



UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE  
UNIDAD DE EDUCACIÓN A DISTANCIA  
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

## ARQUITECTURA DE SOFTWARE

### ACTIVIDAD DE APRENDIZAJE 3

ARQUITECTURA DE MICROSERVICIOS PARA LA GESTION DE  
AUTORES Y PUBLICACIONES

### REPOSITORIO DEL PROYECTO

<https://github.com/adryborja95/microservices-authors-publications>

**AUTOR:**  
BORJA DIAZ ADRIANA MARIBEL

**DOCENTE:**  
ING. ANGEL CUDCO

**FECHA DE ENTREGA:**  
4 DE FEBRERO DEL 2026

## 1. Introducción

El presente documento describe el diseño, implementación y despliegue de una solución basada en arquitectura de microservicios para la gestión de autores y publicaciones en una editorial digital. La solución se compone de dos microservicios independientes Authors Service y Publications Service desarrollados siguiendo principios de arquitectura de software, buenas prácticas de diseño y criterios de desacoplamiento, escalabilidad y mantenibilidad.

El sistema incorpora un frontend web desarrollado en React para la gestión de autores, publicaciones y estados editoriales, así como un modelo de proceso editorial representado mediante BPMN en Camunda Modeler. Dicho modelo permite simular y validar el flujo de negocio de manera independiente a la implementación técnica, utilizando Token Simulation para evaluar distintos escenarios del proceso editorial.

La arquitectura propuesta aplica principios SOLID, patrones de diseño y orquestación mediante Docker Compose, garantizando un despliegue reproducible y consistente. Este enfoque permite validar tanto la correcta separación de responsabilidades entre componentes como la viabilidad técnica de la solución en un entorno distribuido. Arquitectura General del Sistema

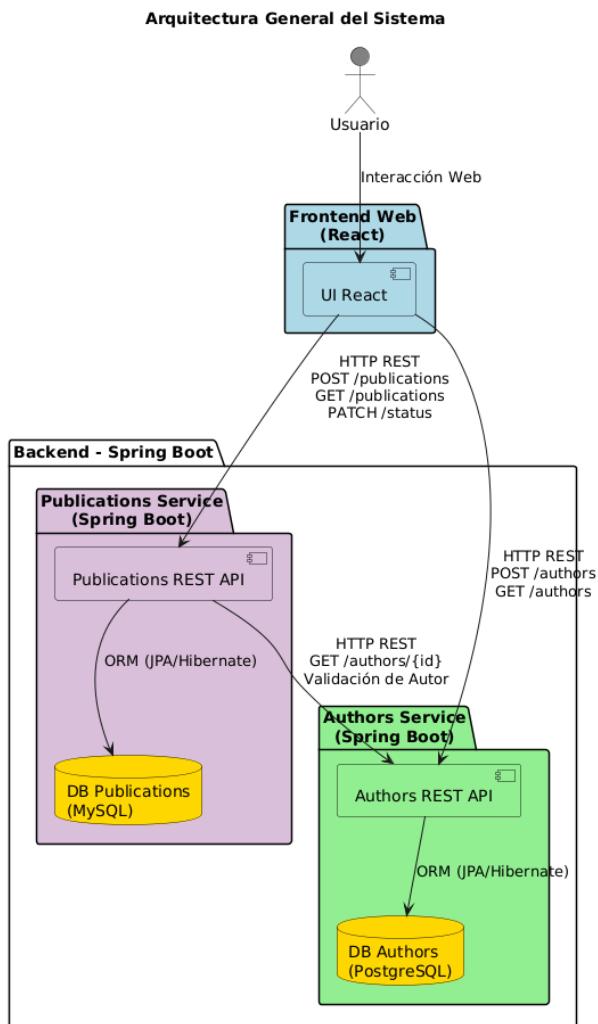


Fig.1. Arquitectura General del Proyecto

La arquitectura general del sistema se diseñó siguiendo el enfoque de microservicios desacoplados, utilizando Spring Boot como framework backend y React como tecnología frontend. Esta arquitectura permite una separación clara de responsabilidades, facilita la escalabilidad del sistema y cumple con los principios de diseño establecidos para aplicaciones distribuidas.

El Frontend Web, desarrollado en React, constituye la capa de presentación del sistema y es el punto de interacción directa con el usuario. A través de esta interfaz, el usuario puede gestionar autores y publicaciones, así como consultar y actualizar el estado editorial de una publicación. El frontend se comunica con los microservicios backend mediante HTTP REST, sin acceder directamente a las bases de datos.

En el backend se implementaron dos microservicios independientes desarrollados con Spring Boot. El microservicio Authors Service es responsable de la gestión del ciclo de vida de los autores y expone una API REST para la creación y consulta de autores. Este microservicio utiliza una base de datos PostgreSQL exclusiva, accedida mediante ORM (JPA/Hibernate), garantizando el aislamiento de datos y evitando dependencias externas.

Por su parte, el microservicio Publications Service administra las publicaciones y su estado editorial. Este servicio utiliza una base de datos MySQL independiente, también gestionada mediante JPA/Hibernate, lo que asegura una persistencia desacoplada del resto del sistema. Al crear una publicación, este microservicio realiza una validación síncrona mediante HTTP REST hacia el microservicio de Autores, consultando el endpoint GET /authors/{id} para verificar la existencia del autor, cumpliendo con la dependencia obligatoria entre microservicios sin generar acoplamiento a nivel de datos.

La comunicación entre microservicios se realiza exclusivamente mediante HTTP REST síncrono, evitando accesos directos a las bases de datos de otros servicios y previniendo dependencias circulares. Este diseño cumple con las buenas prácticas de arquitectura de microservicios, promoviendo la mantenibilidad, la independencia de despliegue y la coherencia estructural del sistema.

## 2. Microservicio Authors Service

### 2.1. Descripción del Microservicio

El microservicio Authors Service es responsable de administrar el ciclo de vida de los autores dentro de la plataforma editorial. Su función principal es permitir el registro y consulta de autores, manteniendo la información desacoplada del microservicio de publicaciones, con el fin de evitar dependencias circulares y facilitar la escalabilidad del sistema.

Este microservicio no consulta ni accede a información de publicaciones, cumpliendo el principio de independencia entre microservicios.

### 2.2. Modelo de Dominio

El modelo de dominio del microservicio Authors Service fue diseñado aplicando principios de orientación a objetos y herencia, con el fin de representar de manera clara y extensible la información asociada a los autores.

Se define una clase abstracta base Persona, la cual agrupa los atributos comunes a cualquier persona dentro del dominio, tales como identificador único (UUID), tipo de identificación, número de identificación, nacionalidad, nombres, apellidos, correo electrónico y teléfono.

A partir de esta clase base se deriva la clase Author, que representa específicamente a un autor dentro del sistema e incorpora atributos propios del contexto editorial, como la biografía y el género literario. Este enfoque permite reutilizar atributos comunes y especializar el modelo sin duplicar código.

Adicionalmente, se emplea la enumeración TipoIdentificacion, la cual restringe los valores permitidos para el tipo de identificación (Cédula, Pasaporte y RUC), garantizando consistencia y validación a nivel de dominio.

El uso de identificadores tipo UUID asegura la unicidad de los registros y favorece el funcionamiento del sistema en entornos distribuidos. Este diseño cumple con el requisito de incluir una clase abstracta base y al menos una clase derivada, y facilita la extensibilidad del sistema ante futuros requerimientos.

The screenshot shows a Java code editor with four tabs open:

- Author.java**: An abstract class extending Persona. It includes fields for id (UUID), tipoidentificacion (enum), nombre, apellido, email, telefono, and nacionalidad. It uses Lombok annotations like @Id, @GeneratedValue, and @Column.
- Persona.java**: A base class with fields for id (UUID), nombre, apellido, email, telefono, and nacionalidad. It uses Lombok annotations like @Id, @GeneratedValue, and @Column.
- Author.java**: A concrete class extending Persona. It adds a biografia field and uses Lombok annotations like @Column and @Entity.
- Tipoidentificacion.java**: An enum with three values: CEDULA, PASAPORTE, and RUC.

Fig.2. Modelo de dominio del microservicio Authors Service

### 2.3. Estructura – Separación por Capas

El microservicio Authors Service fue estructurado siguiendo una arquitectura por capas, alineada con los principios SOLID y las buenas prácticas de diseño en aplicaciones basadas en microservicios.

Esta organización permite una clara separación de responsabilidades, facilitando el mantenimiento, la escalabilidad y la comprensión del sistema.

La estructura principal del proyecto es la siguiente:

- **Controller:** Contiene los controladores REST encargados de exponer los endpoints HTTP del microservicio. Esta capa recibe las solicitudes del cliente, valida los datos de entrada mediante DTOs y delega la lógica de negocio a la capa de servicios.
- **Dto:** Incluye los Data Transfer Objects utilizados para la entrada y salida de datos (AuthorRequestDTO, AuthorResponseDTO). Su uso evita exponer directamente las entidades del dominio y permite un mayor control sobre la información intercambiada.
- **Entity:** Define el modelo de dominio del microservicio, incluyendo la clase abstracta base (Persona) y la clase derivada (Author), así como la enumeración TipoIdentificacion. Estas clases representan la estructura persistente de la base de datos.
- **Repository:** Implementa el acceso a datos mediante Spring Data JPA. El repositorio abstrae la lógica de persistencia, aplicando el Repository Pattern.
- **Service:** Contiene la lógica de negocio del microservicio. La interfaz AuthorService y su implementación AuthorServiceImpl actúan como una fachada entre los controladores y la capa de persistencia.
- **Exception:** Centraliza el manejo de errores del sistema mediante excepciones personalizadas (NotFoundException, ConflictException) y un GlobalExceptionHandler, garantizando respuestas HTTP consistentes y controladas.

Esta organización permite cumplir con el principio de Single Responsibility, ya que cada capa tiene una función claramente definida dentro del microservicio.

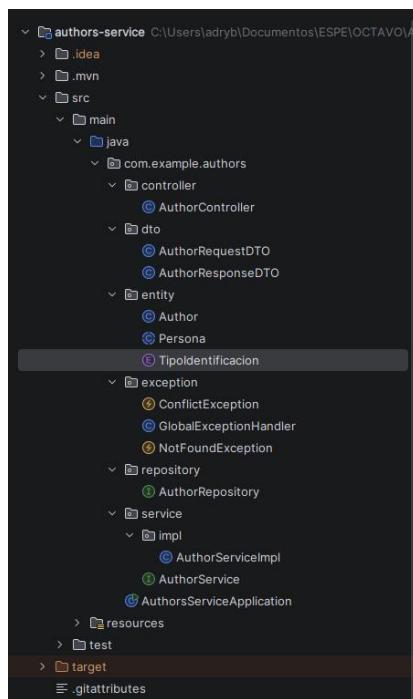


Fig.3. Estructura Microservicio Authors Service

## 2.4. Persistencia y ORM

La persistencia de datos se realiza mediante JPA/Hibernate, utilizando una base de datos PostgreSQL exclusiva para este microservicio (db-authors). Hibernate se encarga de la creación automática de las tablas a partir de las entidades definidas, garantizando consistencia entre el modelo de dominio y la base de datos.

