



ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA



INGENIERÍA EN TECNOLOGÍAS
DE LA INFORMACIÓN

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.

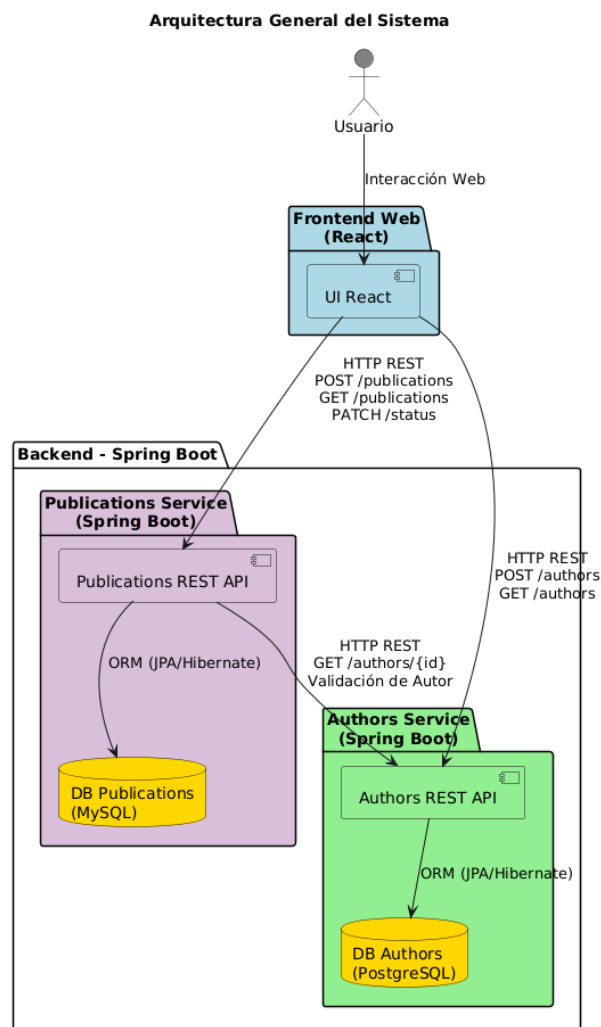


Fig.1. Arquitectura General del Proyecto

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

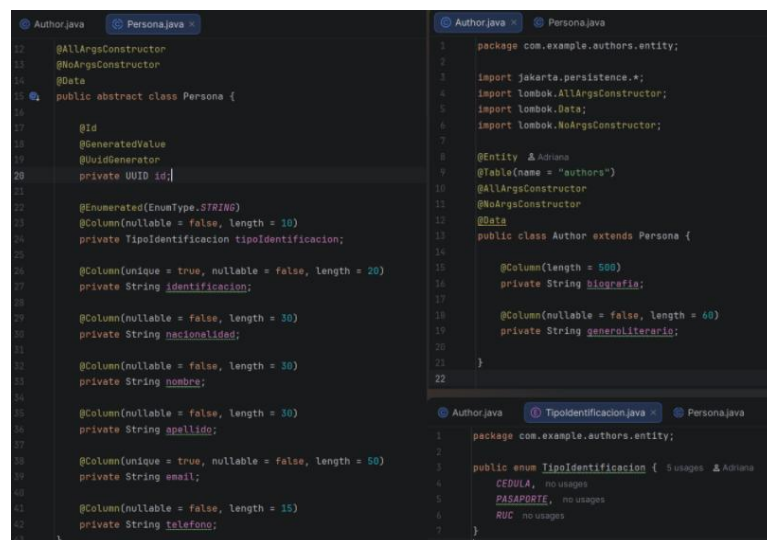
El modelo de dominio del microservicio Authors Service fue diseñado aplicando principios de orientación a objetos y herencia, con el fin de representar de manera clara y extensible la información asociada a los autores.

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.



```

1  @AllArgsConstructor
2  @NoArgsConstructor
3  @Data
4  public abstract class Persona {
5
6      @Id
7      @GeneratedValue
8      @UuidGenerator
9      private UUID id;
10
11      @Enumerated(EnumType.STRING)
12      @Column(nullable = false, length = 10)
13      private TipoIdentificacion tipoIdentificacion;
14
15      @Column(unique = true, nullable = false, length = 20)
16      private String identificacion;
17
18      @Column(nullable = false, length = 30)
19      private String nacionalidad;
20
21      @Column(nullable = false, length = 30)
22      private String nombres;
23
24      @Column(nullable = false, length = 30)
25      private String apellidos;
26
27      @Column(unique = true, nullable = false, length = 50)
28      private String email;
29
30      @Column(nullable = false, length = 15)
31      private String telefono;
32  }
33
34  package com.example.authors.entity;
35
36  import jakarta.persistence.*;
37  import lombok.*;
38
39  @Entity
40  @Table(name = "authors")
41  @AllArgsConstructor
42  @NoArgsConstructor
43  @Data
44  public class Author extends Persona {
45
46      @Column(length = 500)
47      private String biografia;
48
49      @Column(nullable = false, length = 60)
50      private String generoLiterario;
51  }
52
53  package com.example.authors.entity;
54
55  public enum TipoIdentificacion {
56      CEDULA, no usages
57      PASAPORTE, no usages
58      RUC, no usages
59  }

```

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.

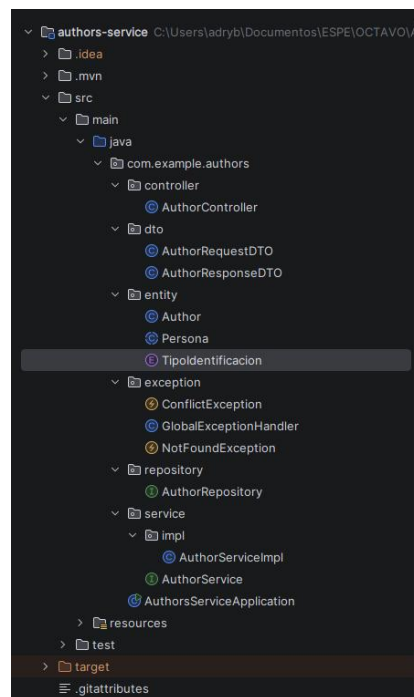


Fig.3. Estructura Microservicio Authors Service

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.

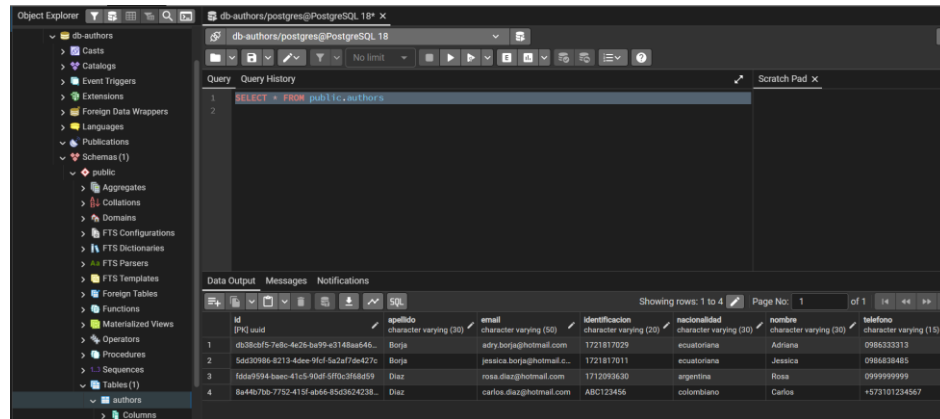


Fig.4. Base de datos PostgreSQL y tabla authors

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:

