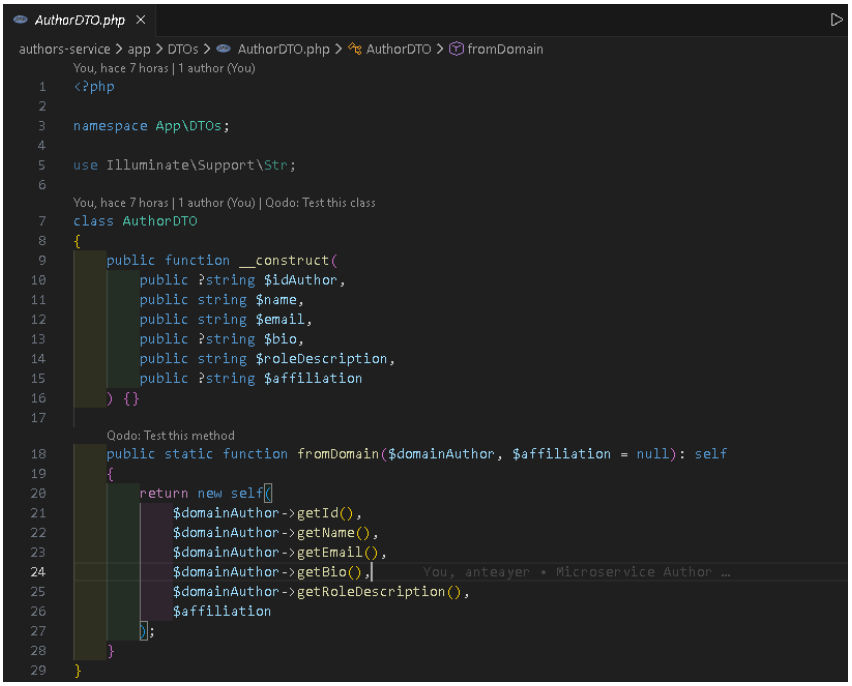

Figura 7.Patrón DTO Author

3. Patrón Adapter (Comunicación y Clientes de API)

El patrón Adapter permite que interfaces incompatibles trabajen juntas. En este proyecto,

se utiliza para integrar servicios externos y APIs de forma transparente.

- **Implementación:**
  - **Backend (authorClienteService.java):** Actúa como un adaptador que convierte las respuestas de una API REST externa (Laravel) en objetos que el dominio de Spring Boot puede procesar. Encapsula la lógica de RestTemplate y el manejo de errores de red.
  - **Frontend (api.js):** Funciona como un adaptador para la aplicación React. Centraliza las llamadas a Axios, abstrayendo a los componentes de las URLs de los microservicios y la estructura de las respuestas HTTP.
- **Beneficio:** Desacoplamiento total de la infraestructura de comunicación. Si se decide cambiar Axios por fetch o RestTemplate por WebClient, solo se modifica el adaptador sin afectar la lógica de los componentes o servicios.

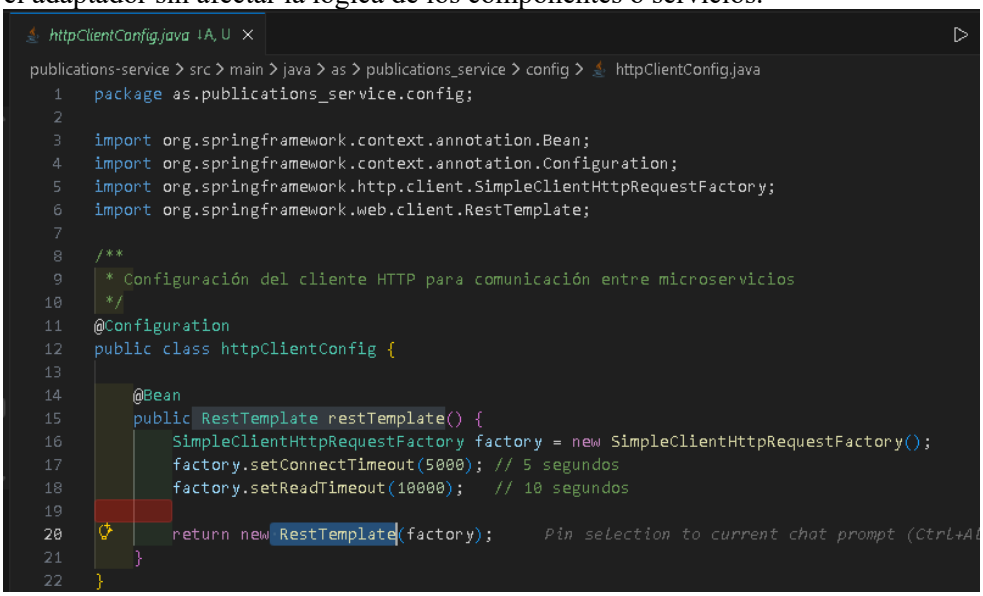

Figura 8.Patrón Adapter Publications

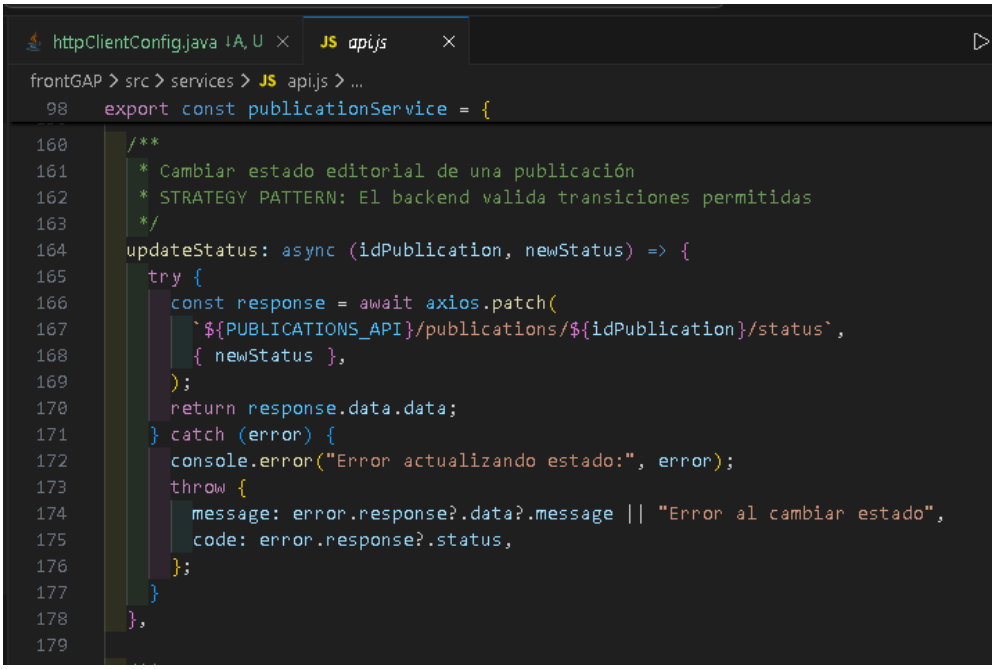

Figura 9.Patrón Adapter Frontend

4. Patrón Factory (Creación de Objetos)

Utilizado para la creación controlada de instancias, centralizando la lógica de instanciación.