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer displays the database schema with the 'authors' table selected under the 'Tables (1)' section. The main pane shows a SQL query: 'SELECT \* FROM public.authors'. The results are displayed in a Data Output grid:

	id [PK] uuid	apellido character varying(30)	email character varying(50)	identificacion character varying(20)	nacionalidad character varying(30)	nombre character varying(30)	telefono character varying(15)
1	db38cf57-e8c-4e25-be99-43148aa546..	Borja	adry.borja@hotmail.com	1721817029	ecuatoriana	Adriana	0986333313
2	5dd30986-e213-4ee-9fc-f2a7de427c	Borja	jessica.borja@hotmail.c...	1721817011	ecuatoriana	Jessica	0986838465
3	ffd5994-8acc-41e5-90df-5ffcc3f6d459	Diaz	rosa.diaz@hotmail.com	1712093630	argentina	Rosa	0999999999
4	8a44b7bb-7752-41f5-9b66-85d5624238..	Diaz	carlos.diaz@hotmail.com	ABC123456	colombiano	Carlos	+573101234567

Fig.4. Base de datos PostgreSQL y tabla authors

## 2.5. API REST – Endpoints Principales

El microservicio expone los siguientes endpoints REST:

- POST /authors: <http://localhost:8081/authors>

Permite registrar un nuevo autor.

- Ejemplo de body:

```
{  
  "tipoIdentificacion": "PASAPORTE",  
  "identificacion": "ABC123456",  
  "nacionalidad": "colombiano",  
  "nombre": "Carlos",  
  "apellido": "Diaz",  
  "email": "carlos.diaz@hotmail.com",  
  "telefono": "+573101234567",  
  "biografia": "Autor colombiano, nacido en el estado de Bogota. Amante de la literatura griega.",  
  "generoLiterario": "Literatura Griega"  
}
```

- Respuesta JSON:

```

POST http://localhost:8081/authors
{
  "tipoIdentificacion": "PASAPORTE",
  "identificacion": "ABC123456789",
  "nacionalidad": "colombiano",
  "nombre": "Martin",
  "apellido": "Lopez",
  "email": "martin.lopez@hotmail.com",
  "telefono": "+573101234599",
  "biografia": "Autor colombiano, nacido en el estado de Medellin. Amante de la literatura griega.",
  "generoliterario": "literatura Griega"
}

```

```

201 Created
{
  "id": "54732894-8a10-4641-9602-cea2dd9476d",
  "tipoIdentificacion": "PASAPORTE",
  "identificacion": "ABC123456789",
  "nacionalidad": "colombiano",
  "nombre": "Martin",
  "apellido": "Lopez",
  "email": "martin.lopez@hotmail.com",
  "telefono": "+573101234599",
  "biografia": "Autor colombiano, nacido en el estado de Medellin. Amante de la literatura griega.",
  "generoliterario": "literatura Griega"
}

```

Fig.5. Creación de autor desde POSTMAN

- GET /authors/{id}: <http://localhost:8081/authors/{id}>  
Obtiene la información de un autor por su identificador.

Key	Value	Description
Key	Value	Description

```

200 OK
{
  "id": "db38cbf5-7e8c-4e26-ba99-e3148aa6462b",
  "tipoIdentificacion": "CEDULA",
  "identificacion": "1721817829",
  "nacionalidad": "ecuatoriana",
  "nombre": "Adriana",
  "apellido": "Borja",
  "email": "ady.borja@hotmail.com",
  "telefono": "+0996333313",
  "biografia": "Autora ecuatoriana. Nacida el 10 de febrero de 1195 en la ciudad de Guayaquil. Amante de la tecnología y la literatura.",
  "generoliterario": "Tecnología"
}

```

Fig.6. Método GET por ID en POSTMAN

- GET /authors: <http://localhost:8081/authors>  
Lista a todos los autores registrados.

```

1  [
2    {
3      "id": "db38cbf5-7e8c-4e26-ba99-e3148aa6462b",
4      "tipoIdentificacion": "CEDULA",
5      "identificacion": "1721817029",
6      "nacionalidad": "ecuatoriana",
7      "nombre": "Adriana",
8      "apellido": "Borja",
9      "email": "adry.borja@hotmail.com",
10     "telefono": "0986633333",
11     "biografia": "Autora ecuatoriana. Nacida el 10 de febrero de 1195 en la ciudad de Guayaquil. Amante de la tecnologia y la literatura.",
12     "generoliterario": "Tecnologia"
13   },
14   {
15     "id": "8dd30986-8213-4dee-9fcf-5a2af7de427c",
16     "tipoIdentificacion": "CEDULA",
17     "identificacion": "1721817011",
18     "nacionalidad": "ecuatoriana",
19     "nombre": "Jessica",
20     "apellido": "Borja",
21     "email": "jessica.borja@hotmail.com",
22     "telefono": "09866383485",
23     "biografia": "Autora ecuatoriana. Amante de la poesia.",
24     "generoliterario": "Poesia"
25   },
26   {
27     "id": "fdda9594-baec-41c5-90df-5ff0c3f68d59",
28     "tipoIdentificacion": "CEDULA",
29     "identificacion": "1722893636",
30     "nacionalidad": "argentina",
31     "nombre": "Rosa",
32     "apellido": "Diaz",

```

Fig.7. Listar todos los autores en POSTMAN

## 2.6. Patrones de Diseño Aplicados

### 2.6.1. Repository Pattern

El patrón Repository se aplica en la interfaz AuthorRepository, la cual extiende JpaRepository. Esta capa abstrae el acceso a la base de datos y encapsula las operaciones de persistencia, evitando que la lógica de negocio interactúe directamente con JPA o sentencias SQL.

Su uso permite desacoplar la capa de servicios de los detalles de almacenamiento, facilitando el mantenimiento y la escalabilidad del sistema. Métodos como existsByEmail y existsByIdentificacion permiten validar reglas del dominio desde el repositorio sin exponer la lógica de persistencia al resto de la aplicación.

```

package com.example.authors.repository;

import com.example.authors.entity.Author;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.UUID;

public interface AuthorRepository extends JpaRepository<Author, UUID> {
    boolean existsByEmail(String email);
    boolean existsByIdentificacion(String identificacion);
}

```

Fig.8. Interface AuthorRepository aplicando Repository Pattern

## 2.7. Manejo de Errores y Validaciones

El microservicio implementa un manejo centralizado de errores mediante GlobalExceptionHandler (anotación @RestControllerAdvice), con el objetivo de devolver respuestas HTTP uniformes y controladas.

Se manejan los siguientes casos:

- **400 BAD\_REQUEST:** errores de validación de entrada (@Valid) capturados con MethodArgumentNotValidException, devolviendo los campos inválidos y sus mensajes.
- **404 NOT\_FOUND:** cuando el autor no existe, usando la excepción personalizada NotFoundException.
- **409 CONFLICT:** conflictos por reglas del dominio (por ejemplo, correo o identificación duplicada) usando ConflictException.
- **409 CONFLICT (DB):** conflictos detectados directamente en base de datos (DataIntegrityViolationException) como respaldo ante restricciones de integridad.
- **500 INTERNAL\_SERVER\_ERROR:** cualquier error no controlado se captura de forma general para evitar respuestas inconsistentes.

Este enfoque asegura validación, control de excepciones y respuestas claras para el cliente.

```

// 400 - Conflict
ExceptionHandler<ConflictException> handleConflict(ConflictException ex) {
    Map<String, Object> body = new HashMap<>();
    body.put("timestamp", LocalDateTime.now());
    body.put("status", HttpStatus.CONFLICT.value());
    body.put("error", "CONFLICT");
    body.put("message", ex.getMessage());
    return ResponseEntity.status(HttpStatus.CONFLICT).body(body);
}

// 409 - Conflicto por base de datos
ExceptionHandler<DataIntegrityViolationException> handleDataIntegrity(DataIntegrityViolationException ex) {
    Map<String, Object> body = new HashMap<>();
    body.put("timestamp", LocalDateTime.now());
    body.put("status", HttpStatus.CONFLICT.value());
    body.put("error", "CONFLICT");
    body.put("message", "Conflict: dato duplicado o restriccion de integridad violada");
    return ResponseEntity.status(HttpStatus.CONFLICT).body(body);
}

// 404 - Autor no encontrado
ExceptionHandler<NotFoundException> handleNotFound(NotFoundException ex) {
    Map<String, Object> body = new HashMap<>();
    body.put("timestamp", LocalDateTime.now());
    body.put("status", HttpStatus.NOT_FOUND.value());
    body.put("error", "NOT_FOUND");
    body.put("message", ex.getMessage());
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
}

```

Fig.9. GlobalExceptionHandler manejo consistente de errores

```

package com.example.authors.exception;

public class ConflictException extends RuntimeException { }

public class NotFoundException extends RuntimeException { }

```

Fig.10. Manejo consistente de errores

### 3. Microservicio Publications Service

#### 3.1. Descripción del Microservicio

El microservicio Publications Service es responsable de gestionar las publicaciones y su estado editorial dentro de la plataforma. Este microservicio depende del microservicio de Autores para validar la existencia del autor asociado a cada publicación, evitando dependencias circulares.

Su responsabilidad incluye la creación de publicaciones, la consulta de información y la gestión controlada del estado editorial a lo largo del ciclo de vida del contenido.

#### 3.2. Modelo de Dominio

El dominio del microservicio de Publicaciones se diseñó con un enfoque centrado en la entidad abstracta Publication, la cual representa el concepto principal del proceso editorial. Esta clase encapsula los atributos comunes del ciclo de vida de una publicación, como el título, el identificador del autor, el estado editorial y las marcas temporales de creación, actualización y publicación.

A partir de esta clase base se deriva la entidad PublicationContent, que incorpora la información específica del contenido, como el resumen, el contenido textual, la categoría y el tipo de publicación. Este diseño permite representar distintos formatos editoriales sin acoplar el dominio a un tipo específico de publicación.

El estado editorial se modela mediante el enumerado EditorialStatus, mientras que el tipo de publicación se define mediante el enumerado TipoPublicacion, manteniendo coherencia con el lenguaje del dominio.

The screenshot shows a code editor with four tabs open, each containing Java code:

- Publication.java**: An abstract class with annotations like @Entity, @Table(name = "publications"), and @Inheritance(strategy = SUPERCLASS\_TABLE). It has fields for id (UUID), summary (String), content (String), category (String), and status (EditorialStatus).
- PublicationContent.java**: A concrete class extending Publication. It has fields for title (String), authorId (UUID), author (Author), createdAt (LocalDateTime), updatedAt (LocalDateTime), publishedAt (LocalDateTime), and type (TipoPublicacion).
- EditorialStatus.java**: An enum with five values: DRAFT, IN\_REVIEW, APPROVED, PUBLISHED, and REJECTED.
- TipoPublicacion.java**: An enum with six values: ARTICULO, LIBRO, INVESTIGACION, INFORME, and OTRO.

Fig.11. Modelo de dominio del microservicio Publications Service

### 3.3. Estructura – Separación por Capas

El microservicio Publications Service fue estructurado siguiendo una arquitectura por capas, alineada con los principios SOLID y las buenas prácticas de diseño en aplicaciones basadas en microservicios.

Esta organización permite una clara separación de responsabilidades, facilitando el mantenimiento, la escalabilidad y la comprensión del sistema.