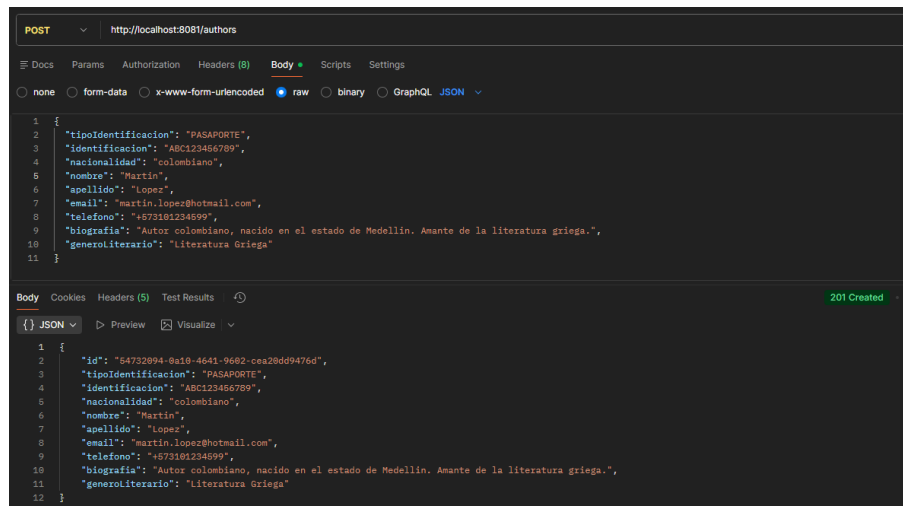


Fig.5. Creación de autor desde POSTMAN

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

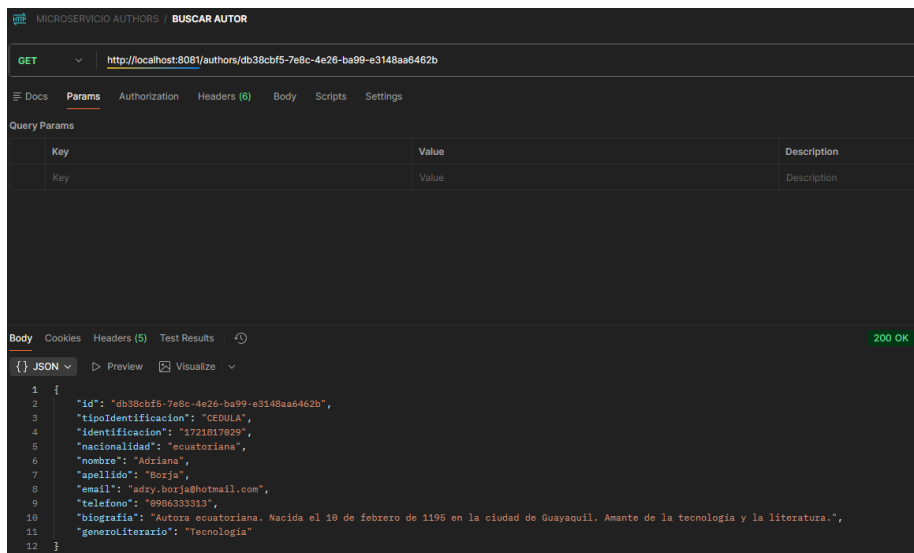


Fig.6. Método GET por ID en POSTMAN

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

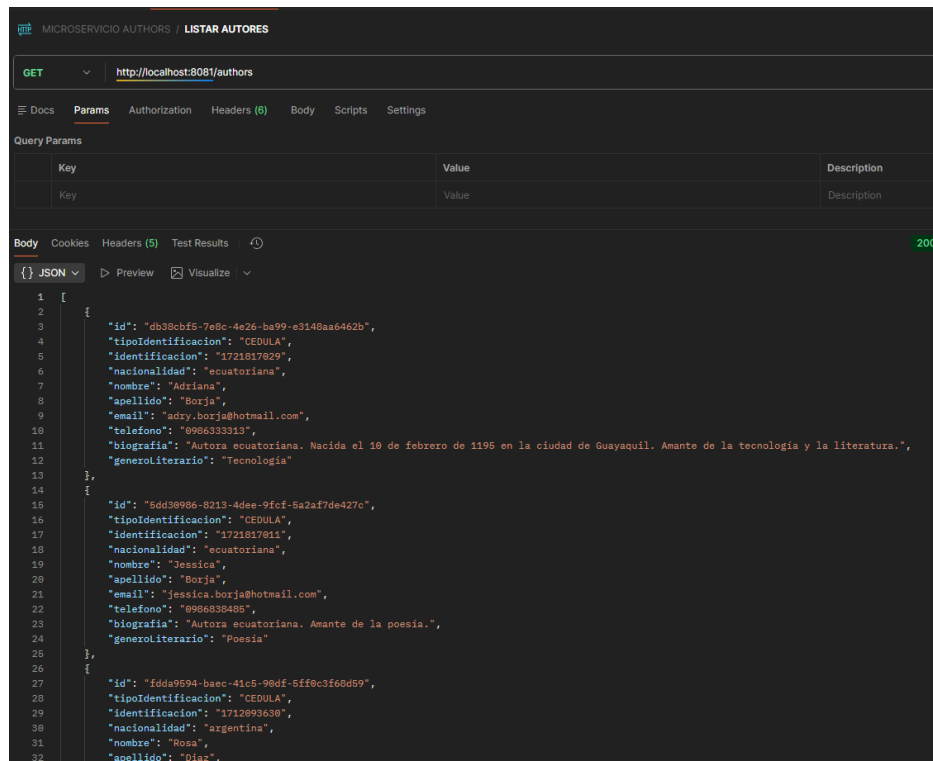


Fig.7. Listar todos los autores en POSTMAN

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.

```

AuthorsServiceApplication.java  AuthorRepository.java x
1 package com.example.authors.repository;
2
3
4 import com.example.authors.entity.Author;
5 import org.springframework.data.jpa.repository.JpaRepository;
6
7 import java.util.UUID;
8
9 public interface AuthorRepository extends JpaRepository<Author, UUID> { 2 usages  ⚠ Adriana
10     boolean existsByEmail(String email); 1 usage  ⚠ Adriana
11     boolean existsByIdentificacion(String identificacion); 1 usage  ⚠ Adriana
12
13 }
  
```

Fig.8. Interface AuthorRepository aplicando Repository Pattern

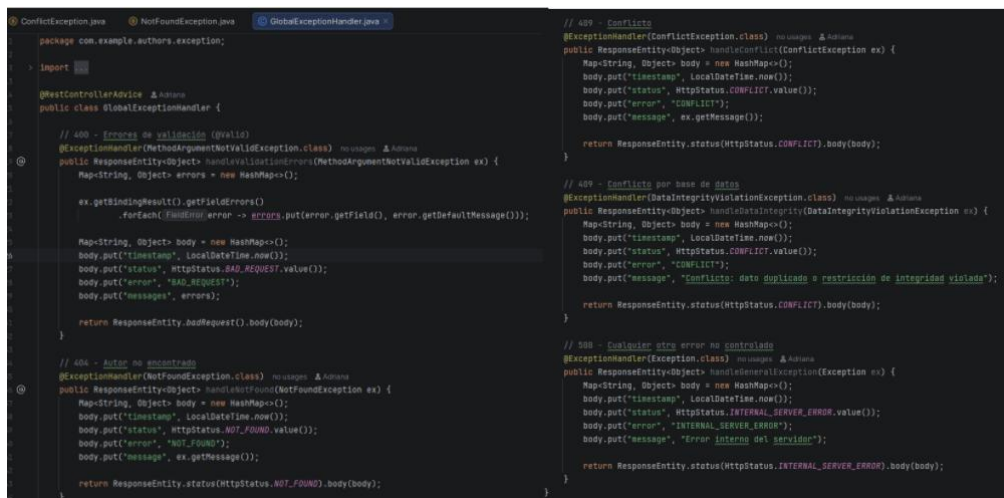
2.7. Manejo de Errores y Validaciones

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.



```

package com.example.authors.exception;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.servlet.mvc.annotation.annotation.AnnotationMethodArgumentResolver;

// 400 - Errores de validación (@Valid)
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<Object> handleValidationErrors(MethodArgumentNotValidException ex) {
    Map<String, Object> errors = new HashMap<>();
    ex.getBindingResult().getFieldErrors().forEach(error -> errors.put(error.getField(), error.getDefaultMessage()));

    Map<String, Object> body = new HashMap<>();
    body.put("timestamp", LocalDateTime.now());
    body.put("status", HttpStatus.BAD_REQUEST.value());
    body.put("error", "BAD_REQUEST");
    body.put("message", errors);

    return ResponseEntity.badRequest().body(body);
}

// 404 - Autor no encontrado
@ExceptionHandler(NotFoundException.class)
public ResponseEntity<Object> handleNotFoundException(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);
}

// 409 - Conflictos
@ExceptionHandler(ConflictException.class)
public ResponseEntity<Object> 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 - Conflictos por base de datos
@ExceptionHandler(DataIntegrityViolationException.class)
public ResponseEntity<Object> 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", "Conflicto: data duplicada o restricción de integridad violada");

    return ResponseEntity.status(HttpStatus.CONFLICT).body(body);
}

// 500 - Cualquier otro error no controlado
@ExceptionHandler(Exception.class)
public ResponseEntity<Object> handleGeneralException(Exception ex) {
    Map<String, Object> body = new HashMap<>();
    body.put("timestamp", LocalDateTime.now());
    body.put("status", HttpStatus.INTERNAL_SERVER_ERROR.value());
    body.put("error", "INTERNAL_SERVER_ERROR");
    body.put("message", "Error interno del servidor");

    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body);
}

```

Fig.9. GlobalExceptionHandler manejo consistente de errores



```

package com.example.authors.exception;

public class ConflictException extends RuntimeException {
    public ConflictException(String message) { super(message); }
}

package com.example.authors.exception;

public class NotFoundException extends RuntimeException {
    public NotFoundException(String message) { super(message); }
}