- **Implementación:**
  - **Laravel Tests:** Se utilizan Factories (ej. author::factory()) para generar autores de prueba con datos consistentes. Esto asegura que los tests de integración siempre cuenten con un estado de base de datos válido.
- **Beneficio:** Reduce la duplicación de código en la configuración de pruebas y garantiza que los objetos creados cumplan siempre con las reglas de integridad del negocio.

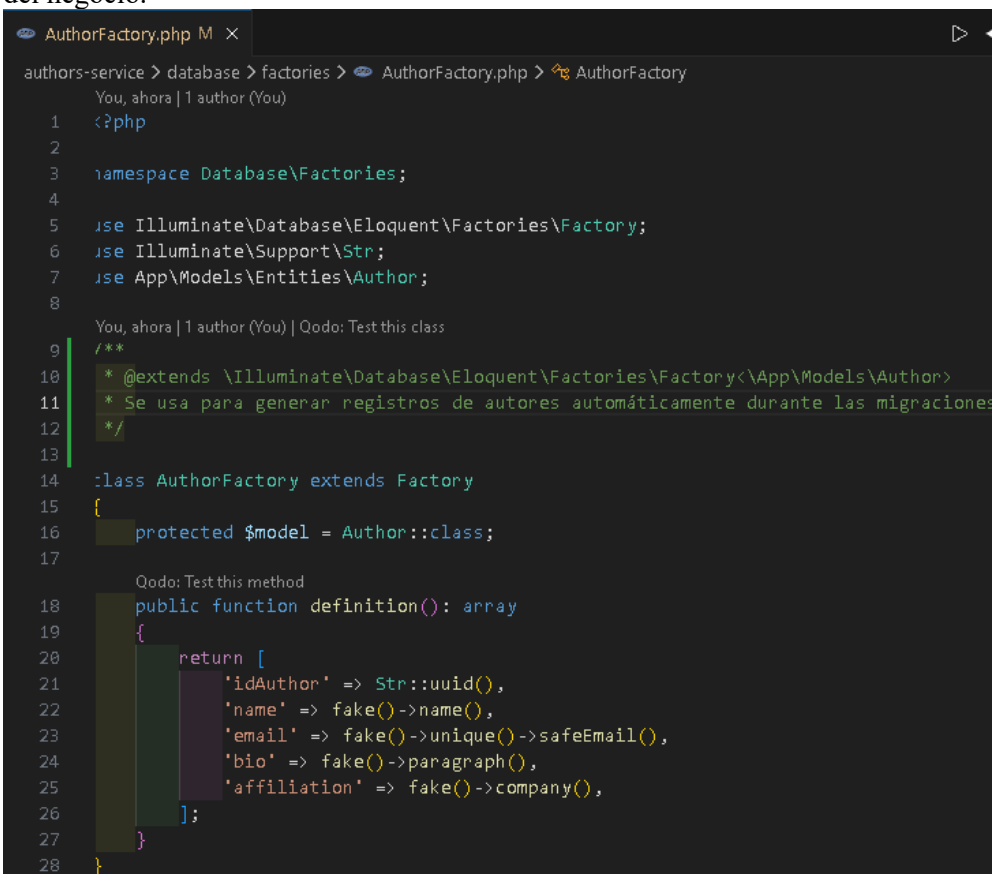

Figura 10.Patrón Factory Author

5. Evidencias de Código

Responsabilidad Única (SRP) en Controladores y Servicios.
Se evidencia en la separación clara de tareas: los controladores solo manejan la entrada/salida HTTP, mientras que la lógica reside en los servicios.

- **Ubicación Laravel:**
  - Controladores: C:\ GestionAut_Pub\authors-service\app\Http\Controllers\
  - Servicios: C: \GestionAut_Pub\authors-service\app\Services\

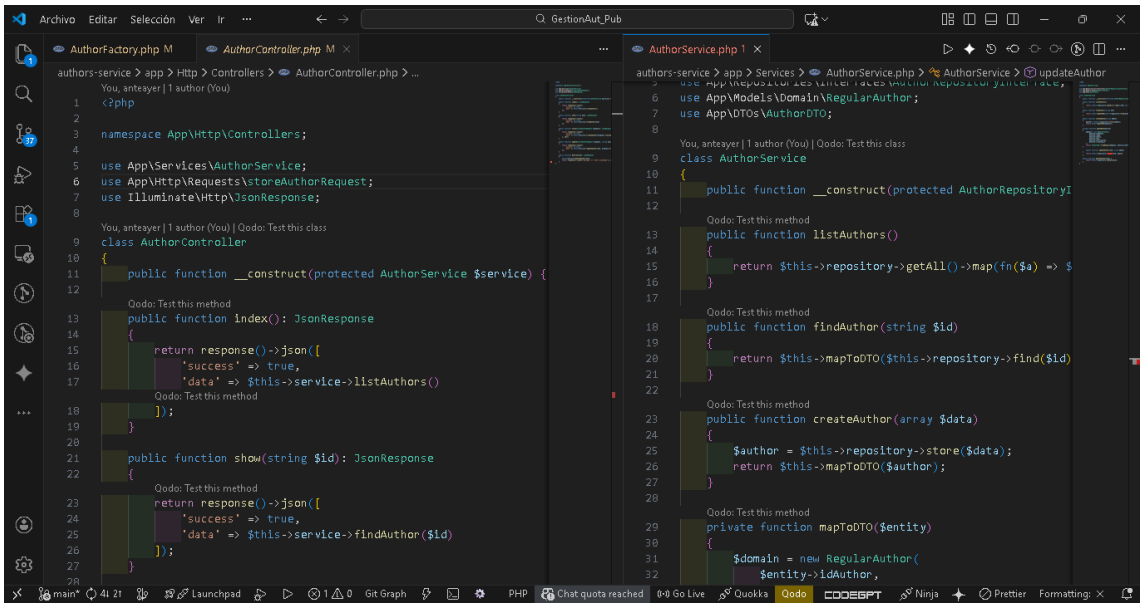

Figura 11.SRP

- **Ubicación Java:**
  - Lógica de carga de datos (separada del controlador): C:/GestionAut_Pub\publications-service\src\main\java\as\publications_service\config\DataLoader.java

