Universidad de las Fuerzas Armadas ESPE
Unidad de Educación a Distancia

Nombres y apellidos: Oscar Domínguez

Asignatura: Arquitectura de software

Link del repositorio:

https://github.com/OscarAdrian2001/microservicios-editorial

ACTIVIDAD 3

Introducción:

S realizo un sistema editorial basado en una arquitectura de microservicios, cuyo objetivo es gestionar autores y publicaciones de manera independiente pero integrada. La solución está compuesta por dos microservicios principales: Authors Service y Publications Service, los cuales interactúan entre sí mediante comunicación HTTP.

Objetivos

Diseñar e implementar un sistema editorial utilizando una arquitectura basada en microservicios, integrando backend, frontend, contenedores Docker y modelado de procesos BPMN, con el fin de aplicar los conceptos fundamentales de Arquitectura de Software.

Objetivos Específicos

- Implementar microservicios para la gestión de autores y publicaciones, utilizando Node.js, Express y TypeScript.
- Establecer la comunicación entre microservicios mediante llamadas HTTP.
- Utilizar Docker y Docker Compose para la orquestación y despliegue de los servicios del sistema.
- Desarrollar un frontend web que permita consumir los microservicios y realizar operaciones básicas de gestión.
- Modelar el proceso editorial mediante BPMN 2.0, representando los flujos de aprobación y rechazo de publicaciones.
- Realizar la simulación del proceso utilizando Token Simulation para validar el comportamiento del modelo BPMN.
- Documentar la arquitectura, las pruebas realizadas y los resultados obtenidos durante el desarrollo del proyecto.

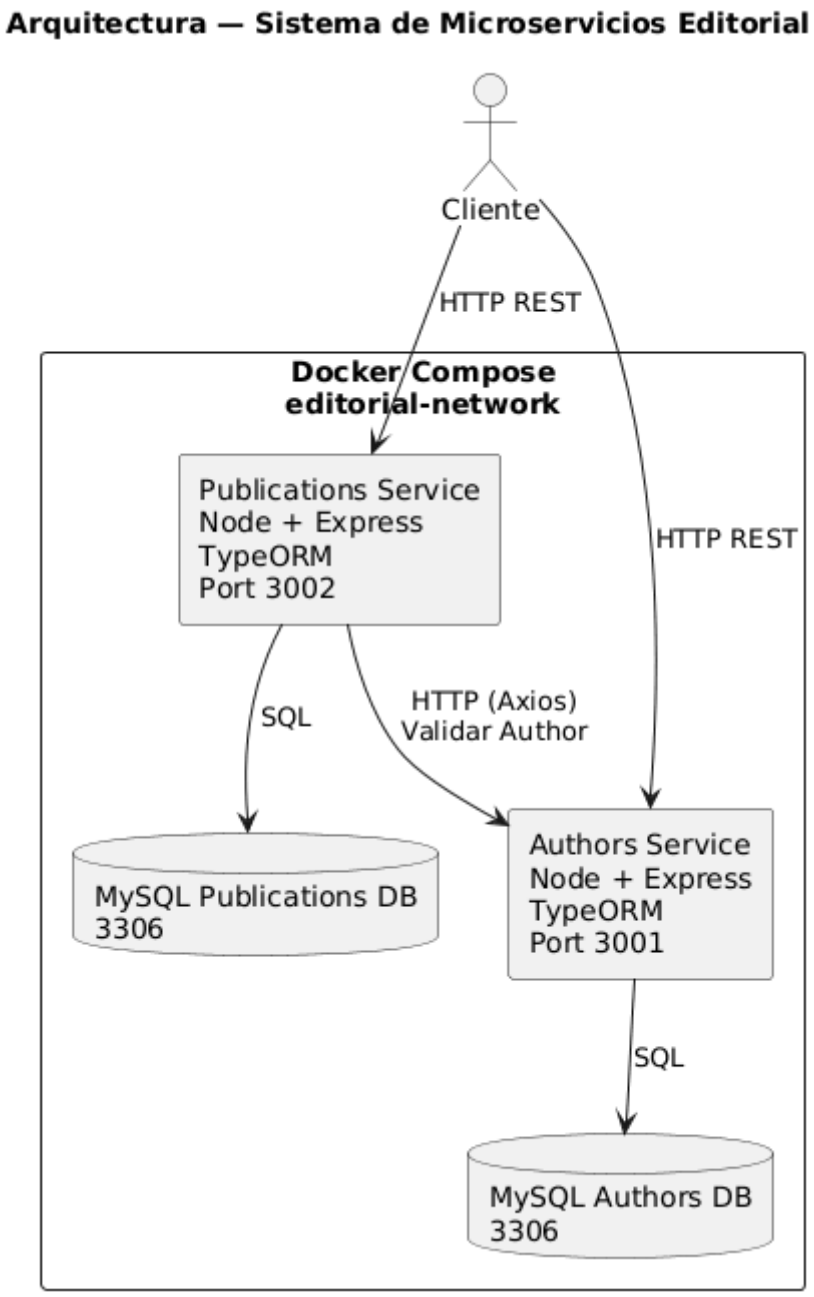
Descripción general

Se implementó una arquitectura de microservicios compuesta por:

- Authors Service
- Publications Service
- Bases de datos independientes
- Comunicación HTTP REST entre servicios
- Contenerización con Docker Compose

Cada microservicio tiene su propia base de datos y se comunica mediante API REST.