```

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.

```
1 package com.example.publications.entity;
2
3 import jakarta.persistence.*;
4 import lombok.*;
5 import org.hibernate.annotations.*;
6
7 import java.time.LocalDateTime;
8 import java.util.UUID;
9
10 @MappedSuperclass
11 @Entity
12 @Table(name = "publications")
13 @AllArgsConstructor
14 @NoArgsConstructor
15 @Data
16 public abstract class Publication {
17
18     @Id
19     @GeneratedValue
20     @UUIDGenerator
21     private UUID id;
22
23     @Column(nullable = false, length = 150)
24     private String title;
25
26     @Column(name = "author_id", nullable = false)
27     private UUID authorId;
28
29     @Enumerated(EnumType.STRING)
30     @Column(nullable = false, length = 10)
31     private EditorialStatus status;
32
33     @Column(name = "created_at", nullable = false)
34     private LocalDateTime createdAt;
35
36     @Column(name = "updated_at")
37     private LocalDateTime updatedAt;
38
39     @Column(name = "published_at")
40     private LocalDateTime publishedAt;
41
42     @PrePersist
43     protected void onCreate() {
44         this.createdAt = LocalDateTime.now();
45         this.status = EditorialStatus.DRAFT;
46     }
47
48     @PreUpdate
49     protected void onUpdate() {
50         this.updatedAt = LocalDateTime.now();
51     }
52 }
```

```
1 package com.example.publications.entity;
2
3 import jakarta.persistence.*;
4
5 @Entity
6 @Table(name = "publications")
7 @AllArgsConstructor
8 @NoArgsConstructor
9 @Data
10 public class PublicationContent extends Publication {
11
12     @Column(nullable = false, length = 150)
13     private String summary;
14
15     @Column(columnDefinition = "TEXT", nullable = false)
16     private String content;
17
18     @Enumerated(EnumType.STRING)
19     @Column(name = "tipo_publicacion", nullable = false, length = 15)
20     private TipoPublicacion tipoPublicacion;
21
22     @Column(nullable = false, length = 50)
23     private String category;
24 }
```

```
1 package com.example.publications.entity;
2
3 public enum EditorialStatus {
4     DRAFT, 2 usages
5     IN_REVIEW, 2 usages
6     APPROVED, 2 usages
7     PUBLISHED, 2 usages
8     REJECTED, 2 usages
9 }
10
```

```
1 package com.example.publications.entity;
2
3 public enum TipoPublicacion { 7 usages
4
5     ARTICULO, no usages
6     LIBRO, no usages
7     INVESTIGACION, no usages
8     INFORME, no usages
9     OTRO, no usages
10 }
```

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.

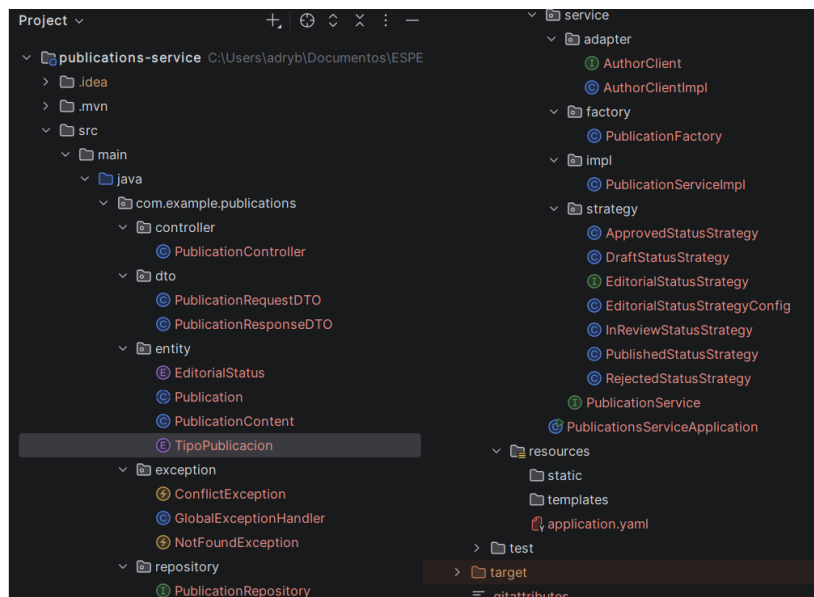


Fig.12. Estructura Microservicio Publications Service

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.

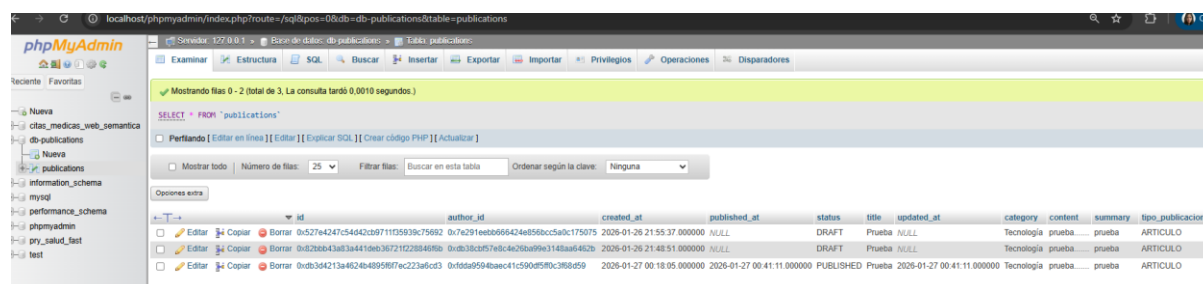


Fig.13. Base de datos MYSQL y tabla publications

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:

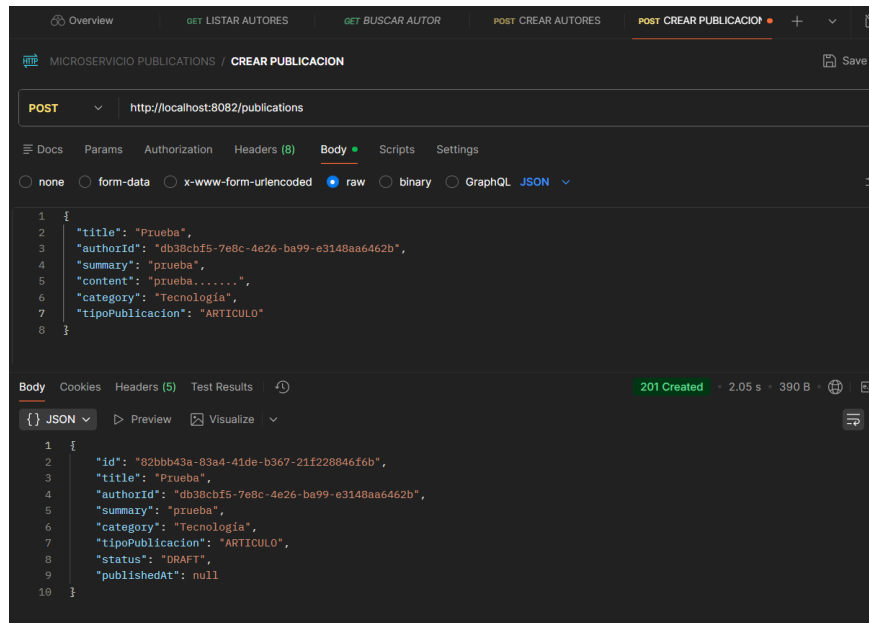


Fig.14. Creación de publicación desde POSTMAN

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

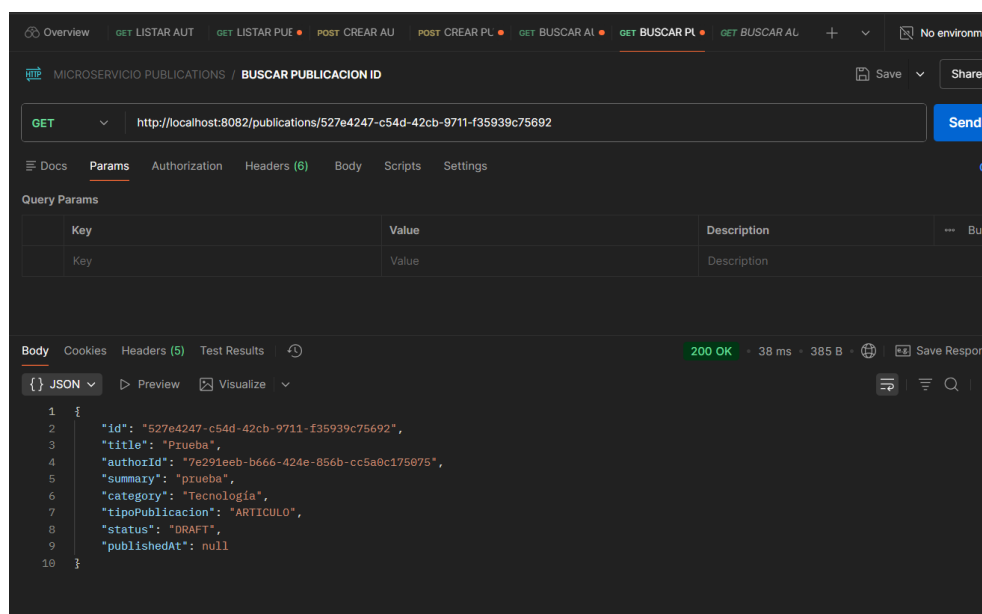


Fig.15. Método GET para obtener una publicación por id en POSTMAN

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

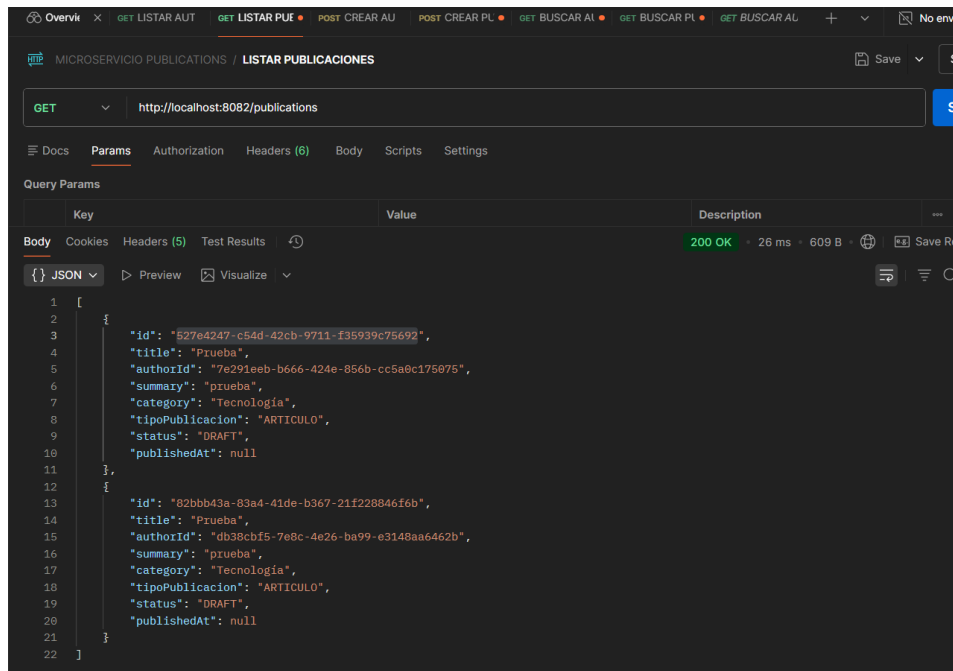


Fig.16. Listar todas las publicaciones en POSTMAN

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

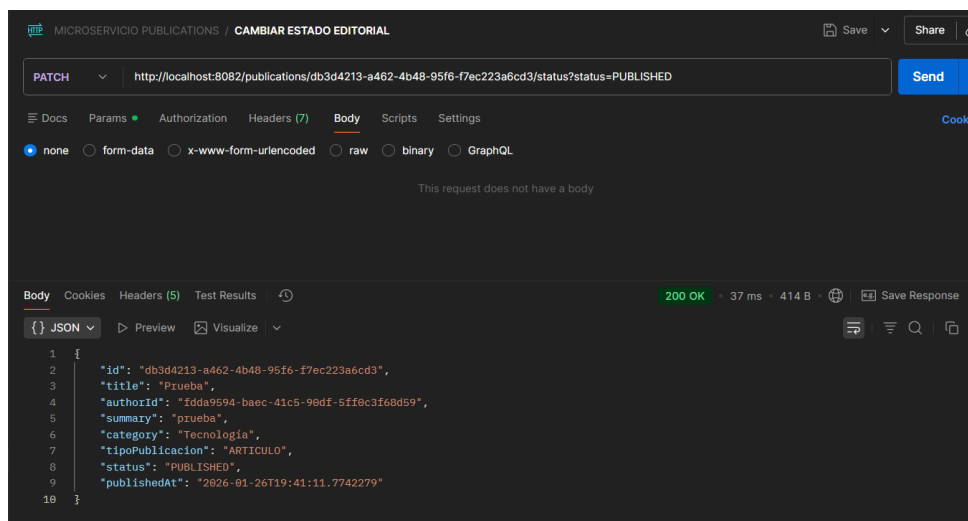


Fig.17. Cambio de status editorial en POSTMAN

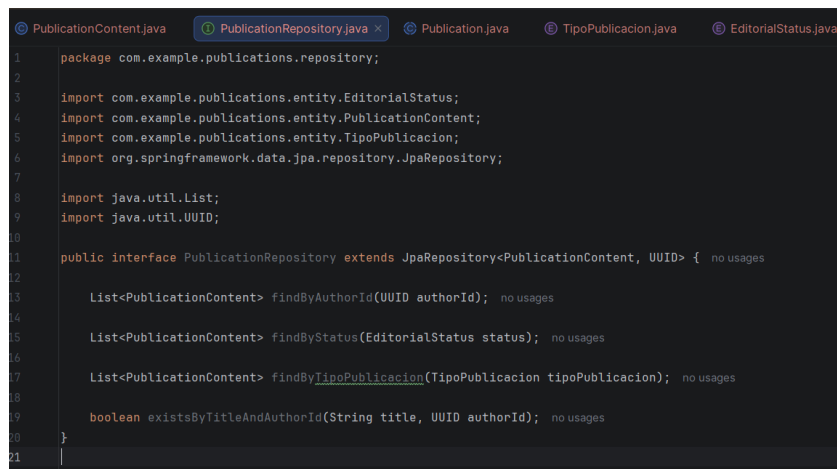
3.6. Patrones de Diseños Aplicados

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.



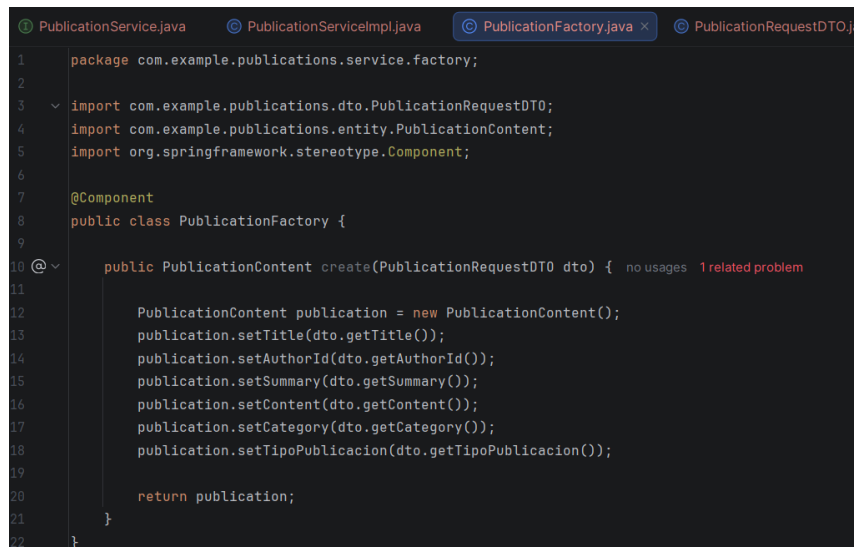
```
1 package com.example.publications.repository;
2
3 import com.example.publications.entity.EditorialStatus;
4 import com.example.publications.entity.PublicationContent;
5 import com.example.publications.entity.TipoPublicacion;
6 import org.springframework.data.jpa.repository.JpaRepository;
7
8 import 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     public PublicationContent create(PublicationRequestDTO dto) {
11
12         PublicationContent publication = new PublicationContent();
13         publication.setTitle(dto.getTitle());
14         publication.setAuthorId(dto.getAuthorId());
15         publication.setSummary(dto.getSummary());
16         publication.setContent(dto.getContent());
17         publication.setCategory(dto.getCategory());
18         publication.setTipoPublicacion(dto.getTipoPublicacion());
19
20         return publication;
21     }
22 }

```

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.