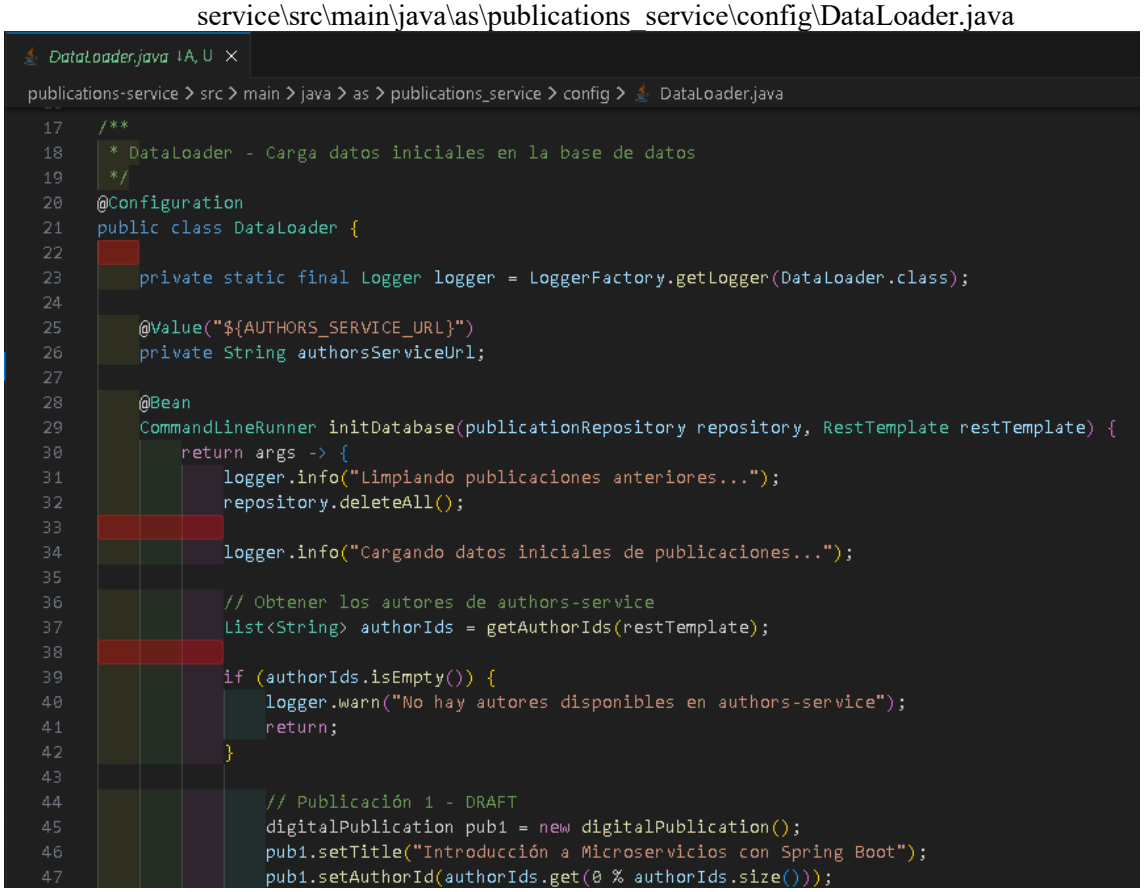

Figura 12.SRP Publication

O - Open/Closed Principle (Abierto/Cerrado)
Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación. En c:\ \GestionAut_Pub\authors-

service\app\Providers\RepositoryServiceProvider.php, el sistema está configurado para usar AuthorRepositoryInterface.

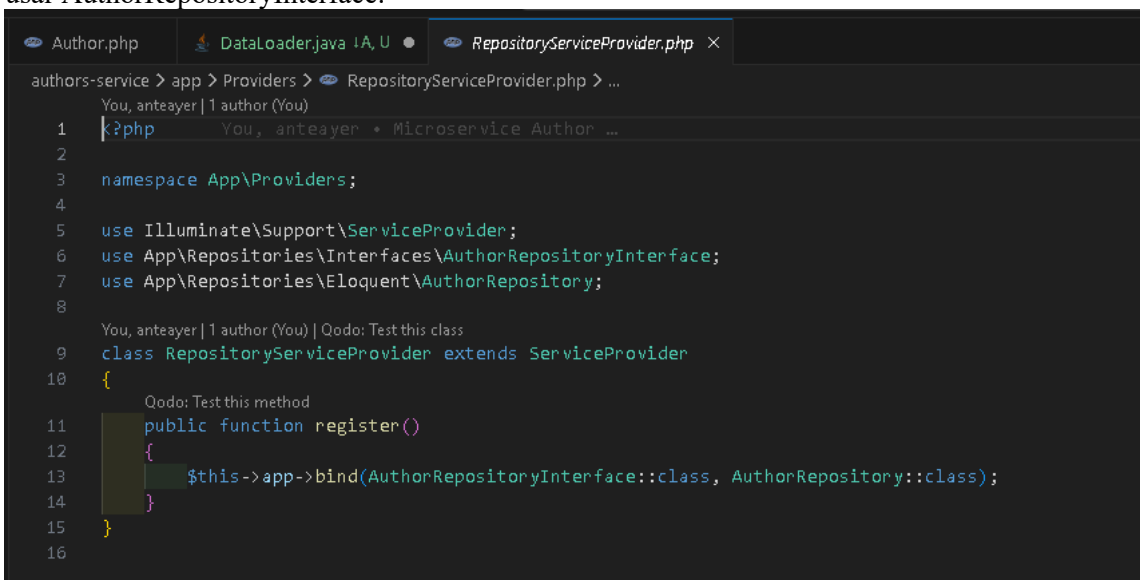

Figura 13.Open/Closed

Segregación por Capas e Inversión de Dependencias.

El sistema utiliza interfaces para desacoplar los componentes. En Spring Boot, el servicio no depende de una base de datos específica, sino de la abstracción del repositorio.

- **Evidencia en Java:** C:\GestionAut_Pub\publications-service\src\main\java\as\publications_service\repository\publicationRepository.java

Figura 14.Segregación Capas

L - Liskov Substitution Principle (Sustitución de Liskov)

Se utiliza la herencia para reutilizar comportamiento común y definir contratos.

- **Evidencia en Laravel (Factories):** C \GestionAut_Pub\authors-service\database\factories\AuthorFactory.php
  - Aquí, AuthorFactory deriva de la clase base Factory de Laravel, heredando métodos para la persistencia de datos de prueba.
- **Evidencia en Laravel (Modelos):** GestionAut_Pub\authors-service\app\Models\Entities\Author.php
  - El modelo extiende de Eloquent\Model, adquiriendo todas las capacidades del ORM de forma derivada.