La estructura principal del proyecto es la siguiente:

- **Controller:** Contiene los controladores REST encargados de exponer los endpoints HTTP del microservicio. Esta capa recibe las solicitudes del cliente y delega la lógica de negocio a la capa de servicios, manteniendo los controladores livianos y enfocados únicamente en la gestión de peticiones y respuestas.
- **Dto:** Incluye los Data Transfer Objects utilizados para la entrada y salida de datos (PublicationRequestDTO y PublicationResponseDTO). Su uso evita exponer directamente las entidades del dominio y permite un mayor control y validación de la información intercambiada entre el cliente y el backend.
- **Entity:** Define el modelo de dominio del microservicio, incluyendo la clase abstracta base Publication, la clase derivada PublicationContent y los enumerados EditorialStatus y TipoPublicacion. Estas clases representan la estructura persistente de la base de datos y modelan el ciclo de vida editorial de una publicación.
- **Repository:** Implementa el acceso a datos mediante Spring Data JPA. El repositorio encapsula las operaciones de persistencia y consulta de publicaciones, aplicando el Repository Pattern para desacoplar la lógica de negocio de los detalles de acceso a la base de datos.
- **Service:** Contiene la lógica de negocio del microservicio. La interfaz PublicationService y su implementación PublicationServiceImpl actúan como una fachada entre los controladores y la capa de persistencia, coordinando además la aplicación de patrones de diseño como Factory, Strategy y Adapter.
- **Exception:** Centraliza el manejo de errores del sistema mediante excepciones personalizadas (NotFoundException, ConflictException) y un GlobalExceptionHandler, garantizando respuestas HTTP consistentes y controladas ante distintos escenarios de error.

Esta organización permite cumplir con el principio de Single Responsibility, ya que cada capa tiene una función claramente definida dentro del microservicio de Publicaciones.

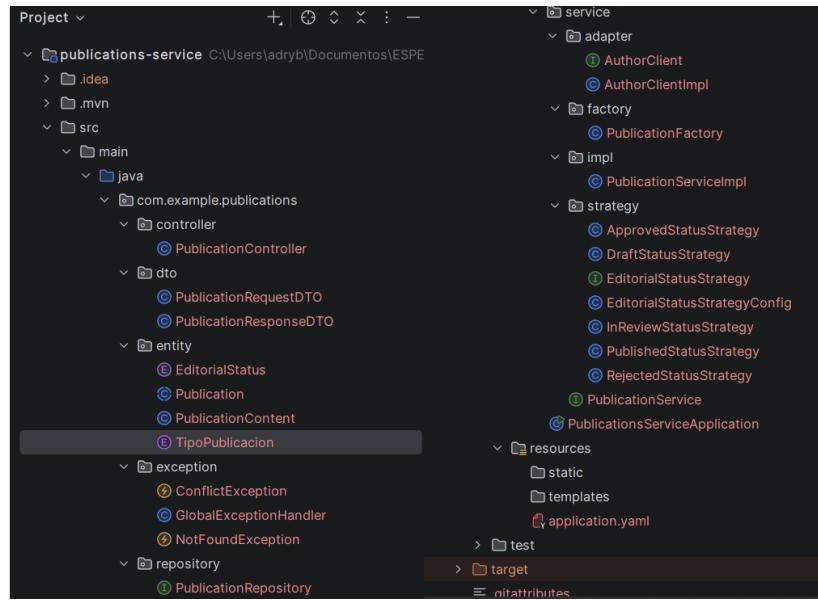


Fig.12. Estructura Microservicio Publications Service

### 3.4. Persistencia y ORM

La persistencia se implementa mediante JPA/Hibernate utilizando una base de datos MySQL independiente (db-publications). Hibernate se encarga de generar automáticamente las tablas a partir del modelo de dominio.

	<a href="#">id</a>	<a href="#">author_id</a>	<a href="#">created_at</a>	<a href="#">published_at</a>	<a href="#">status</a>	<a href="#">title</a>	<a href="#">updated_at</a>	<a href="#">category</a>	<a href="#">content</a>	<a href="#">summary</a>	<a href="#">tipo_publicacion</a>
<input type="checkbox"/>	<a href="#">Editar</a> <a href="#">Copiar</a> <a href="#">Borrar</a> 0x527e4247-c54d42cb9711f25939c75697	0x7e291eebb666424e858bc5a0c175075	2026-01-26 21:55:37 000000	NULL	DRAFT	Prueba	NULL	Tecnologia	prueba.....	prueba	ARTICULO
<input type="checkbox"/>	<a href="#">Editar</a> <a href="#">Copiar</a> <a href="#">Borrar</a> 0xd2bb643a33a44140eb36712f22884696	0x0db30c5767e04e20b09e14baaa0f462b	2026-01-26 21:45:51 000000	NULL	DRAFT	Prueba	NULL	Tecnologia	prueba.....	prueba	ARTICULO
<input type="checkbox"/>	<a href="#">Editar</a> <a href="#">Copiar</a> <a href="#">Borrar</a> 0x0bd3d4213a4624b48958f7eac223a5cd5	0xdd095940aec41c5900590c3f88e59	2026-01-27 00:18:05 000000	PUBLISHED	Prueba	2026-01-27 00:41:11 000000	PUBLISHED	Tecnologia	prueba.....	prueba	ARTICULO

Fig.13. Base de datos MYSQL y tabla publications

### 3.5. API REST – Endpoints Principales

El microservicio expone los siguientes endpoints REST:

- POST /publications: <http://localhost:8082/publications>

Permite registrar una nueva publicación validando con el Id de Author.

Al crear una publicación, el estado editorial se inicializa automáticamente como DRAFT mediante un método @PrePersist, garantizando que todo contenido siga el flujo editorial definido. El cambio de estado se gestiona posteriormente a través de un endpoint específico, aplicando el patrón Strategy para controlar las transiciones de estado.

- Ejemplo de body:

```
{
  "title": "Prueba",
  "authorId": "fdda9594-baec-41c5-90df-5ff0c3f68d59",
  "summary": "prueba",
  "content": "prueba.....",
  "category": "Tecnología",
  "tipoPublicacion": "ARTICULO"
}
```

- Respuesta JSON:

The screenshot shows the Postman interface for a POST request to 'http://localhost:8082/publications'. The request body contains the JSON data from the previous code block. The response tab shows a 201 Created status with the following JSON data:

```

1 {
2   "id": "82bb43a-83a4-41de-b367-21f228846f6b",
3   "title": "Prueba",
4   "authorId": "db38cbf5-7e8c-4e26-ba99-e3148aa6462b",
5   "summary": "prueba",
6   "content": "prueba.....",
7   "category": "Tecnología",
8   "tipoPublicacion": "ARTICULO",
9   "status": "DRAFT",
10  "publishedAt": null
}

```

Fig.14. Creación de publicación desde POSTMAN

- GET /publications/{id}: <http://localhost:8082/publications/{id}>  
Obtiene la información de una publicación por su identificador.

The screenshot shows the Postman interface for a GET request to 'http://localhost:8082/publications/527e4247-c54d-42cb-9711-f35939c75692'. The request includes a 'Key' parameter in the query params table. The response tab shows a 200 OK status with the following JSON data:

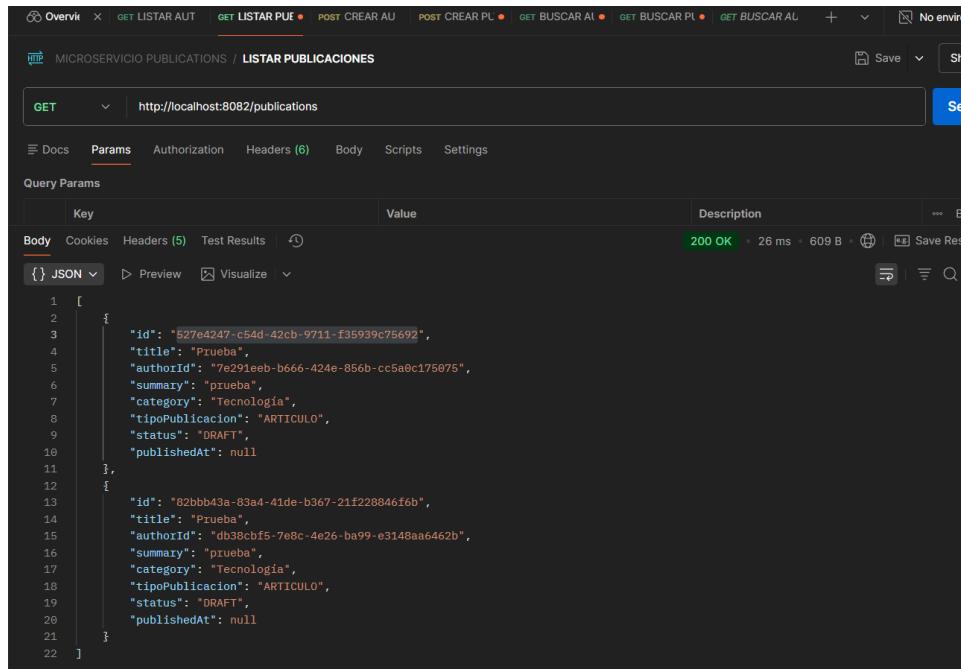
```

1 {
2   "id": "527e4247-c54d-42cb-9711-f35939c75692",
3   "title": "Prueba",
4   "authorId": "7e291eeb-b666-424e-856b-cc5a0c175075",
5   "summary": "prueba",
6   "category": "Tecnología",
7   "tipoPublicacion": "ARTICULO",
8   "status": "DRAFT",
9   "publishedAt": null
10 }

```

Fig.15. Método GET para obtener una publicación por id en POSTMAN

- GET /publications: <http://localhost:8082/publications>  
Lista a todas las publicaciones registradas.



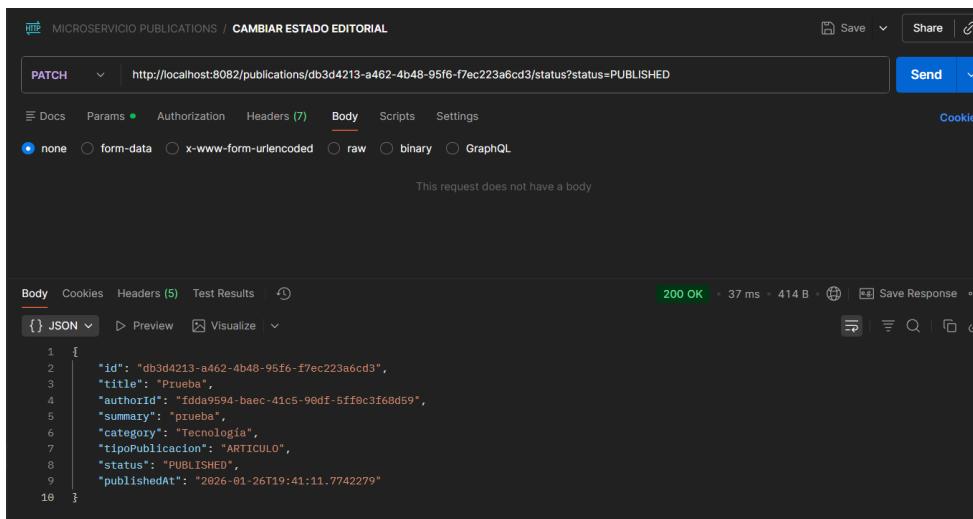
```

1 [
2   {
3     "id": "527e4247-c54d-42cb-9711-f35939c75692",
4     "title": "Prueba",
5     "authorId": "7e293eeb-b666-424e-856b-cc5a0c175075",
6     "summary": "prueba",
7     "category": "Tecnología",
8     "tipoPublicacion": "ARTICULO",
9     "status": "DRAFT",
10    "publishedAt": null
11  },
12  {
13    "id": "82bbb43a-83a4-41de-b367-21f228846f6b",
14    "title": "Prueba",
15    "authorId": "db38cbf5-7e8c-4e26-ba99-e3148aa6462b",
16    "summary": "prueba",
17    "category": "Tecnología",
18    "tipoPublicacion": "ARTICULO",
19    "status": "DRAFT",
20    "publishedAt": null
21  }
22 ]