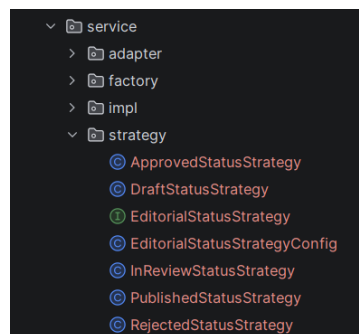
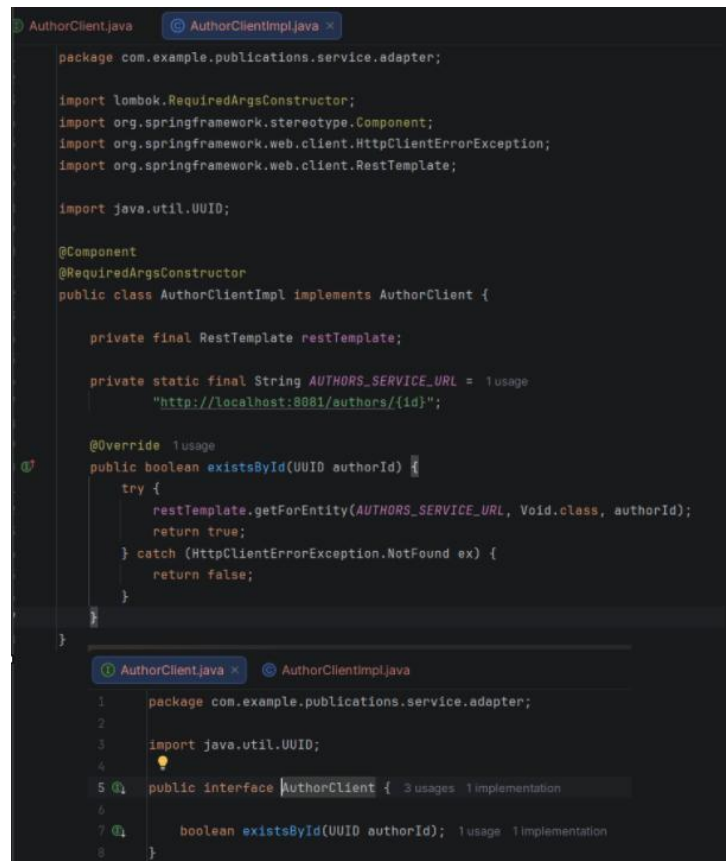


Fig.20. Paquete strategy para aplicar el Patrón Strategy

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 = 1usage
        "http://localhost:8081/authors/{id}";

    @Override 1usage
    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 { 3 usages 1 implementation

    boolean existsById(UUID authorId); 1usage 1 implementation
}
```

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

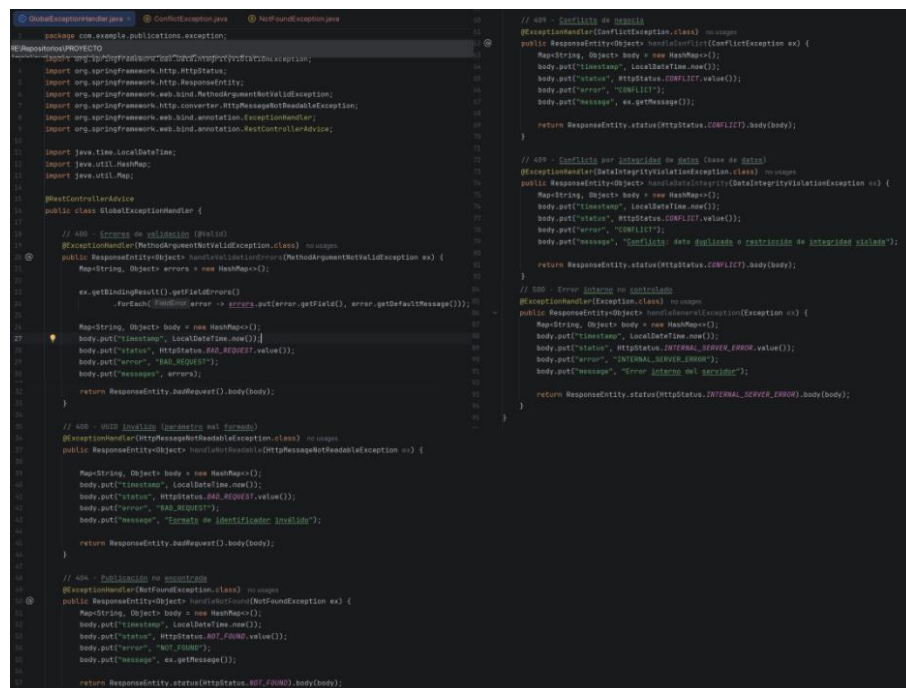
El microservicio Publications Service implementa un manejo centralizado de errores y validaciones mediante la clase `GlobalExceptionHandler`, anotada con `@RestControllerAdvice`, con el objetivo de garantizar respuestas HTTP consistentes, controladas y alineadas a las reglas del dominio.

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.



```

package com.example.publications.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

import java.time.LocalDateTime;
import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    // 404 - Resource not found (Not found)
    @ExceptionHandler({NotFoundException.class})
    public ResponseEntity<Object> handleNotFoundException(NotFoundException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("status", HttpStatus.NOT_FOUND.value());
        body.put("error", "404 Not Found");
        body.put("message", ex.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
    }

    // 409 - Conflict (Duplicate or invalid transition)
    @ExceptionHandler({ConflictException.class})
    public ResponseEntity<Object> handleConflictException(ConflictException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("status", HttpStatus.CONFLICT.value());
        body.put("error", "409 Conflict");
        body.put("message", ex.getMessage());
        return ResponseEntity.status(HttpStatus.CONFLICT).body(body);
    }

    // 409 - Conflict (Data integrity violation)
    @ExceptionHandler({DataIntegrityViolationException.class})
    public ResponseEntity<Object> handleDataIntegrityViolationException(DataIntegrityViolationException ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("status", HttpStatus.CONFLICT.value());
        body.put("error", "409 Conflict");
        body.put("message", "Conflicto de integridad de datos");
        return ResponseEntity.status(HttpStatus.CONFLICT).body(body);
    }

    // 500 - Error internal (unhandled)
    @ExceptionHandler({Exception.class})
    public ResponseEntity<Object> handleInternalException(Exception ex) {
        Map<String, Object> body = new HashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("status", HttpStatus.INTERNAL_SERVER_ERROR.value());
        body.put("error", "500 Internal Server Error");
        body.put("message", "Error interno no controlado");
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body);
    }
}

```

Fig.22. GlobalExceptionHandler manejo consistente de errores



```

package com.example.publications.exception;

public class ConflictException extends RuntimeException { 5 usages
    public ConflictException(String message) { super(message); }
}

package com.example.publications.exception;