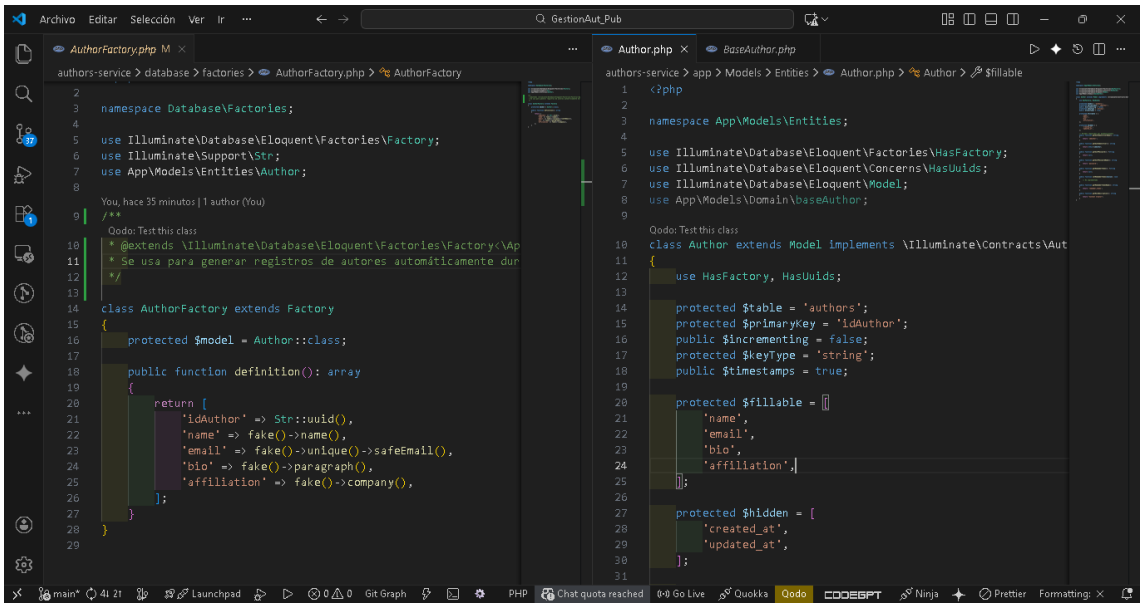

Figura 15.Liskov Substitution

6. Documentación de la API

6.1. Endpoints de Authors Service (Request/Response).

Base URL: http://localhost:8000/api
El servicio de autores gestiona la entidad principal de la cual dependen las publicaciones. Utiliza un formato de respuesta estandarizado: { "success": boolean, "data": object|array }.
Tabla 2. Endpoint Author

Método	Endpoint	Descripción	Payload (Request Body)
GET	/authors	Lista todos los autores.	N/A
POST	/authors	Crea un nuevo autor.	{"name": "string", "email": "string", "bio": "string", "affiliation": "string"}
GET	/authors/{id}	Obtiene un autor por UUID.	N/A
PUT	/authors/{id}	Actualiza datos del autor.	{"name": "string", "email": "string", ...}
DELETE	/authors/{id}	Elimina un autor.	N/A

6.2. Endpoints de Publications Service y Manejo de Estados.

Base URL: http://localhost:8080/api
Este servicio gestiona el ciclo de vida editorial de las publicaciones. Es fundamental el manejo de authorId, el cual debe existir previamente en el Authors Service.
Tabla 3.Endpoint Publication

Método	Endpoint	Descripción	Payload (Request Body)
POST	/publications	Crea una publicación.	{"title": "string", "content": "string", "authorId": "UUID", "status": "DRAFT"}
GET	/publications	Lista publicaciones.	Opcional: ?idAuthor=UUID
GET	/publications/{id}	Obtiene una publicación por UUID	N/A
PATCH	/publications/{id}/status	Cambia el estado editorial.	{"status": "EDITORIAL_STATUS"}

DELETE	/publications/{id}	Elimina publicacion	N/A

Manejo de Estados (Editorial Status)

El servicio implementa una máquina de estados simple a través del enum editorialStatus. Los estados permitidos son:

1. **DRAFT (Borrador):** Estado inicial de la publicación.
2. **IN_REVIEW (En Revisión):** La publicación está siendo evaluada por pares.
3. **APPROVED (Aprobado):** Lista para ser publicada pero aún no visible al público.
4. **PUBLISHED (Publicado):** Estado final, visible en el portal.

7. Guía de Despliegue y Ejecución

7.1. Requisitos Previos (Docker y Docker Compose).

Antes de iniciar, asegúrese de tener instaladas las siguientes herramientas en su estación de trabajo:

- **Docker:** Versión 20.10.0 o superior.
- **Docker Compose:** Versión 2.0.0 o superior.
- **Git:** Para la gestión del repositorio.
- **Recursos de Hardware:** Se recomienda un mínimo de 4GB de RAM disponibles para la ejecución simultánea de los 5 contenedores.

7.2. Configuración de Variables de Entorno.

El proyecto utiliza un archivo .env en la raíz para centralizar la configuración de red y credenciales.

1. Localice el archivo .env.example en la raíz de la carpeta GestionAut_Pub.
2. Cree una copia y renómbrela a .env:

```
cp .env.example .env
```

3. Variables clave a verificar:
   - AUTHORS_SERVICE_URL: Debe apuntar a http://authors-service:8000/api para la comunicación interna entre contenedores.
   - MYSQL_DATABASE / POSTGRES_DB: Nombres de las bases de datos para cada microservicio.
   - FRONTEND_PORT: Puerto donde se expondrá la aplicación React (por defecto 5173).

7.3. Comandos de Despliegue: docker compose up --build.

Para levantar todo el stack tecnológico, abra una terminal en la raíz del proyecto y ejecute:

```
docker compose up -d --build
```

Acciones Adicionales de Inicialización

Una vez que los contenedores estén arriba, es necesario preparar la base de datos de autores (Laravel):

- **Ejecutar Migraciones y Seeders:**

```
docker compose exec authors-service php artisan migrate:fresh --seed
```

Esto creará las tablas en MySQL y poblará los autores iniciales necesarios para que el

servicio de publicaciones pueda validar datos.

7.4. Acceso a Servicios (URLs de Frontend y APIs).

Una vez completado el despliegue, puede acceder a los distintos componentes a través de las siguientes URLs:

Tabla 4.Acceso a servicios

Componente	URL de Acceso	Puerto Host
Frontend (Vite + React)	http://localhost:5173	5173
Authors Service (API)	http://localhost:8000/api/authors	8000
Publications Service (API)	http://localhost:8080/api/publications	8080
MySQL (Autores)	localhost	3307
PostgreSQL (Publicaciones)	localhost	5433

Notas de Mantenimiento

- **Logs:** Para monitorear el comportamiento de un servicio específico (por ejemplo, errores de conexión en Spring Boot), use: docker compose logs -f publications-service.
- **Limpieza:** Para detener el sistema y eliminar los contenedores (manteniendo los datos), use docker compose down. Si desea borrar también los datos de las bases de datos, use docker compose down -v.