```

Fig.16. Listar todas las publicaciones en POSTMAN

- PATCH /{id}/status?status=STATUS:  
<http://localhost:8082/publications/{id}/status?status=PUBLISHED>  
Cambiamos el estado editorial de una publicación



```

1 {
2   "id": "db3d4213-a462-4b48-95f6-f7ec223a6cd3",
3   "title": "Prueba",
4   "authorId": "7da9594-baec-41c5-90df-5ff0c3f68d59",
5   "summary": "prueba",
6   "category": "Tecnología",
7   "tipoPublicacion": "ARTICULO",
8   "status": "PUBLISHED",
9   "publishedAt": "2026-01-26T19:41:11.7742279"
10  }

```

Fig.17. Cambio de status editorial en POSTMAN

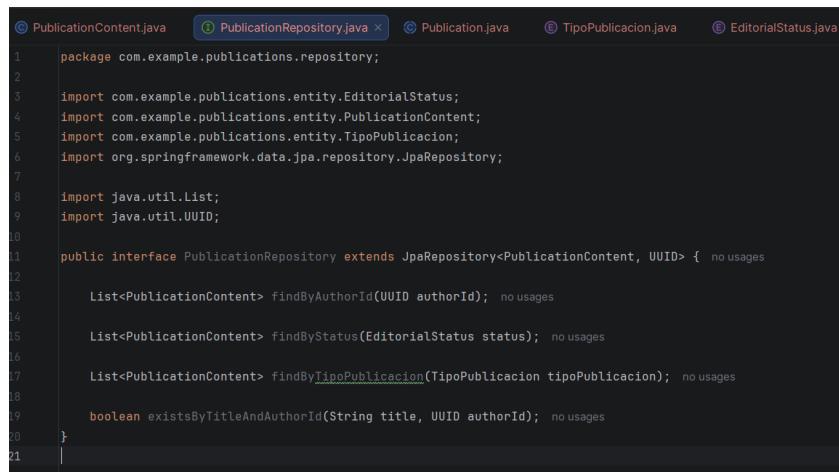
### 3.6. Patrones de Diseños Aplicados

En el microservicio Publications Service se aplicaron múltiples patrones de diseño con el objetivo de cumplir principios de ingeniería de software como bajo acoplamiento, alta cohesión, extensibilidad y mantenibilidad. A continuación, se describen los principales patrones utilizados, indicando su ubicación en el código y la justificación de su uso.

### 3.6.1. Repository Pattern

El patrón Repository se implementó mediante la interfaz PublicationRepository, la cual extiende JpaRepository proporcionado por Spring Data JPA. Este componente encapsula el acceso a la base de datos y abstrae las operaciones de persistencia y consulta de publicaciones, evitando que la lógica de negocio interactúe directamente con la capa de acceso a datos o con sentencias SQL.

Gracias a este patrón, la capa de servicios permanece desacoplada de los detalles de almacenamiento, facilitando el mantenimiento del sistema y permitiendo cambios futuros en la tecnología de persistencia sin afectar la lógica de negocio.



The screenshot shows a Java code editor with the following code:

```
1 package com.example.publications.repository;
2
3 import com.example.publications.entity.EditorialStatus;
4 import com.example.publications.entity.PublicationContent;
5 import com.example.publications.entity.TipoPublicacion;
6 import org.springframework.data.jpa.repository.JpaRepository;
7
8 import java.util.List;
9 import java.util.UUID;
10
11 public interface PublicationRepository extends JpaRepository<PublicationContent, UUID> { no usages
12
13     List<PublicationContent> findByAuthorId(UUID authorId); no usages
14
15     List<PublicationContent> findByStatus(EditorialStatus status); no usages
16
17     List<PublicationContent> findByTipoPublicacion(TipoPublicacion tipoPublicacion); no usages
18
19     boolean existsByTitleAndAuthorId(String title, UUID authorId); no usages
20 }
21 |
```

Fig.18. Interface PublicationRepository aplicando Repository Pattern

### 3.6.2. Factory Method

El patrón Factory Method se aplicó en la clase PublicationFactory, la cual centraliza la creación de objetos del dominio PublicationContent a partir de los datos recibidos en el PublicationRequestDTO.

Este enfoque evita que la lógica de instanciación se disperse en la capa de servicios y permite encapsular reglas de inicialización como la asignación de valores por defecto y la construcción coherente del objeto de dominio. Además, facilita la extensibilidad del sistema ante la incorporación de nuevos tipos de publicaciones sin modificar la lógica existente.

```

1 package com.example.publications.service.factory;
2
3 import com.example.publications.dto.PublicationRequestDTO;
4 import com.example.publications.entity.PublicationContent;
5 import org.springframework.stereotype.Component;
6
7 @Component
8 public class PublicationFactory {
9
10    @Override
11    public PublicationContent create(PublicationRequestDTO dto) { no usages 1 related problem
12
13        PublicationContent publication = new PublicationContent();
14        publication.setTitle(dto.getTitle());
15        publication.setAuthorId(dto.getAuthorId());
16        publication.setSummary(dto.getSummary());
17        publication.setContent(dto.getContent());
18        publication.setCategory(dto.getCategory());
19        publication.setTipoPublicacion(dto.getTipoPublicacion());
20
21        return publication;
22    }
23}

```

Fig.19. Clase PublicationFactory aplicando Factory Method

### 3.6.3. Strategy Pattern (Estados Editoriales)

El patrón Strategy se utilizó para encapsular el comportamiento asociado a los distintos estados editoriales de una publicación. Cada estado del flujo editorial (DRAFT, IN REVIEW, APPROVED, PUBLISHED y REJECTED) se implementa como una estrategia independiente que define cómo debe comportarse la publicación al cambiar a dicho estado.

Este diseño evita el uso de estructuras condicionales extensas (if/else o switch) en la lógica de negocio y facilita la incorporación de nuevos estados editoriales sin modificar el código existente. La selección dinámica de la estrategia adecuada se realiza en la capa de servicios, promoviendo un diseño abierto a extensión y cerrado a modificación.

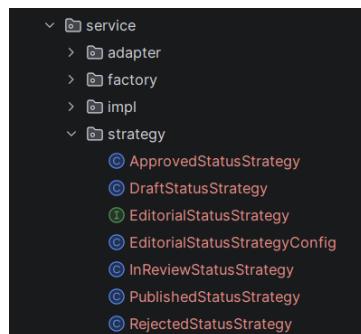


Fig.20. Paquete strategy para aplicar el Patrón Strategy

### 3.6.4. Adapter Pattern – AuthorClient

El patrón Adapter se implementó para gestionar la comunicación entre el microservicio de Publicaciones y el microservicio de Autores. Esto se realiza mediante la interfaz AuthorClient, la cual define las operaciones necesarias para verificar la existencia de un autor sin acoplar directamente el código a la implementación interna del microservicio externo.

La clase AuthorClientImpl actúa como adaptador concreto, encapsulando la lógica de comunicación HTTP mediante un cliente REST. Este patrón permite que el microservicio de Publicaciones dependa únicamente de una abstracción, facilitando cambios futuros en la forma de comunicación (por ejemplo, uso de Feign o mensajería) sin afectar la lógica de negocio.

```

package com.example.publications.service.adapter;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.RestTemplate;

import java.util.UUID;

@Component
@RequiredArgsConstructor
public class AuthorClientImpl implements AuthorClient {

    private final RestTemplate restTemplate;

    private static final String AUTHORS_SERVICE_URL = "usage"
        "http://localhost:8081/authors/{id}";

    @Override
    public boolean existsById(UUID authorId) {
        try {
            restTemplate.getForEntity(AUTHORS_SERVICE_URL, Void.class, authorId);
            return true;
        } catch (HttpClientErrorException.NotFound ex) {
            return false;
        }
    }
}

package com.example.publications.service.adapter;

import java.util.UUID;

public interface AuthorClient {
    boolean existsById(UUID authorId);
}

```

Fig.21. Paquete adapter para aplicar el Adapter Pattern

La aplicación conjunta de los patrones Repository, Factory Method, Strategy, Facade y Adapter permite que el microservicio Publications Service mantenga un diseño desacoplado, extensible y alineado con los principios SOLID. Esta arquitectura facilita la evolución del sistema, el mantenimiento del código y la correcta integración entre microservicios dentro de la plataforma editorial.

### 3.7. Manejo de Errores y Validaciones

El microservicio Publications Service implementa un manejo centralizado de errores y validaciones mediante la clase GlobalExceptionHandler, anotada con @RestControllerAdvice, con el objetivo de garantizar respuestas HTTP consistentes, controladas y alineadas a las reglas del dominio.

Las validaciones de entrada se realizan a través de anotaciones (@Valid) en los DTOs, capturando errores de formato o campos inválidos mediante la excepción MethodArgumentNotValidException, devolviendo mensajes claros por campo con el código HTTP 400 BAD\_REQUEST. Adicionalmente, se controla el ingreso de datos mal

formados, como identificadores o estructuras JSON inválidas, utilizando `HttpMessageNotReadableException`, evitando fallos internos del sistema.

Para la gestión de reglas de negocio, se emplean excepciones personalizadas como NotFoundException y ConflictException. La primera se utiliza cuando una publicación o autor asociado no existe, devolviendo un 404 NOT\_FOUND, mientras que la segunda gestiona conflictos de negocio, como intentos de crear publicaciones duplicadas o transiciones editoriales no válidas, respondiendo con 409 CONFLICT. Como mecanismo de respaldo, se maneja también DataIntegrityViolationException para capturar violaciones de integridad detectadas directamente en la base de datos.

Finalmente, cualquier error no controlado es capturado por un manejador genérico de excepciones, retornando un 500 INTERNAL\_SERVER\_ERROR con un mensaje estándar, evitando la exposición de detalles internos del sistema. Este enfoque asegura robustez, coherencia en las respuestas y una clara separación entre la lógica de negocio y la capa de presentación.

Fig.22. GlobalExceptionHandler manejo consistente de erros

```
GlobalExceptionHandler.java ConflictException.java NotFoundException.java

1 package com.example.publications.exception;
2
3 public class ConflictException extends RuntimeException {
4     public ConflictException(String message) { super(message); }
5 }

GlobalExceptionHandler.java ConflictException.java NotFoundException.java

1 package com.example.publications.exception;
2
3 public class NotFoundException extends RuntimeException {
4     public NotFoundException(String message) { super(message); }
5 }
```

Fig.23. Manejo consistente de erros

#### 4. Frontend Web

El frontend fue desarrollado utilizando React y una biblioteca de interfaz enriquecida (Material UI), permitiendo construir una aplicación web moderna, reutilizable y visualmente consistente, manteniendo la separación entre lógica y presentación.