public class NotFoundException extends RuntimeException{ 6 usages
    public NotFoundException(String message) { super(message); } 3 usages
}

```

Fig.23. Manejo consistente de errores

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.

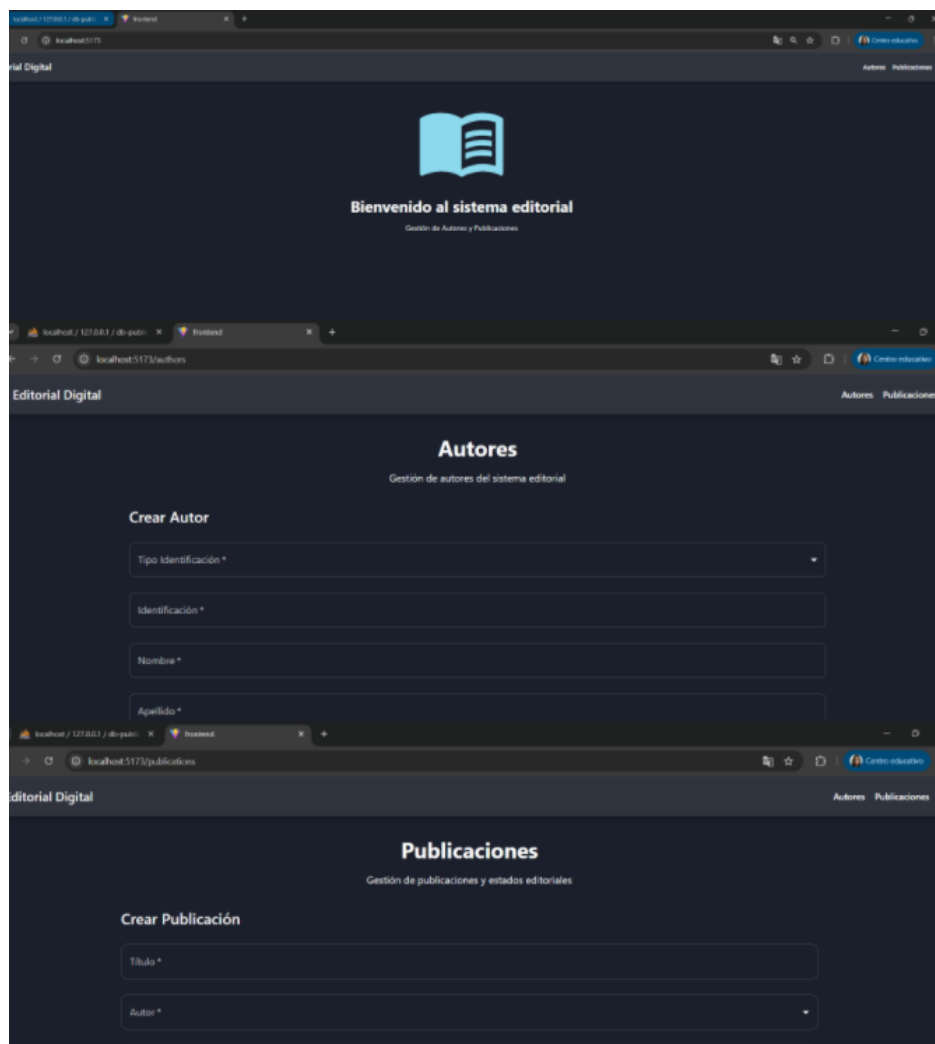


Fig.24. Páginas frontend Editorial Digital

4.1. Funcionalidades implementadas

El frontend implementa las siguientes funcionalidades:

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

Autores

Gestión de autores del sistema editorial

Crear Autor

Tipo Identificación *

Identificación *

Nombre *

Apellido *

Email *

Teléfono *

Nacionalidad *

Biografía

Género Literario *

Guardar Autor

Lista de Autores

Buscar autor por ID (UUID)

Buscar

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

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 *

Guardar Publicación

Lista de Publicaciones

Buscar publicación por ID (UUID)

Buscar

Título	Autor	Tipo	Categoría	Resumen	Estado	Fecha publicación	Acciones
Prueba	Ricardo Lopez	ARTICULO	Tecnología	prueba	REJECTED	-	<div>Ver contenido</div> <div>Cambiar estado</div>
Prueba	Adriana Borja	ARTICULO	Tecnología	prueba	PUBLISHED	28/1/2026	<div>Ver contenido</div> <div>Cambiar estado</div>
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	-	<div>Ver contenido</div> <div>Cambiar estado</div>

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.

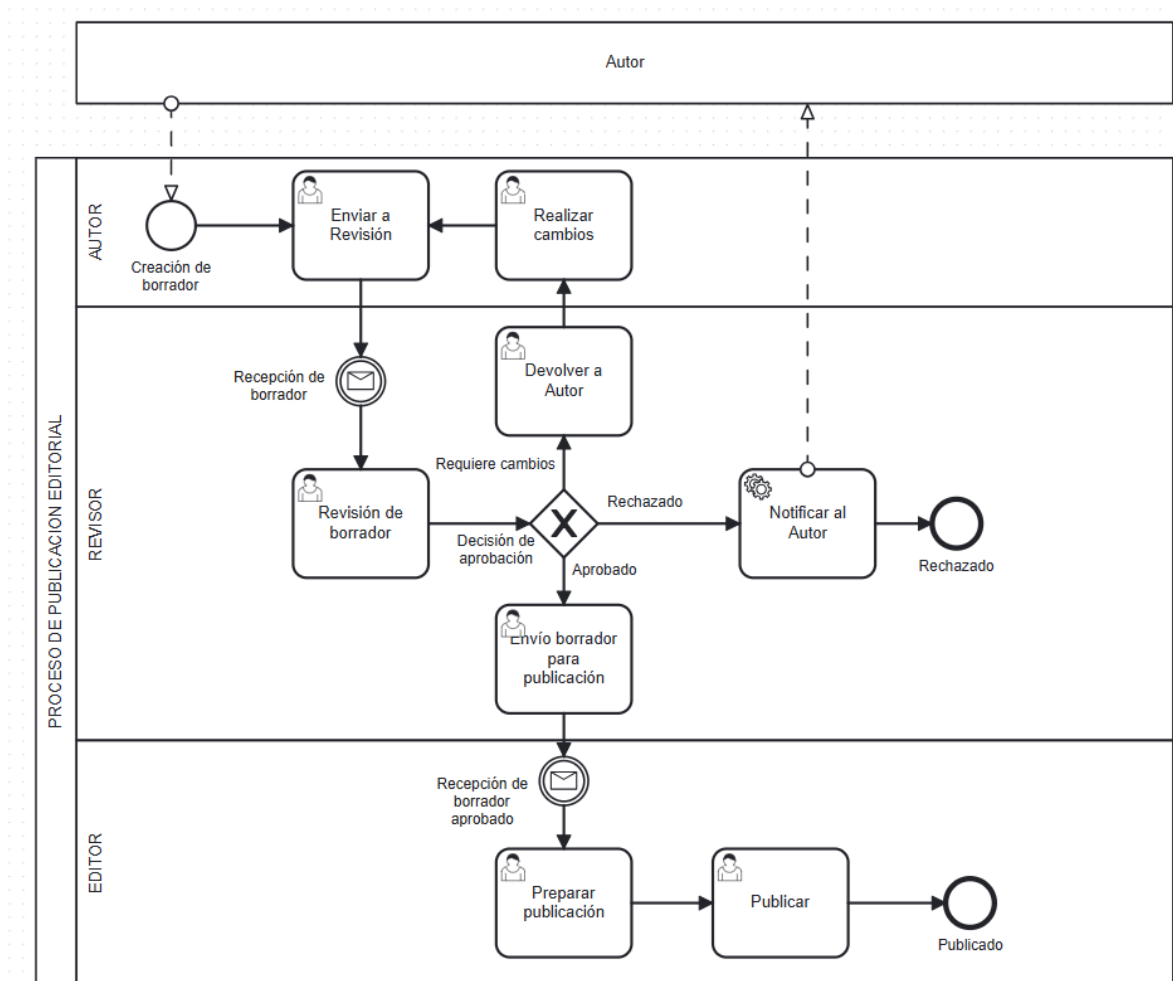


Fig.27. Modelado BPM de proceso de publicación editorial digital

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.

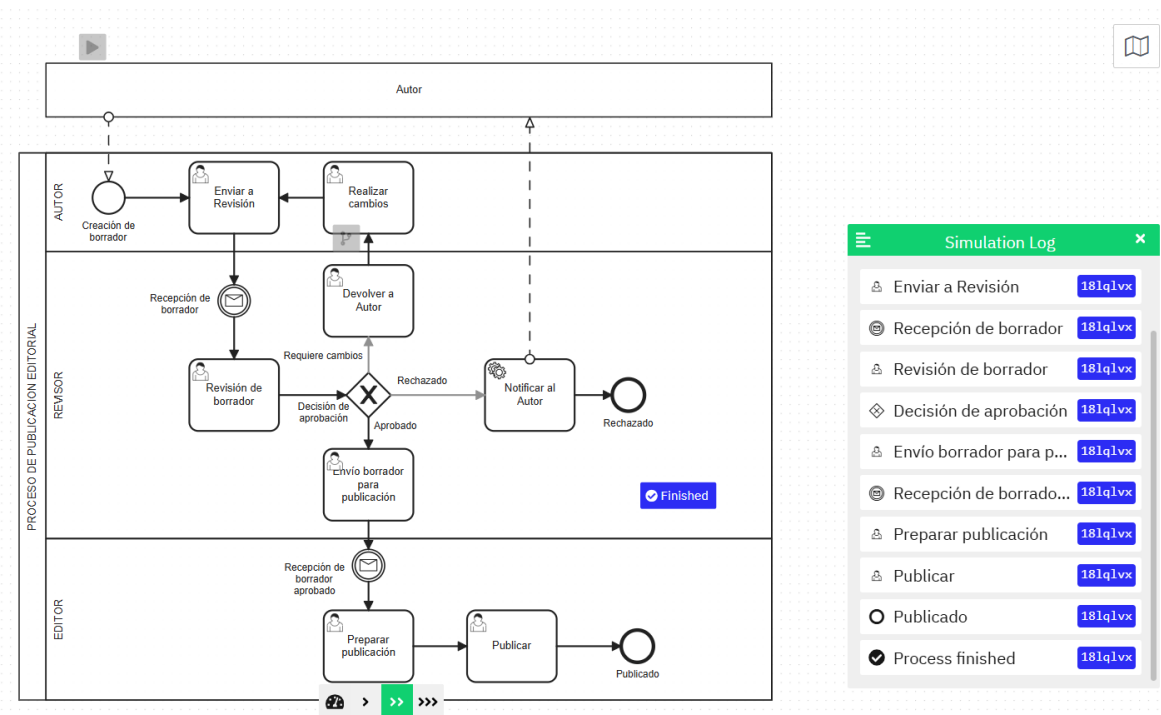


Fig.28. Modelado BPM escenario de aprobación

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.

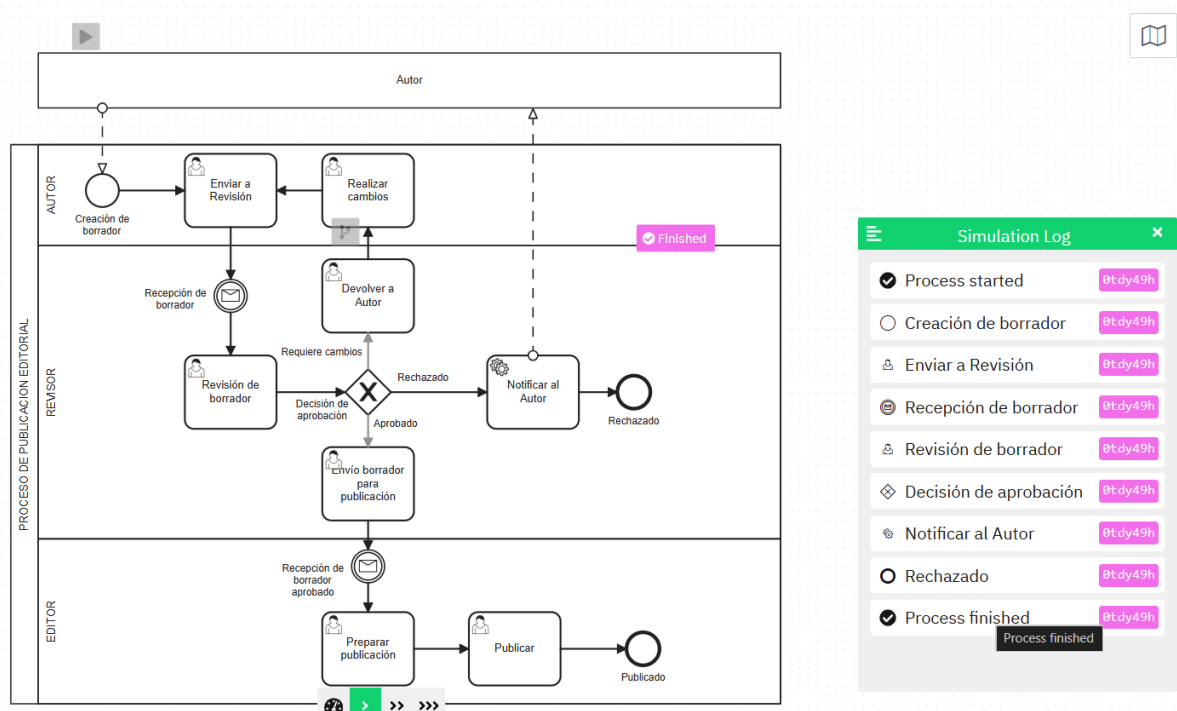


Fig.29. Modelado BPM escenario de rechazo

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.

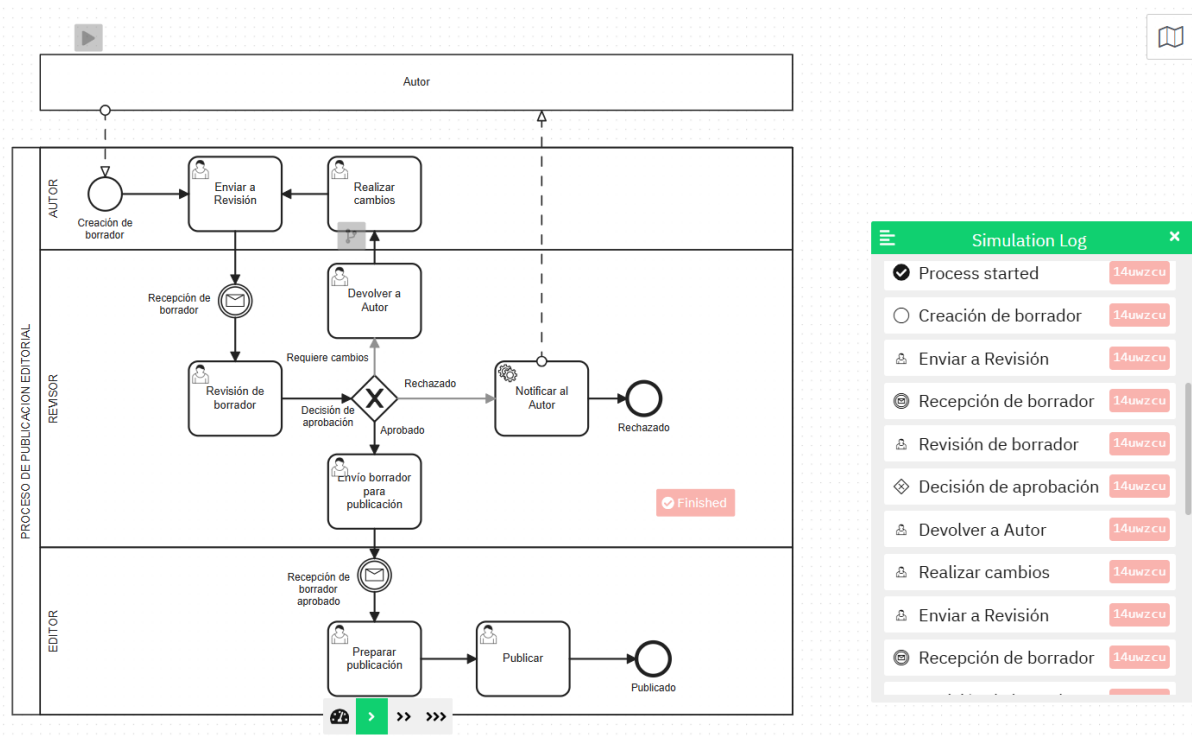


Fig.30. Modelado BPM escenario de requiere cambios y retrabajo

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
1 version: "3.9"
2
3 networks:
4   editorial-net:
5     driver: bridge
6
7 volumes:
8   authors-db-data:
9   publications-db-data:
10
11 services:
12
13   # =====
14   # DATABASE AUTHORS (PostgreSQL)
15   # =====
16   db-authors:
17     image: postgres:15
18     container_name: db-authors
19     env_file:
20       - .env
21     environment:
22       POSTGRES_DB: ${POSTGRES_DB}
23       POSTGRES_USER: ${POSTGRES_USER}
24       POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
25     volumes:
26       - authors-db-data:/var/lib/postgresql/data
27     networks:
28       - editorial-net
29     healthcheck:
30       test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER}"]
31       interval: 10s
32       timeout: 5s
33       retries: 5
34
35   # =====
36   # DATABASE PUBLICATIONS (MySQL)
37   # =====
38   db-publications:
39     image: mysql:8.0
40     container_name: db-publications
41     env_file:
42       - .env
43     environment:
44       MYSQL_DATABASE: ${MYSQL_DATABASE}
45       MYSQL_USER: ${MYSQL_USER}
46       MYSQL_PASSWORD: ${MYSQL_PASSWORD}
47       MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
48     volumes:
49       - publications-db-data:/var/lib/mysql
50     networks:
51       - editorial-net
52     healthcheck:
53       test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
54       interval: 10s
55       timeout: 5s
56       retries: 5
57
58   # =====
59   # AUTHORS SERVICE
60   # =====
61   authors-service:
62     build: ./authors-service
63     container_name: authors-service
64     env_file:
65       - .env
66     depends_on:
67       db-authors:
68         condition: service_healthy
69     ports:
70       - "8081:8081"
71     networks:
72       - editorial-net
73     environment:
74       SPRING_PROFILES_ACTIVE: docker
75
76   # =====
77   # PUBLICATIONS SERVICE
78   # =====
79   publications-service:
80     build: ./publications-service
81     container_name: publications-service
82     env_file:
83       - .env
84     depends_on:
85       db-publications:
86         condition: service_healthy
87       authors-service:
88         condition: service_started
89     ports:
90       - "8082:8082"
91     networks:
92       - editorial-net
93     environment:
94       SPRING_PROFILES_ACTIVE: docker
95
96   # =====
97   # FRONTEND
98   # =====
99   frontend:
100     build: ./frontend
101     ports:
102       - "5173:5173"
103     env_file:
104       - ./frontend/.env.docker
105     depends_on:
106       - authors-service
107       - publications-service
108     networks:
109       - editorial-net
110
```