Si es exitoso el despliegue mostrara lo siguiente:

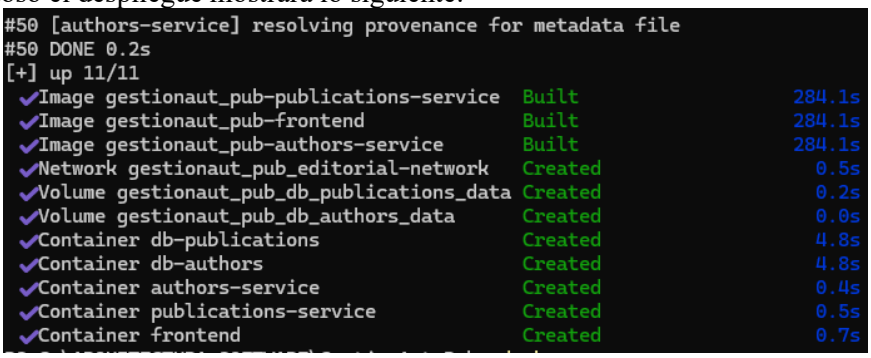

Figura 16.Resultado esperado

8. Conclusiones y Recomendaciones

- Eficacia de la Arquitectura de Microservicios: La separación de responsabilidades entre el Authors Service (PHP/Laravel) y el Publications Service (Java/Spring Boot) ha demostrado ser exitosa. Esta estructura permite que cada servicio escale de forma independiente y utilice el stack tecnológico que mejor se adapte a sus necesidades (CRUD rápido en Laravel y lógica de negocio robusta en Spring Boot).
- Interoperabilidad y Contratos de Datos: El uso de DTOs y el cumplimiento de estándares REST han facilitado la comunicación entre servicios. La capacidad del servicio de publicaciones para validar la existencia de autores en un sistema externo garantiza la integridad referencial lógica, incluso sin claves foráneas físicas entre bases de datos distintas.
- Despliegue Consistente: Gracias a Docker y Docker Compose, se ha eliminado el problema de "funciona en mi máquina". El ecosistema completo, incluyendo bases de datos heterogéneas (MySQL y PostgreSQL), se levanta con un solo comando, asegurando que todos los desarrolladores y entornos de prueba operen sobre la misma infraestructura.

9. Pruebas en Postman

La colección de Postman se encuentra en el repositorio: https://github.com/saoricoder/GestionAutor_Publicacion/blob/ef54944ec976978e198e24c4e7a414c2298c0fb7/Author-Publications.postman_collection.json