Se implementó un sistema de tematización global mediante `ThemeProvider`, con una configuración centralizada en un archivo independiente (`theme.js`), lo que garantiza coherencia visual y facilita el mantenimiento de la interfaz.

La navegación se gestiona mediante `React Router`, permitiendo una separación clara de vistas (Home, Autores y Publicaciones) bajo un enfoque de aplicación de una sola página (SPA).

La composición de información entre autores y publicaciones se realiza en el frontend a partir de la comunicación con los microservicios correspondientes, evitando dependencias circulares y respetando los principios de una arquitectura desacoplada.

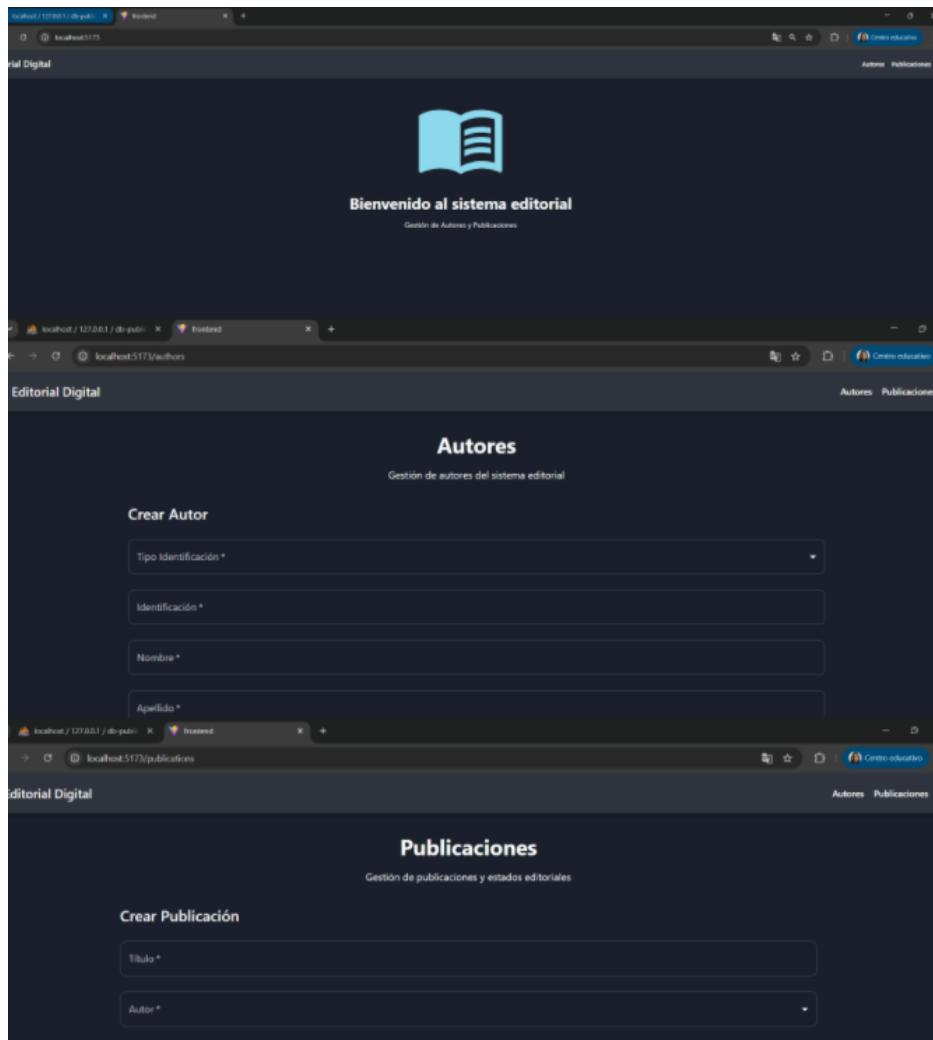


Fig.24. Páginas frontend Editorial Digital

##### 4.1. Funcionalidades implementadas

El frontend implementa las siguientes funcionalidades:

- Creación y listado de autores.
- Consulta de autores por identificador único (UUID).

Nombre	Identificación	Nacionalidad	Email	Teléfono	Género	Acciones
Adriana Borja	CEDULA - 1721817029	ecuatoriana	ady.borja@hotmail.com	0986333313	Tecnología	Ver detalle
Jessica Borja	CEDULA - 1721817011	ecuatoriana	jessica.borja@hotmail.com	0986838485	Poesía	Ver detalle
Rosa Diaz	CEDULA - 1712093630	argentina	rosa.diaz@hotmail.com	0999999999	Literatura Griega	Ver detalle
Carlos Diaz	PASAPORTE - ABC123456	colombiano	carlos.diaz@hotmail.com	+573101234567	Literatura Griega	Ver detalle
Martin Lopez	PASAPORTE - ABC123456789	colombiano	martin.lopez@hotmail.com	+573101234599	Literatura Griega	Ver detalle
Ricardo Lopez	PASAPORTE - ABC1234567891	colombiano	ricardo.lopez@hotmail.com	+573101234599	Literatura Griega	Ver detalle

Fig.25.Funcionalidades implementadas en Autores.

- Creación y listado de publicaciones.
- Consulta del detalle de una publicación.
- Visualización del contenido completo de una publicación.
- Cambio del estado editorial de una publicación.

**Publicaciones**  
Gestión de publicaciones y estados editoriales

### Crear Publicación

Título \*

Autor \*

Resumen \*

Contenido \*

Tipo de publicación \*

Categoría \*



### Lista de Publicaciones

Buscar publicación por ID (UUID)  Q Buscar

Título	Autor	Tipo	Categoría	Resumen	Estado	Fecha publicación	Acciones
Prueba	Ricardo Lopez	ARTICULO	Tecnología	prueba	REJECTED	-	 Ver contenido  Cambiar estado
Prueba	Adriana Borja	ARTICULO	Tecnología	prueba	PUBLISHED	28/1/2026	 Ver contenido  Cambiar estado
Arquitectura de Microservicios en Sistemas Editoriales	Ricardo Lopez	ARTICULO	Arquitectura de Software	Este artículo analiza la aplicación de arquitecturas basadas en microservicios dentro de plataformas editoriales modernas.	DRAFT	-	 Ver contenido  Cambiar estado

Fig.26.Funcionalidades implementadas en Publicaciones.

## 4.2. Gestión de Autores

El módulo de autores permite el registro de nuevos autores mediante formularios con validaciones básicas, así como la visualización de un listado general y la consulta del detalle de un autor utilizando su identificador único, consumiendo el microservicio de autores.

## 4.3. Gestión de Publicaciones

El módulo de publicaciones permite crear publicaciones asociadas a autores existentes, mostrar un listado con información relevante y gestionar el estado editorial.

Además, se implementa la visualización del contenido completo de una publicación mediante un modal, sin afectar el flujo principal de la interfaz.

## 5. Modelado BPMN con Camunda

El proceso editorial modelado en BPMN representa el flujo completo que sigue una publicación dentro de una editorial digital, desde la creación inicial del borrador por parte del autor hasta su publicación final o rechazo definitivo. Este modelo tiene como objetivo simular y validar el comportamiento del negocio antes de una automatización completa, permitiendo analizar las decisiones, roles involucrados y posibles escenarios del proceso editorial.

El proceso se estructura mediante lanes que representan claramente los roles participantes: Autor, Revisor y Editor, garantizando una adecuada separación de responsabilidades. Asimismo, se emplean tareas humanas (User Tasks) para actividades que requieren intervención de personas y tareas automáticas (Service Tasks) para acciones ejecutadas por el sistema, como notificaciones.

El modelo incorpora eventos intermedios de mensaje, los cuales simulan el envío y recepción de información entre los distintos actores, reflejando un entorno real donde la comunicación no es inmediata ni directa. Finalmente, un gateway exclusivo (XOR) permite tomar decisiones editoriales basadas en variables del proceso, como la aprobación, rechazo o solicitud de cambios, asegurando que el flujo continúe únicamente por una de las rutas posibles.

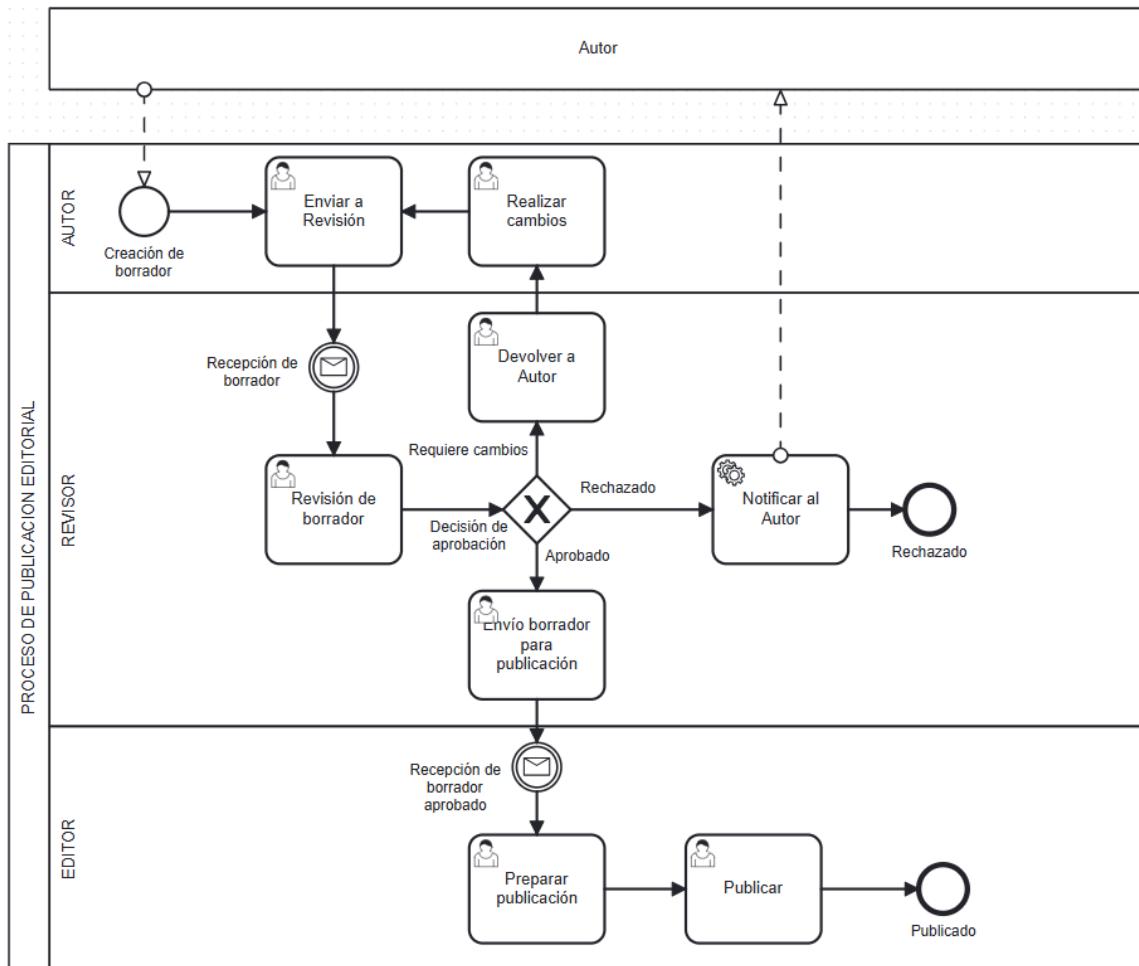


Fig.27. Modelado BPM de proceso de publicación editorial digital

### 5.1. Escenario 1 – Aprobación directa de la publicación

En el escenario de aprobación, el proceso inicia cuando el Autor crea el borrador de la publicación y lo envía a revisión. Una vez enviado, el Revisor recibe el borrador mediante un evento intermedio de mensaje, que representa la notificación o entrega del contenido para su análisis.