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
```

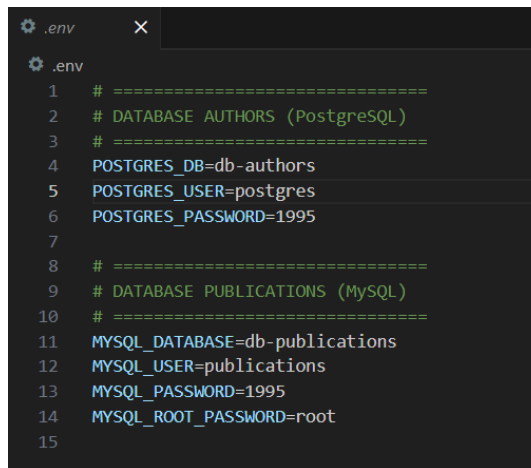
Fig.31. Archivo docker-compose.yml configuración de bases de datos

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).



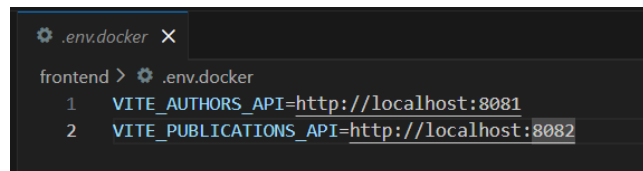
```

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.