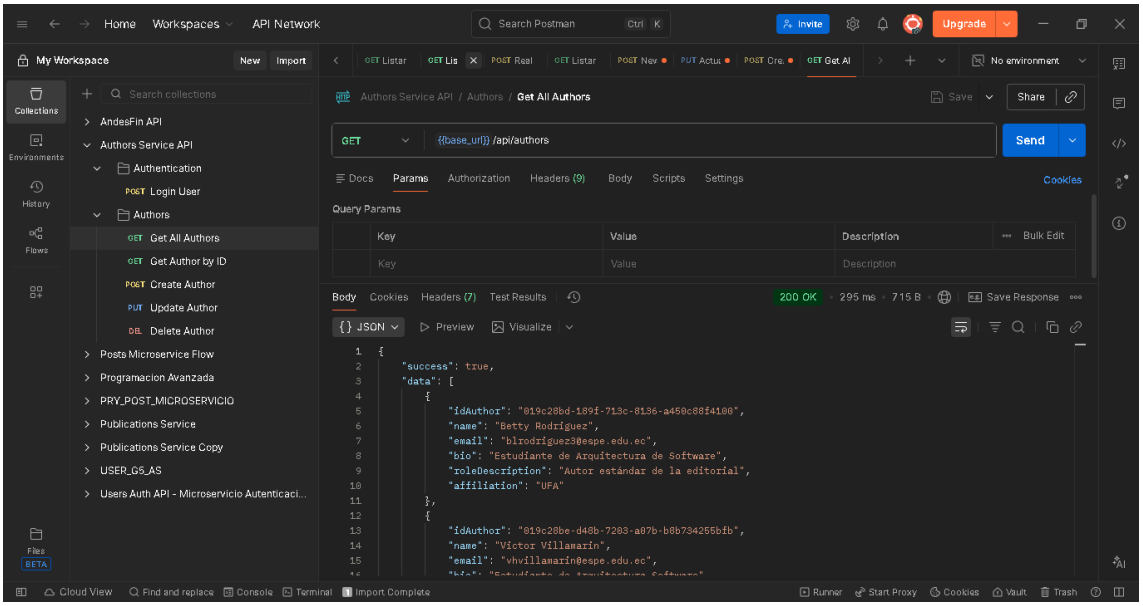

Figura 17.Prueba de api Authors con postman

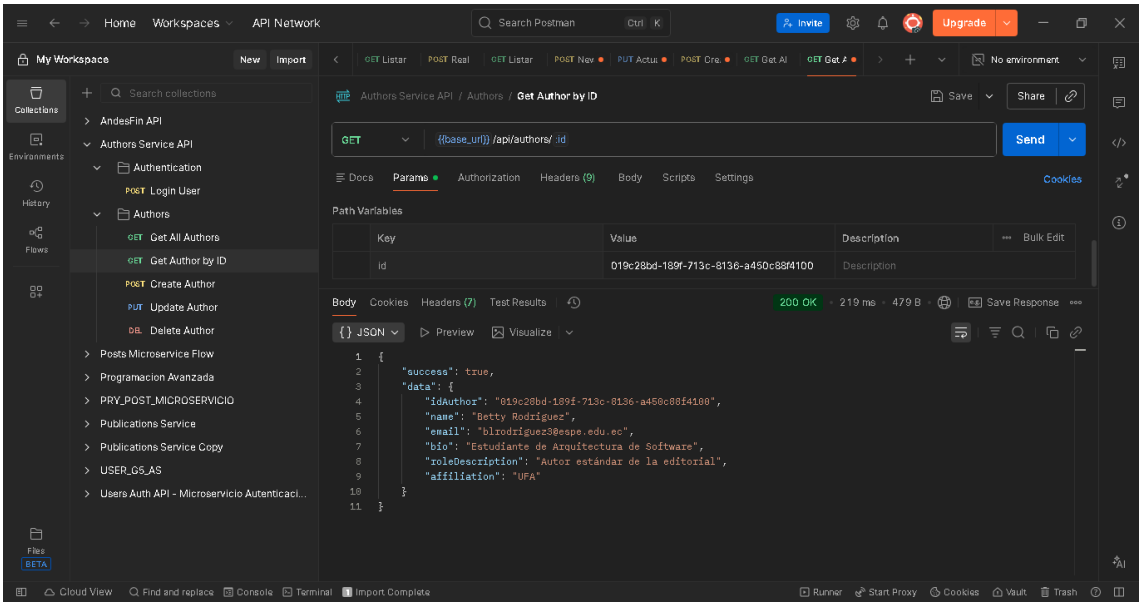

Figura 18.Prueba de crud para obtener datos con id

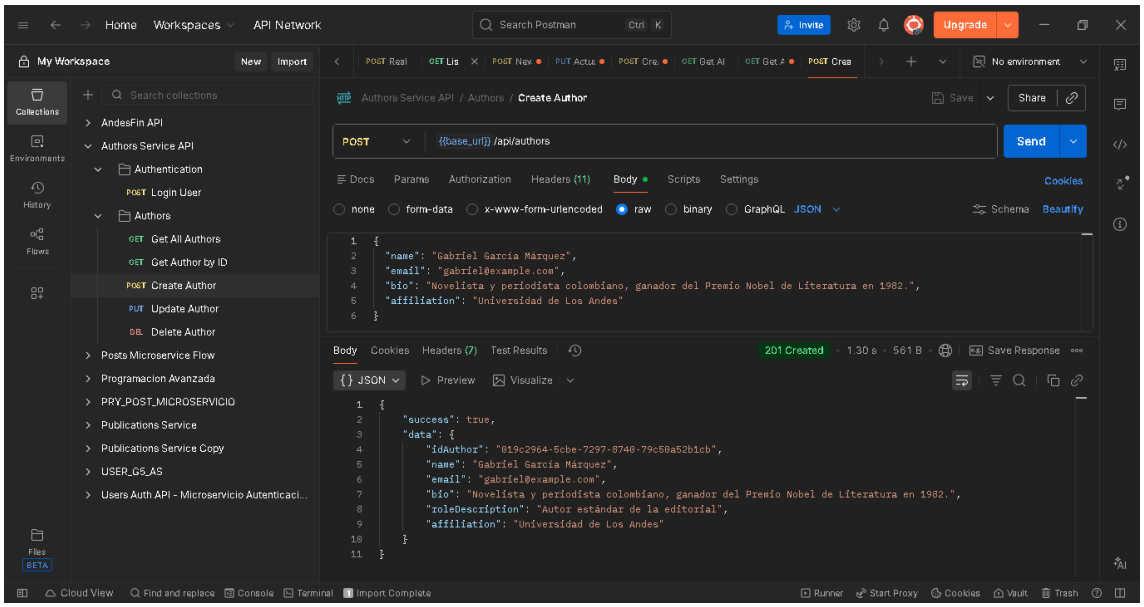

Figura 19.Prueba de método post en api - Authors

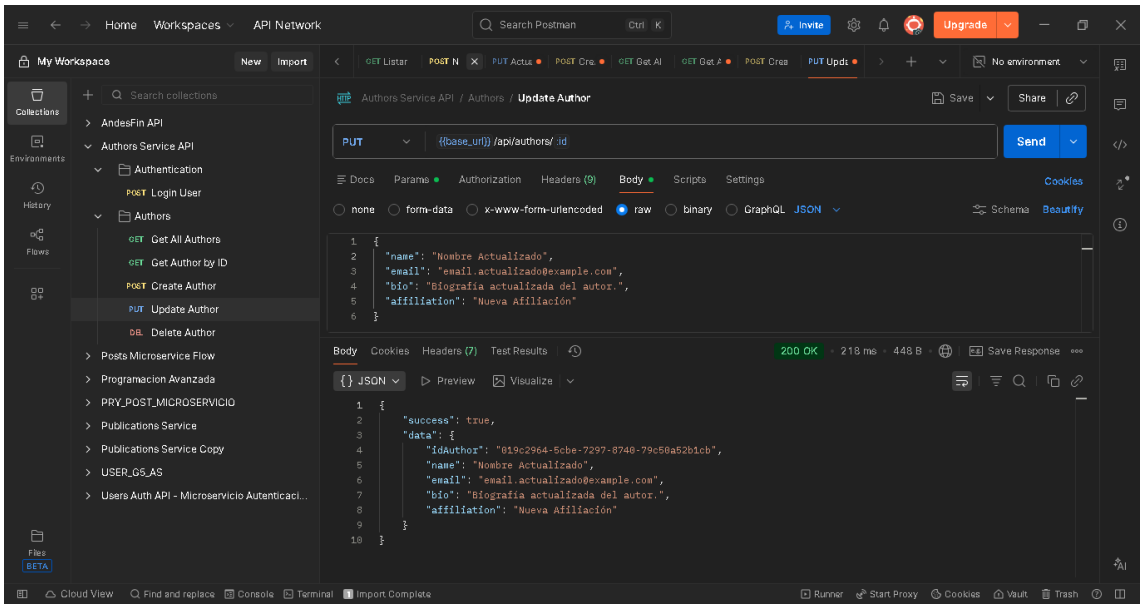

Figura 20.Prueba de método put en api - Authors