El Revisor realiza la revisión editorial del borrador y, tras evaluarlo, toma la decisión de aprobarlo en el gateway exclusivo de decisión editorial. Al ser aprobado, el borrador es enviado al Editor, quien recibe la notificación correspondiente y procede a preparar la publicación, realizando los ajustes finales necesarios para su difusión.

Posteriormente, el Editor ejecuta la tarea de publicar, concluyendo el proceso con el evento de fin “Publicado”, lo que indica que la publicación ha superado satisfactoriamente todas las etapas del proceso editorial.

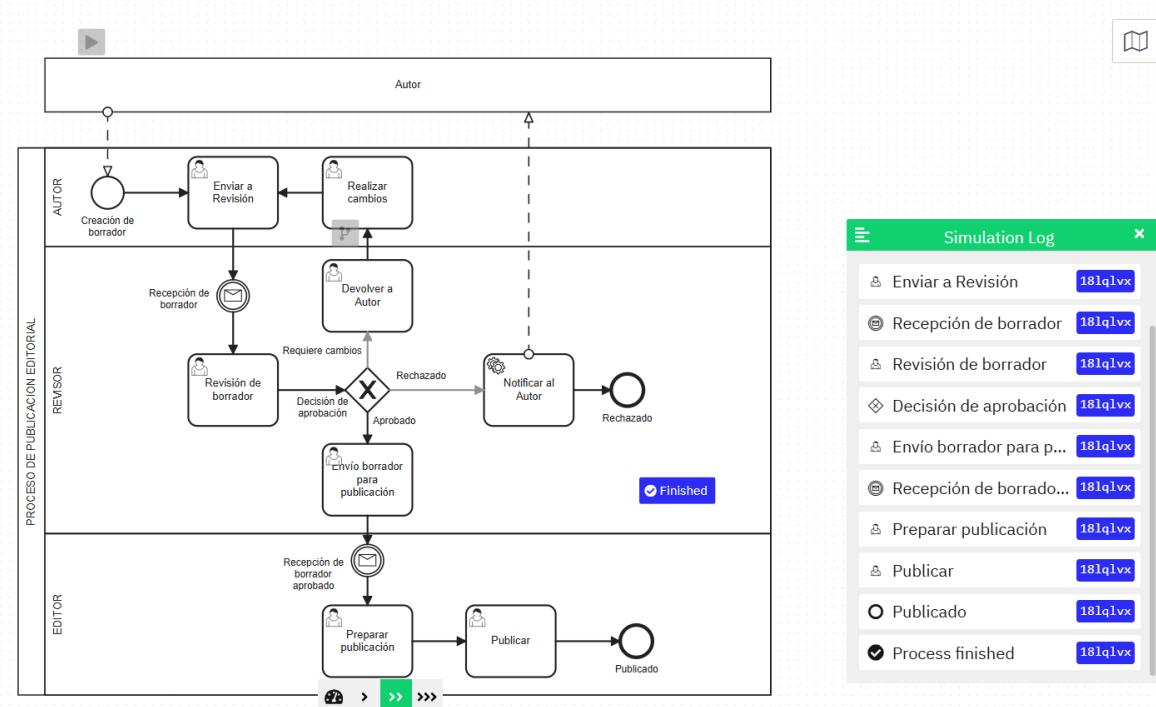


Fig.28. Modelado BPM escenario de aprobación

## 5.2. Escenario 2 – Rechazo de la publicación

En el escenario de rechazo, el proceso sigue inicialmente el mismo flujo: el Autor crea el borrador, lo envía a revisión y el Revisor recibe y analiza el contenido. Sin embargo, tras la revisión editorial, el Revisor determina que la publicación no cumple con los criterios establecidos.

Ante esta decisión, el flujo sigue la ruta de rechazo en el gateway exclusivo. En este caso, se ejecuta una tarea de servicio (Service Task) para notificar automáticamente al Autor sobre el rechazo de la publicación, simulando el envío de una notificación por correo u otro medio automatizado.

Una vez realizada la notificación, el proceso finaliza mediante el evento de fin “Rechazado”, cerrando formalmente el ciclo editorial para dicha publicación sin posibilidad de continuar el flujo.

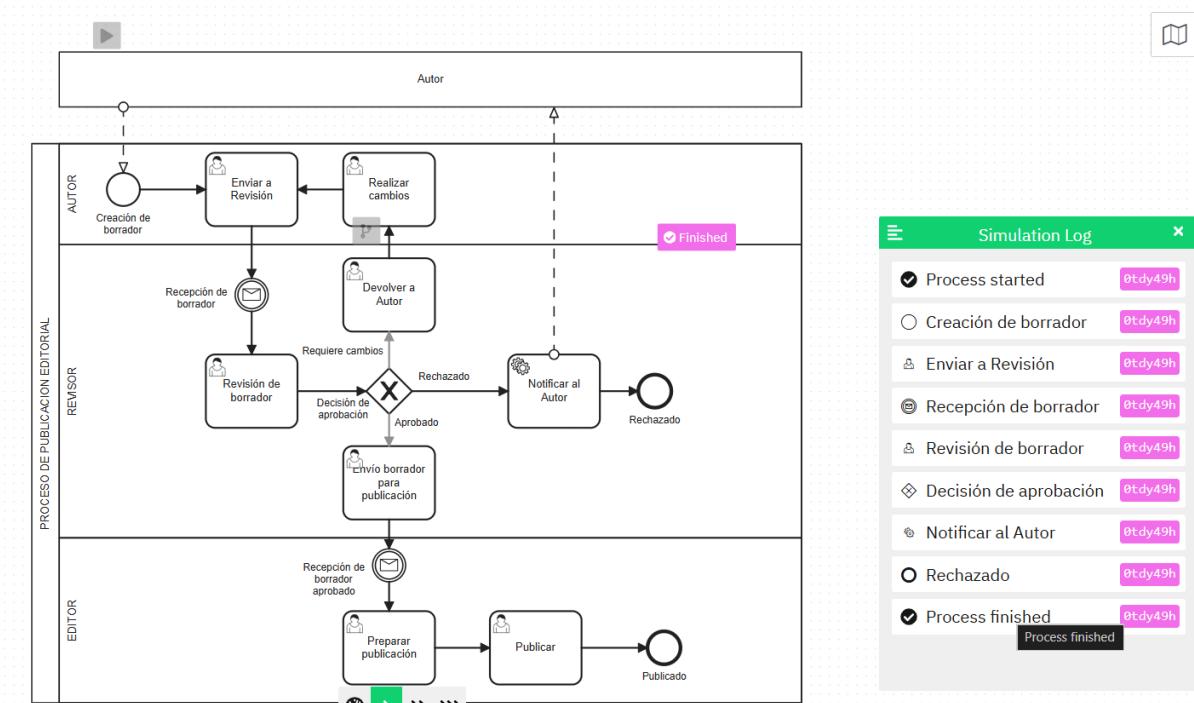


Fig.29. Modelado BPM escenario de rechazo

### 5.3. Escenario 3 – Requiere cambios y retrabajo

El escenario de retrabajo representa una situación común dentro de los procesos editoriales. Tras la creación del borrador por parte del Autor y su envío a revisión, el Revisor evalúa el contenido y determina que la publicación requiere cambios antes de poder ser aprobada o rechazada definitivamente.

En este caso, el flujo del proceso se dirige hacia la tarea “Devolver a Autor”, mediante la cual el Revisor envía observaciones y comentarios al Autor. El Autor recibe el borrador y realiza la tarea “Realizar cambios”, ajustando el contenido conforme a las observaciones recibidas.

Una vez completadas las modificaciones, el Autor vuelve a enviar el borrador a revisión, reiniciando el ciclo de evaluación. Este escenario permite que el proceso regrese nuevamente al punto de decisión editorial, donde el Revisor puede optar por aprobar, rechazar o solicitar nuevos cambios, simulando iteraciones reales del proceso editorial.

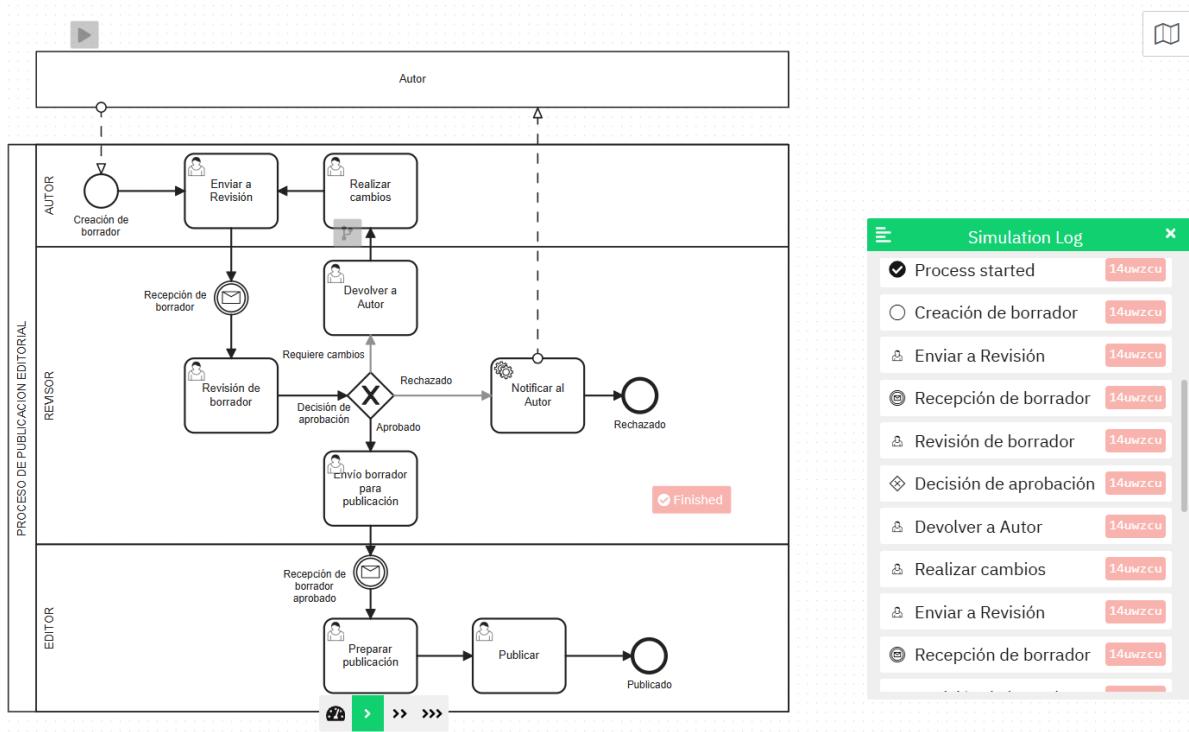


Fig.30. Modelado BPM escenario de requiere cambios y retrabajo

### Uso de eventos y variables para simulación

El modelo BPMN incorpora variables de proceso, tales como aprobado y requiereCambios, las cuales son utilizadas en el gateway exclusivo para dirigir el flujo hacia el escenario correspondiente durante la Token Simulation. Esto permite simular múltiples ejecuciones del proceso y validar el comportamiento del sistema ante diferentes decisiones editoriales.