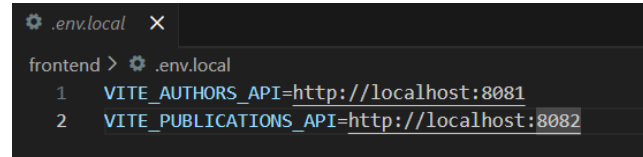
```

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.



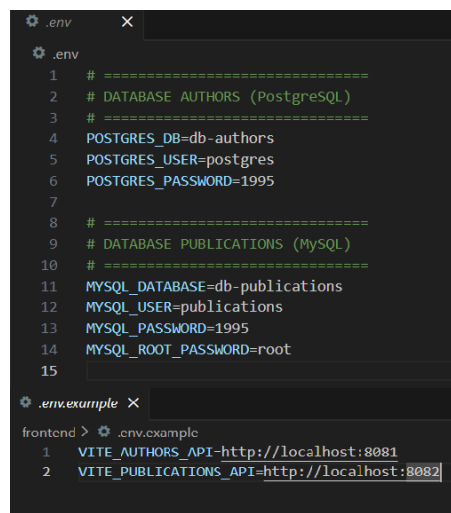
```

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
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
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.

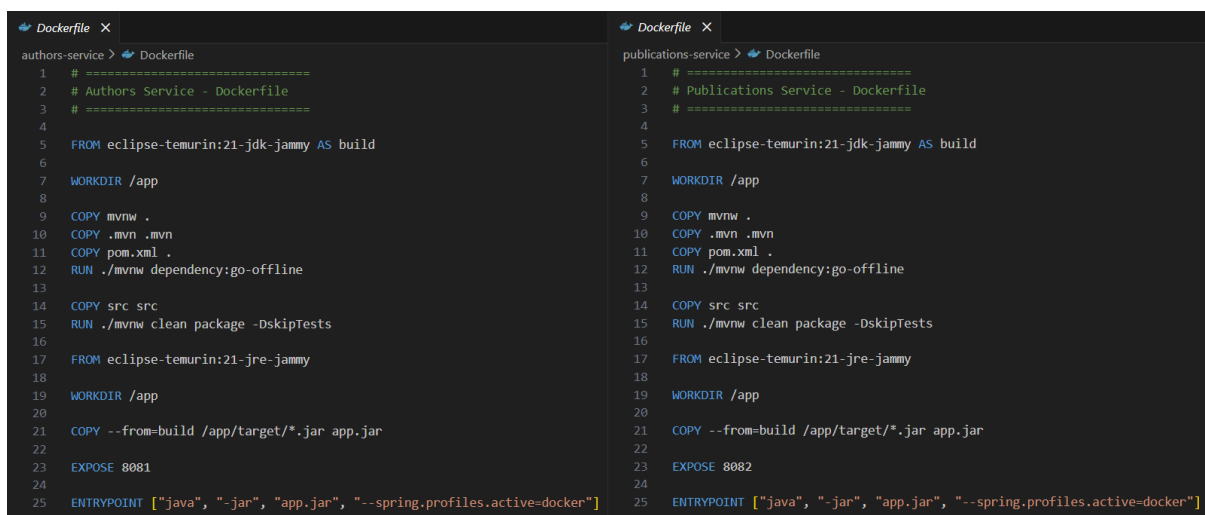


Fig.36. Archivos Dockerfile de los microservicios

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.

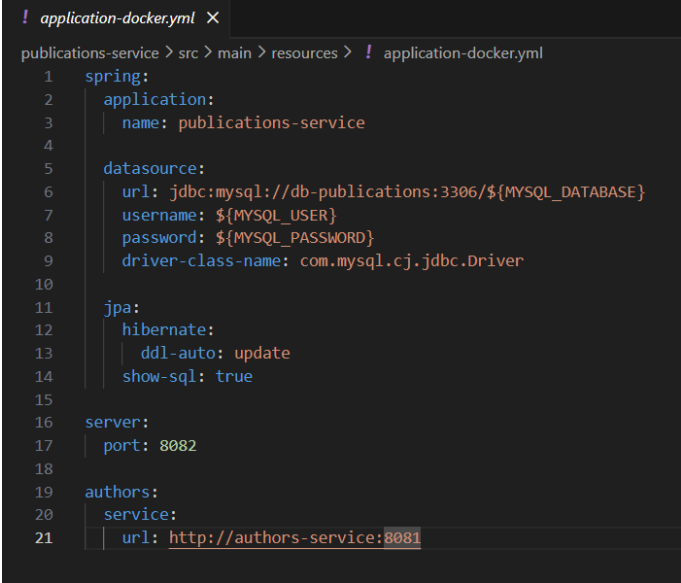
6.7. Comunicación entre microservicios en Docker

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
```

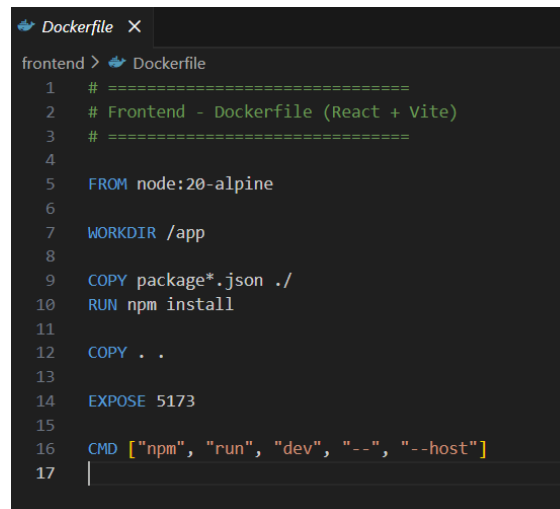
Fig.37. Archivo application-docker.yml del microservicio publications-service

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
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

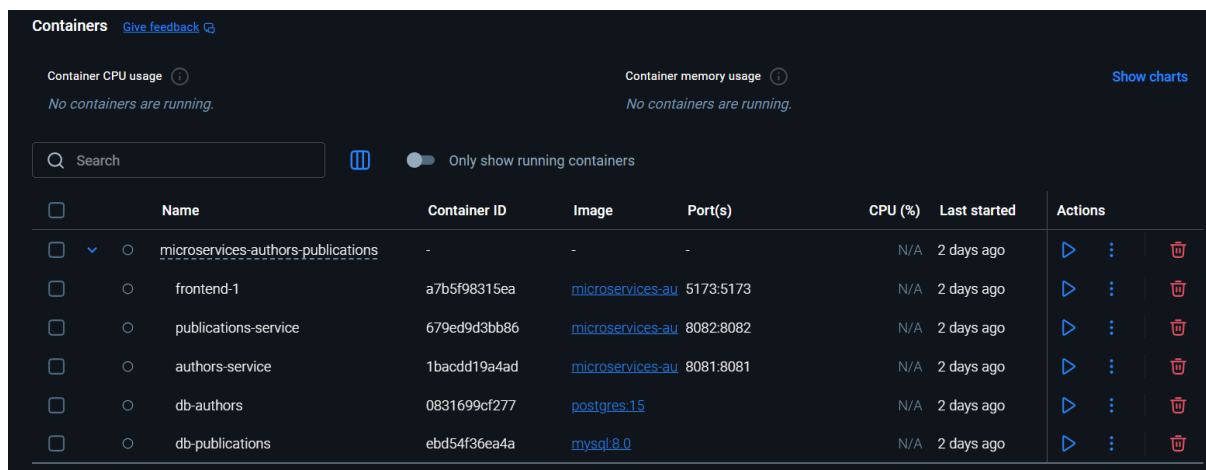
6.9. Ejecución del sistema completo

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.



	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	microservices-au	5173:5173	N/A	2 days ago	
<input type="checkbox"/>	publications-service	679ed9d3bb86	microservices-au	8082:8082	N/A	2 days ago	
<input type="checkbox"/>	authors-service	1bacdd19a4ad	microservices-au	8081:8081	N/A	2 days ago	
<input type="checkbox"/>	db-authors	0831699cf277	postgres:15		N/A	2 days ago	
<input type="checkbox"/>	db-publications	ebd54f36ea4a	mysql:8.0		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

Ingresa a la carpeta del frontend y crea 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

Abre un terminal dentro de Visual Studio Code y ejecuta 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:

```
Ctrl + 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.