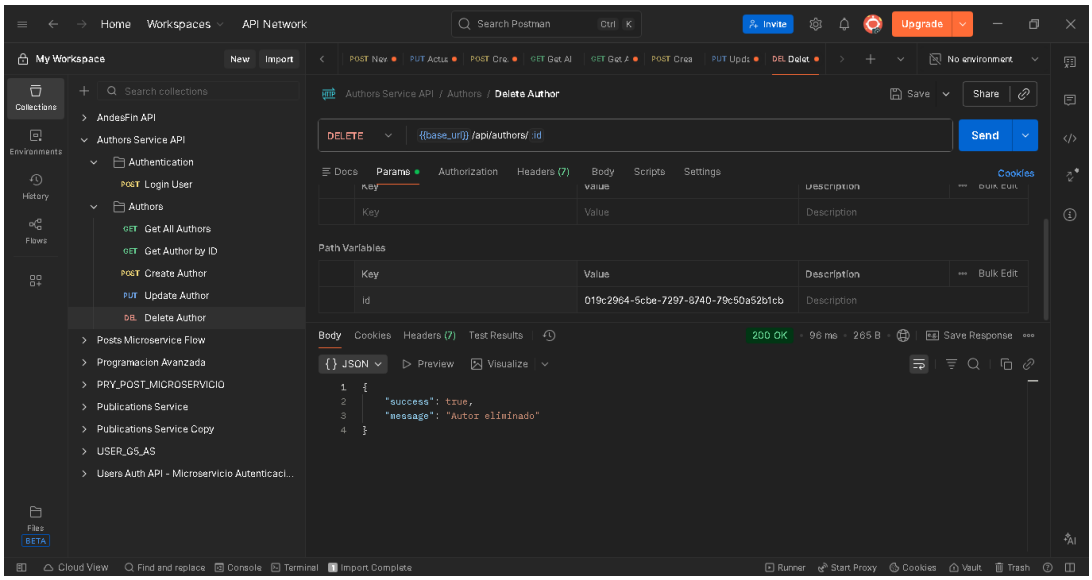

Figura 21.Prueba de método put en api - Authors

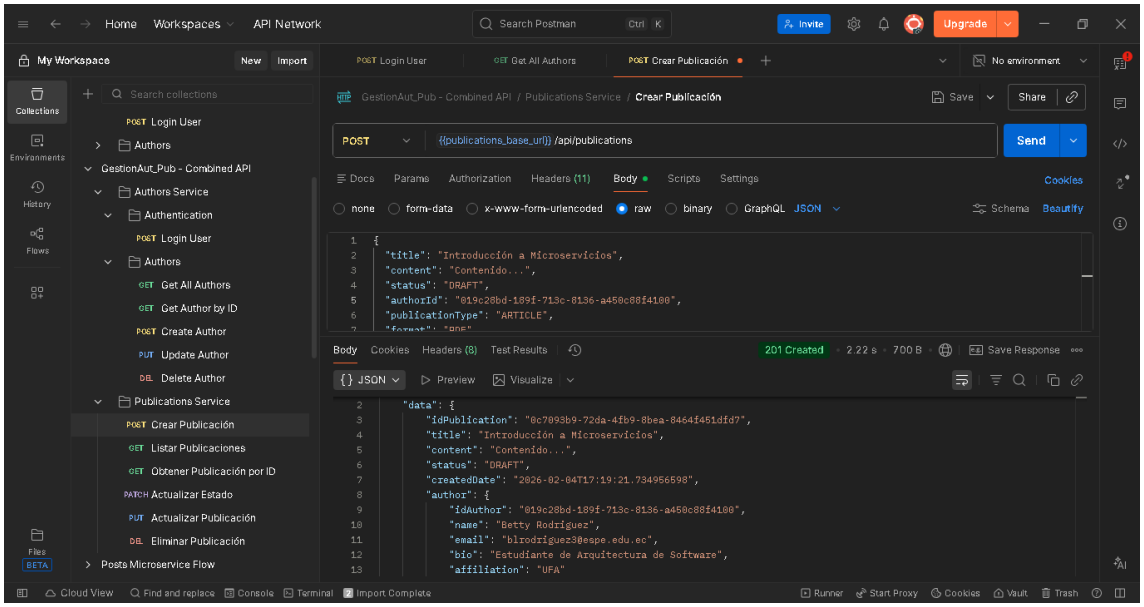

Figura 22.Prueba de método post en api publications

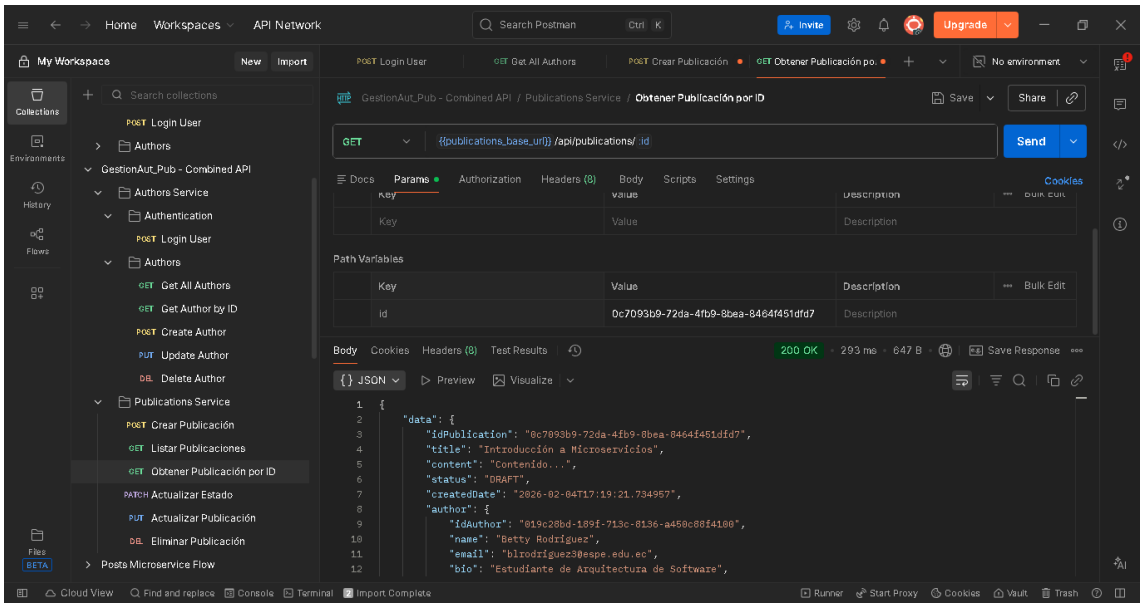

Figura 23.Prueba de método get en api publications

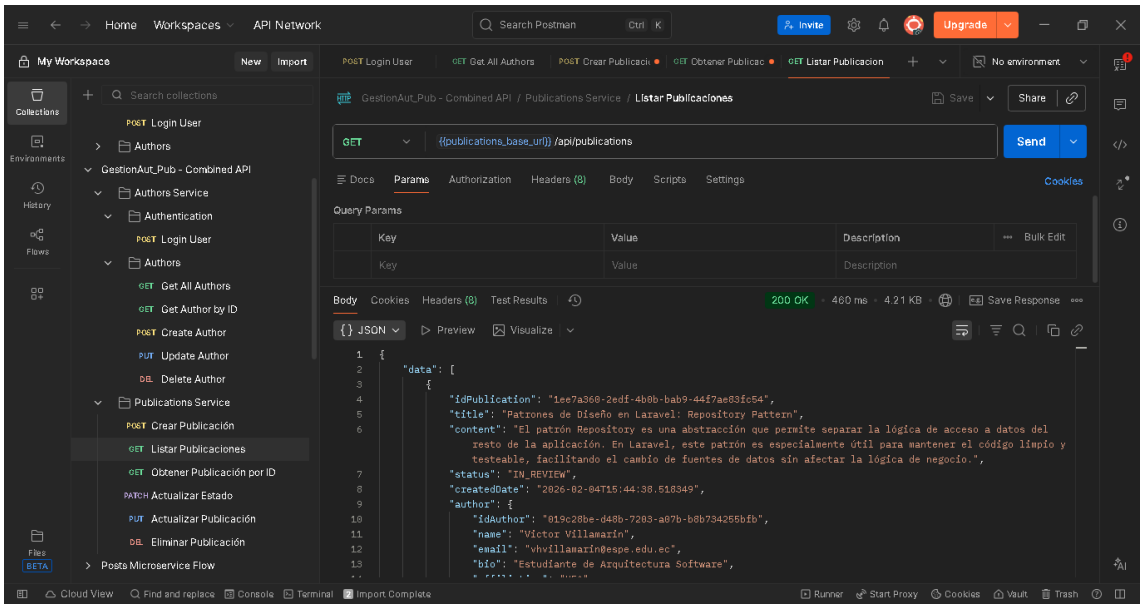