### Valor agregado del modelo BPMN

El modelo desarrollado aporta un mayor nivel de realismo al proceso editorial al:

- Diferenciar claramente entre tareas humanas y automáticas.
- Incorporar comunicación asincrónica mediante eventos de mensaje.
- Permitir iteraciones del proceso mediante rutas de retrabajo.
- Facilitar la simulación de escenarios reales antes de una implementación técnica.

Este enfoque garantiza una correcta validación del flujo de negocio y una base sólida para futuras integraciones o automatizaciones.

## 6. Despliegue con Docker Compose

### 6.1. Objetivos del despliegue

El objetivo del despliegue con Docker Compose fue unificar la ejecución de todos los componentes del sistema editorial en un entorno controlado, reproducible y desacoplado del sistema operativo del desarrollador. Mediante Docker se garantiza que el frontend, los microservicios backend y las bases de datos se ejecuten con configuraciones consistentes, facilitando la instalación, pruebas y evaluación del proyecto.

Este enfoque permite levantar toda la solución mediante un único comando, cumpliendo con los principios de portabilidad y automatización del despliegue.

## 6.2. Servicios definidos en Docker Compose

La solución fue orquestada mediante el archivo docker-compose.yml, ubicado en la raíz del proyecto, el cual define los siguientes servicios:

- **authors-service:** Microservicio backend desarrollado en Spring Boot encargado de la gestión de autores.
- **publications-service:** Microservicio backend desarrollado en Spring Boot encargado de la gestión de publicaciones y estados editoriales.
- **frontend:** Aplicación web desarrollada en React + Material UI.
- **db-authors:** Base de datos PostgreSQL exclusiva para el microservicio de autores.
- **db-publications:** Base de datos MySQL exclusiva para el microservicio de publicaciones.

Cada servicio se ejecuta dentro de su propio contenedor, comunicándose a través de una red interna definida por Docker.

```

# docker-compose.yml
# docker-compose.yml
  version: "3.9"
  networks:
    editorial-net:
      driver: bridge
  volumes:
    authors-db-data:
    publications-db-data:
  services:
    # -----
    # DATABASE AUTHORS (PostgreSQL)
    db-authors:
      image: postgres:15
      container_name: db-authors
      env_file:
        - .env
      environment:
        POSTGRES_DB: ${POSTGRES_DB}
        POSTGRES_USER: ${POSTGRES_USER}
        POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      volumes:
        - authors-db-data:/var/lib/postgresql/data
      networks:
        - editorial-net
      healthcheck:
        test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER}"]
        interval: 10s
        timeout: 5s
        retries: 5
    # -----
    # DATABASE PUBLICATIONS (MySQL)
    db-publications:
      image: mysql:8.0
      container_name: db-publications
      env_file:
        - .env
      environment:
        MYSQL_DATABASE: ${MYSQL_DATABASE}
        MYSQL_USER: ${MYSQL_USER}
        MYSQL_PASSWORD: ${MYSQL_PASSWORD}
        MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      volumes:
        - publications-db-data:/var/lib/mysql
      networks:
        - editorial-net
      healthcheck:
        test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
        interval: 10s
        timeout: 5s
        retries: 5
    # =====
    # AUTHORS SERVICE
    # =====
    authors-service:
      build: ./authors-service
      container_name: authors-service
      env_file:
        - .env
      depends_on:
        db-authors:
          condition: service_healthy
      ports:
        - "8081:8081"
      networks:
        - editorial-net
      environment:
        SPRING_PROFILES_ACTIVE: docker
    # =====
    # PUBLICATIONS SERVICE
    # =====
    publications-service:
      build: ./publications-service
      container_name: publications-service
      env_file:
        - .env
      depends_on:
        db-publications:
          condition: service_healthy
        authors-service:
          condition: service_started
      ports:
        - "8082:8082"
      networks:
        - editorial-net
      environment:
        SPRING_PROFILES_ACTIVE: docker
    # -----
    # FRONTEND
    # -----
    frontend:
      build: ./frontend
      ports:
        - "5173:5173"
      env_file:
        - ./frontend/.env.docker
      depends_on:
        - authors-service
        - publications-service
      networks:
        - editorial-net

```

Fig.31. Archivo docker-compose.yml

### 6.3. Configuración de redes y persistencia de datos

Se definió una red personalizada de tipo bridge, denominada editorial-net, que permite la comunicación interna entre los contenedores sin exponer directamente los servicios internos.

Adicionalmente, se configuraron volúmenes para la persistencia de datos:

- **authors-db-data:** Persistencia de datos de PostgreSQL.
- **publications-db-data:** Persistencia de datos de MySQL.

Esto garantiza que la información no se pierda al detener o reconstruir los contenedores.

```
# =====#
# DATABASE AUTHORS (PostgreSQL)
# =====#
db-authors:
  image: postgres:15
  container_name: db-authors
  env_file:
    - .env
  environment:
    POSTGRES_DB: ${POSTGRES_DB}
    POSTGRES_USER: ${POSTGRES_USER}
    POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
  volumes:
    - authors-db-data:/var/lib/postgresql/data
  networks:
    - editorial-net
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER}"]
    interval: 10s
    timeout: 5s
    retries: 5
# =====#
# DATABASE PUBLICATIONS (MySQL)
# =====#
db-publications:
  image: mysql:8.0
  container_name: db-publications
  env_file:
    - .env
  environment:
    MYSQL_DATABASE: ${MYSQL_DATABASE}
    MYSQL_USER: ${MYSQL_USER}
    MYSQL_PASSWORD: ${MYSQL_PASSWORD}
    MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
  volumes:
    - publications-db-data:/var/lib/mysql
  networks:
    - editorial-net
  healthcheck:
    test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
    interval: 10s
    timeout: 5s
    retries: 5
```

Fig.31. Archivo docker-compose.yml configuración de bases de datos

### 6.4. Manejo de variables de entorno

Para evitar el uso de configuraciones sensibles directamente en el código fuente, se implementó el manejo de variables de entorno mediante archivos .env.

#### Archivos utilizados:

- **.env (raíz del proyecto):** Variables utilizadas por Docker Compose (credenciales de bases de datos).

```

.env
.
.env
1 # =====
2 # DATABASE AUTHORS (PostgreSQL)
3 # =====
4 POSTGRES_DB=db-authors
5 POSTGRES_USER=postgres
6 POSTGRES_PASSWORD=1995
7
8 # =====
9 # DATABASE PUBLICATIONS (MySQL)
10 # =====
11 MYSQL_DATABASE=db-publications
12 MYSQL_USER=publications
13 MYSQL_PASSWORD=1995
14 MYSQL_ROOT_PASSWORD=root
15

```

Fig.32. Archivo .env en raíz de proyecto con credenciales de bases de datos

- **.env.docker (frontend):** URLs de los microservicios cuando el frontend se ejecuta en Docker.

```

.env.docker
.
.env.docker
frontend > .env.docker
1 VITE_AUTHORS_API=http://localhost:8081
2 VITE_PUBLICATIONS_API=http://localhost:8082

```

Fig.33. Archivo .env.docker dentro de carpeta frontend

- **.env.local (frontend, opcional):** URLs de los microservicios para ejecución local.

```

.env.local
.
.env.local
frontend > .env.local
1 VITE_AUTHORS_API=http://localhost:8081
2 VITE_PUBLICATIONS_API=http://localhost:8082

```

Fig.34. Archivo .env.local dentro de carpeta frontend

Por buenas prácticas de seguridad, estos archivos no se incluyen en el repositorio Git, y se proporcionan archivos .env.example como referencia.

```

.env
.
.env
1 # =====
2 # DATABASE AUTHORS (PostgreSQL)
3 # =====
4 POSTGRES_DB=db-authors
5 POSTGRES_USER=postgres
6 POSTGRES_PASSWORD=1995
7
8 # =====
9 # DATABASE PUBLICATIONS (MySQL)
10 # =====
11 MYSQL_DATABASE=db-publications
12 MYSQL_USER=publications
13 MYSQL_PASSWORD=1995
14 MYSQL_ROOT_PASSWORD=root
15

.env.example
.
.env.example
frontend > .env.example
1 VITE_AUTHORS_API=http://localhost:8081
2 VITE_PUBLICATIONS_API=http://localhost:8082

```

Fig.35. Archivos .env.example

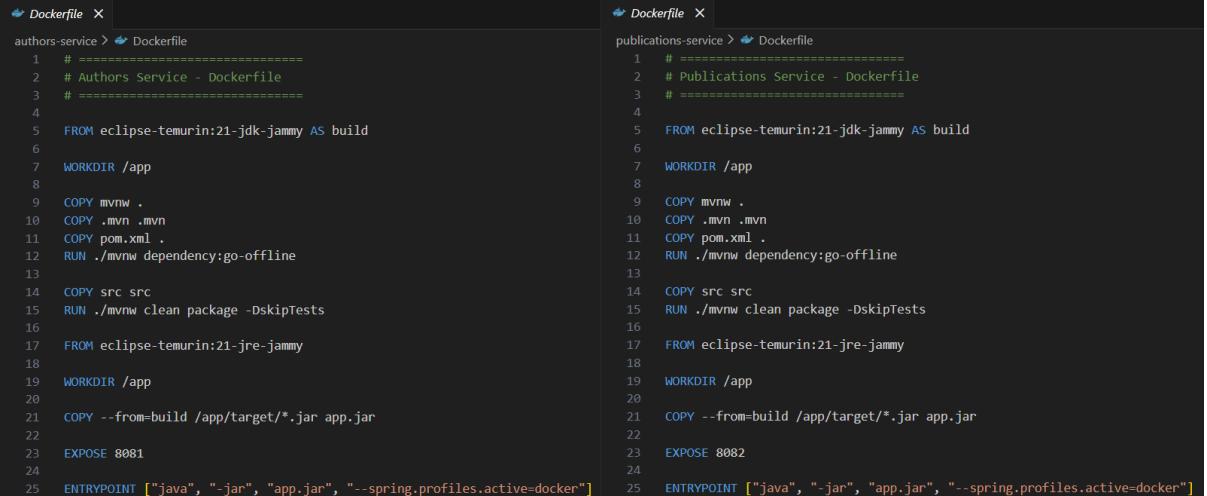
## 6.5. Dockerización de los microservicios backend

Cada microservicio backend cuenta con su propio Dockerfile, utilizando una estrategia de multi-stage build, la cual permite:

- Compilar el proyecto usando Maven.
- Generar el archivo .jar.
- Ejecutar la aplicación en una imagen liviana de Java (JRE).

Este enfoque reduce el tamaño final de la imagen y mejora el rendimiento del despliegue.