Figura 24.Prueba de método get para listar publicaciones

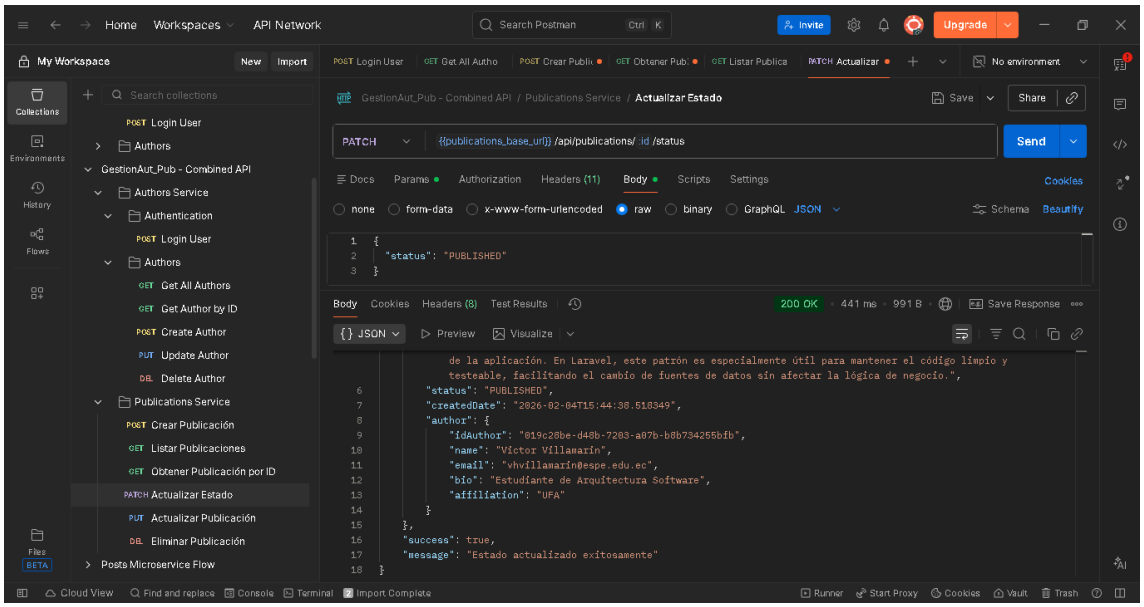

Figura 25.Prueba de método patch en api publications

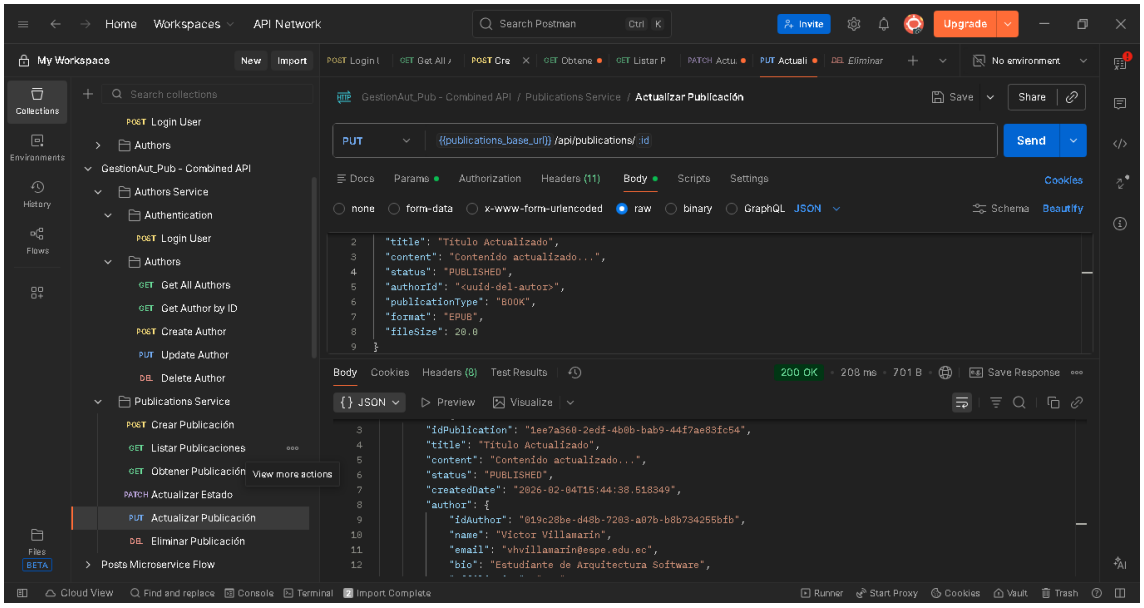

Figura 26.Prueba de método actualizar en api publications

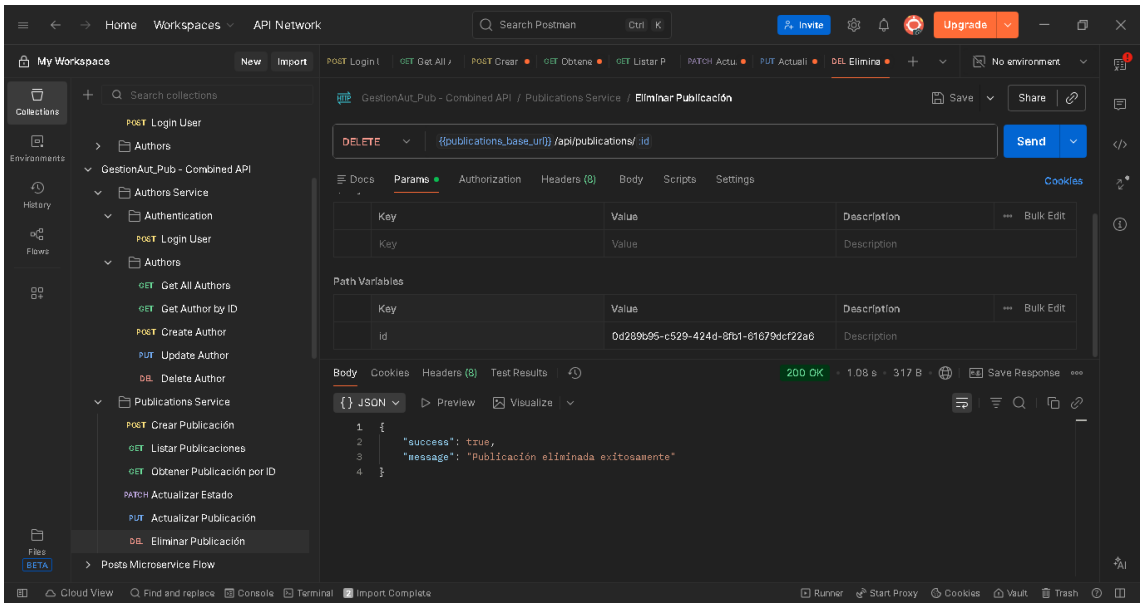

Figura 27.Prueba de método eliminar en api publications