Los microservicios se configuran para usar perfiles específicos (docker) mediante la variable de entorno SPRING\_PROFILES\_ACTIVE.



```
authors-service > Dockerfile
1  # =====
2  # Authors Service - Dockerfile
3  #
4
5  FROM eclipse-temurin:21-jdk-jammy AS build
6
7  WORKDIR /app
8
9  COPY mvnw .
10 COPY .mvn .mvn
11 COPY pom.xml .
12 RUN ./mvnw dependency:go-offline
13
14 COPY src src
15 RUN ./mvnw clean package -DskipTests
16
17 FROM eclipse-temurin:21-jre-jammy
18
19 WORKDIR /app
20
21 COPY --from=build /app/target/*.jar app.jar
22
23 EXPOSE 8081
24
25 ENTRYPOINT ["java", "-jar", "app.jar", "--spring.profiles.active=docker"]

publications-service > Dockerfile
1  # =====
2  # Publications Service - Dockerfile
3  #
4
5  FROM eclipse-temurin:21-jdk-jammy AS build
6
7  WORKDIR /app
8
9  COPY mvnw .
10 COPY .mvn .mvn
11 COPY pom.xml .
12 RUN ./mvnw dependency:go-offline
13
14 COPY src src
15 RUN ./mvnw clean package -DskipTests
16
17 FROM eclipse-temurin:21-jre-jammy
18
19 WORKDIR /app
20
21 COPY --from=build /app/target/*.jar app.jar
22
23 EXPOSE 8082
24
25 ENTRYPOINT ["java", "-jar", "app.jar", "--spring.profiles.active=docker"]
```

Fig.36. Archivos Dockerfile de los microservicios

## 6.6. Configuración de bases de datos en contenedores

Las bases de datos se ejecutan como servicios independientes dentro de Docker:

- PostgreSQL para authors-service.
- MySQL para publications-service.

Cada microservicio se conecta únicamente a su base de datos correspondiente, evitando accesos cruzados y respetando el principio de independencia entre microservicios.

Las credenciales y nombres de bases de datos se inyectan mediante variables de entorno.

## 6.7. Comunicación entre microservicios en Docker

La comunicación entre los microservicios se realiza a través de la red interna de Docker, utilizando el nombre del servicio como hostname.

En el caso del microservicio de publicaciones, se valida la existencia del autor consultando al microservicio de autores mediante un adaptador, configurando la URL de acceso según el entorno:

- En Docker: <http://authors-service:8081>
- En local: <http://localhost:8081>

Este enfoque evita dependencias rígidas y permite cambiar de entorno sin modificar el código.

```
! application-docker.yml ×
publications-service > src > main > resources > ! application-docker.yml
1  spring:
2    application:
3      | name: publications-service
4
5    datasource:
6      | url: jdbc:mysql://db-publications:3306/${MYSQL_DATABASE}
7      | username: ${MYSQL_USER}
8      | password: ${MYSQL_PASSWORD}
9      | driver-class-name: com.mysql.cj.jdbc.Driver
10
11   jpa:
12     hibernate:
13       | ddl-auto: update
14       | show-sql: true
15
16   server:
17     | port: 8082
18
19 authors:
20   service:
21     | url: http://authors-service:8081
```

Fig.37. Archivo application-docker.yml del microservicio publications-service

## 6.8. Despliegue del frontend con Docker

El frontend fue dockerizado utilizando un Dockerfile que:

- Instala las dependencias con npm install.
- Ejecuta la aplicación con Vite.
- Expone el puerto 5173.

Las URLs de los microservicios son configuradas dinámicamente mediante variables de entorno, permitiendo que el frontend funcione tanto en Docker como en ejecución local.

```

Dockerfile X
frontend > Dockerfile
1 # =====
2 # Frontend - Dockerfile (React + Vite)
3 # =====
4
5 FROM node:20-alpine
6
7 WORKDIR /app
8
9 COPY package*.json ./
10 RUN npm install
11
12 COPY . .
13
14 EXPOSE 5173
15
16 CMD ["npm", "run", "dev", "--", "--host"]
17

```

Fig.38. Archivo Dockerfile de frontend

## 6.9. Ejecución del sistema completo

Una vez configurados todos los archivos necesarios, el sistema completo se levanta ejecutando el siguiente comando desde la raíz del proyecto:

**docker compose up --build**

Este comando construye las imágenes necesarias y levanta todos los servicios de forma coordinada.

Al finalizar, el sistema queda accesible desde el navegador y los endpoints REST están disponibles para consumo.

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	microservices-authors-publications	-	-	-	N/A	2 days ago	
<input type="checkbox"/>	frontend-1	a7b5f98315ea	<a href="#">microservices-au</a>	5173:5173	N/A	2 days ago	
<input type="checkbox"/>	publications-service	679ed9d3bb86	<a href="#">microservices-au</a>	8082:8082	N/A	2 days ago	
<input type="checkbox"/>	authors-service	1bacdd19a4ad	<a href="#">microservices-au</a>	8081:8081	N/A	2 days ago	
<input type="checkbox"/>	db-authors	0831699cf277	<a href="#">postgres:15</a>	-	N/A	2 days ago	
<input type="checkbox"/>	db-publications	ebd54f36ea4a	<a href="#">mysql:8.0</a>	-	N/A	2 days ago	

Fig.39. Contenedores creados en docker.desktop

## 7. Instrucciones de Despliegue y Ejecución

Este apartado describe de forma detallada los pasos necesarios para ejecutar el sistema editorial, tanto utilizando Docker Compose como mediante ejecución local, permitiendo evaluar el proyecto en diferentes entornos.

## 7.1. Requisitos previos

Antes de ejecutar el sistema, se deben cumplir los siguientes requisitos:

### Para despliegue con Docker

- Docker Desktop instalado y en ejecución
- Docker Compose habilitado
- Git instalado

### Para ejecución local

- Java JDK 21
- Node.js
- Npm
- PostgreSQL
- MySQL
- IDE recomendado: IntelliJ IDEA (backend) y Visual Studio Code (frontend)

## 7.2. Despliegue y ejecución con Docker Compose

Este método permite levantar todo el sistema completo (frontend, backend y bases de datos) mediante un único comando.

### Paso 1: Clonar el repositorio

Ubicarse en la carpeta donde desea clonar el proyecto, abrir el terminal de git bash e ingresar el siguiente comando:

```
git clone <url-del-repositorio>
```

### Paso 2: Abrir el proyecto con Visual Studio Code

Ubicarse dentro de la carpeta microservices-authors-publications, abrir un terminal e ingresar el siguiente comando:

```
code .
```

También puede abrir el proyecto completo desde el IDE.

### Paso 3: Crear archivo de variables de entorno (.env)

En la raíz del proyecto, crear el archivo .env usando como referencia el archivo .env.example, definiendo las credenciales necesarias para las bases de datos:

### Paso 4: Configurar variables de entorno del frontend

Ingresar a la carpeta del frontend y crear el archivo .env.docker usando como referencia .env.example:

```
VITE_AUTHORS_API=http://authors-service:8081
```

```
VITE_PUBLICATIONS_API=http://publications-service:8082
```

Este archivo permite que el frontend se comunique correctamente con los microservicios dentro de Docker.

### **Paso 5: Volver a la raíz del proyecto**

### **Paso 6: Levantar todos los servicios con Docker Compose**

Abrir un terminal dentro de Visual Studio Code y ejecutar el siguiente comando:

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

**Nota:** Debe revisar que este comando sea ejecutado en la raíz del proyecto.

Docker se encargará automáticamente de:

- Construir las imágenes
- Levantar las bases de datos
- Ejecutar los microservicios
- Ejecutar el frontend

### **Paso 7: Acceso al sistema**

Una vez levantados los contenedores, el sistema estará disponible en:

**Frontend:** <http://localhost:5173>

Interfaz gráfica del sistema editorial.

**Authors API:** <http://localhost:8081/authors>

**Publications API:** <http://localhost:8082/publications>

### **Paso 8: Detener al sistema**

Para detener todos los servicios:

```
Crtl + c
```

Opcionalmente, para eliminar contenedores y volúmenes ejecutar en el terminal:

```
docker compose down -v
```

### 7.3. Ejecución del sistema en entorno local

Este modo permite ejecutar cada componente de forma independiente sin Docker.

#### 7.3.1. Configuración de bases de datos locales

##### **PostgreSQL (Authors)**

- Crear la base de datos en el motor de base de datos de PostgreSQL, la base de datos debe tener el nombre db-authors:

```
CREATE DATABASE db-authors;
```

##### **MySQL (Publications)**

- Crear la base de datos en el motor de base de datos de MySQL, la base de datos debe tener el nombre db-publications:

```
CREATE DATABASE db-publications;
```

#### 7.3.2. Ejecución de los microservicios backend

No es necesario instalar dependencias manualmente, Maven se encarga automáticamente.

##### **Authors Service**

1. Abrir el proyecto authors-service en IntelliJ IDEA.
2. Verificar y configurar el archivo application.yml con las credenciales a la base de datos.
3. Ejecutar la clase principal: **AuthorsServiceApplication**
4. El servicio se ejecutará en: <http://localhost:8081>
5. Opcional: realizar pruebas en Postman, para ello verifique las URL y los body en el apartado 3.5 del documento.

##### **Publications Service**

1. Abrir el proyecto el proyecto authors-service en IntelliJ IDEA.
2. Verificar y configurar el archivo application.yml con las credenciales a la base de datos.
3. Ejecutar la clase principal: **PublicationsServiceApplication**
4. El servicio se ejecutará en: <http://localhost:8082>
5. Opcional: realizar pruebas en Postman, para ello verifique las URL y los body en el apartado 4.5 del documento.

#### 7.3.3. Ejecución del frontend en local

1. Ingresar a la carpeta del frontend desde el IDE Visual Studio Code.
2. Crear el archivo .env.local usando como referencia .env.example

VITE\_AUTHORS\_API=http://localhost:8081

VITE\_PUBLICATIONS\_API=http://localhost:8082

3. Instalar dependencias para ello abrir el terminal del IDE y ejecutar el siguiente comando:

npm install

4. Ejecutar el frontend ejecutando el siguiente comando

npm run dev

Nota: debe estar ejecutándose los dos microservicios antes de ejecutar el frontend.

5. El frontend quedara disponible en:

<http://localhost:5173>

## 8. Conclusiones

La arquitectura de microservicios implementada permitió una separación clara de responsabilidades entre la gestión de autores y publicaciones, evitando dependencias circulares y garantizando la independencia de datos mediante el uso de bases de datos dedicadas por microservicio. Este enfoque mejora la mantenibilidad, escalabilidad y capacidad de evolución del sistema.

La aplicación de principios SOLID y patrones de diseño como Repository, Factory Method, Strategy y Adapter contribuyó a un diseño desacoplado y extensible, facilitando la incorporación de nuevos requerimientos sin afectar la lógica existente. El uso de DTOs y el manejo centralizado de errores fortalecieron la robustez y consistencia del backend.

El modelado del proceso editorial en BPMN con Camunda permitió validar el flujo de negocio antes de una automatización completa, evidenciando distintos escenarios reales como aprobación, rechazo y retrabajo. La simulación mediante Token Simulation demostró la utilidad del modelado de procesos como herramienta de análisis y validación temprana.

Finalmente, el uso de Docker Compose permitió un despliegue reproducible y controlado de todo el ecosistema, integrando frontend, microservicios y bases de datos en un único entorno. Esto facilita la ejecución, evaluación y futura escalabilidad de la solución, cumpliendo con los objetivos planteados para una arquitectura basada en microservicios.