Diagrama de Arquitectura



El sistema se compone de dos microservicios: Authors Service y Publications Service, Cada microservicio cuenta con su propia lógica de negocio y su propio BD, lo que facilita la comunicación entre los microservicios donde las peticiones se realizan a través de HTTP, permitiendo que el microservicio de publicaciones valide la existencia de un autor antes de registrar una publicación.

Por su parte, el acceso al sistema se realiza a través de un frontend, el cual consume los microservicios para realizar operaciones como crear y listar autores y publicaciones entre otros. Por tanto, la ejecución y el despliegue del sistema, se utiliza Docker Compose, que permite levantar todos los servicios y bases de datos de forma conjunta y controlada.

Backend

Authors Service

Creamos el proyecto en nuestra carpeta principal

```
PS C:\espe1\workspace\microservicios-editorial> mkdir authors-service
PS C:\espe1\workspace\microservicios-editorial> cd authors-service
PS C:\espe1\workspace\microservicios-editorial\authors-service> npm init -y
Wrote to C:\espe1\workspace\microservicios-editorial\authors-service\package.json:
```

Instalamos dependencias necesarias

```
PS C:\espe1\workspace\microservicios-editorial\authors-service> npm install express typeorm mysql2 reflect-metadata axios
```

```
found 0 vulnerabilities
PS C:\espe1\workspace\microservicios-editorial\authors-service> npm install -D typescript ts-node-dev @types/node @types/express
npm warn deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do not use it. Check out lru-cache if you want a goo
```

tsconfig.json

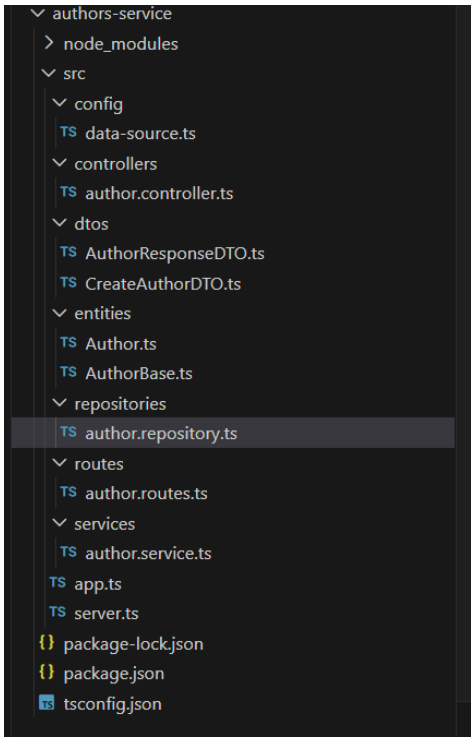
```
authors-service > tsconfig.json > ...
```

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "outDir": "dist",
    "rootDir": "src",
    "strict": true,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  }
}
```

Responsabilidad

Gestiona la información de autores.

Estructura interna del microservicio



Arquitectura aplicada

Se implementó arquitectura por capas:

- Controller:maneja HTTP

```
TS author.controller.ts ×  TS author.service.ts  TS AuthorBase.ts
authors-service > src > controllers > TS author.controller.ts > ...
export class AuthorController {
  // Crear autor
  async create(req: Request, res: Response) {
    try {
      const author = await authorService.createAuthor(req.body);
      return res.status(201).json(author);
    } catch (error: any) {
      return res.status(400).json({
        message: error.message || "Error creating author"
      });
    }
  }

  // Obtener autor por ID
  async getById(req: Request, res: Response) {
    const id = Number(req.params.id);

    if (isNaN(id)) {
      return res.status(400).json({
        message: "Invalid ID"
      });
    }

    try {
      const author = await authorService.getAuthorById(id);

      if (!author) {
        return res.status(404).json({
          message: "Author not found"
        });
      }
```

- Service:lógica de negocio

```
TS author.service.ts ×  TS author.controller.ts
authors-service > src > services > TS author.service.ts > ...
export class AuthorService {
  async createAuthor(data: CreateAuthorDTO): Promise<Author> {
    }

    async getAuthorById(id: number): Promise<Author | null> {
      return await AuthorRepository.findOneBy({ id });
    }

    async getAllAuthors(): Promise<Author[]> {
      return await AuthorRepository.find();
    }
  }
```

- Repository:acceso a datos

TS author.repository.ts

authors-service > src > repositories > TS author.repository.ts > ...

```ts
import { AppDataSource } from "../config/data-source";
import { Author } from "../entities/Author";

export const AuthorRepository = AppDataSource.getRepository(Author);

```

- Entity:modelo ORM

TS Author.ts

authors-service > src > entities > TS Author.ts > ...

```ts
import { Entity, Column } from "typeorm";
import { AuthorBase } from "./AuthorBase";

@Entity()
export class Author extends AuthorBase {

  @Column({ nullable: true })
  biography?: string;
}

```

TS AuthorBase.ts

authors-service > src > entities > TS AuthorBase.ts > ...

```ts
import { PrimaryGeneratedColumn, Column } from "typeorm";

export abstract class AuthorBase {

  @PrimaryGeneratedColumn()
  id!: number;

  @Column({ nullable: false })
  name!: string;

  @Column({ unique: true, nullable: false })
  email!: string;
}

```

Patrones usados

- Repository Pattern

TS data-source.ts | TS author.repository.ts | TS CreateAuthorDTO.ts | TS AuthorResponse

authors-service > src > repositories > TS author.repository.ts > ...

```ts
import { AppDataSource } from "../config/data-source";
import { Author } from "../entities/Author";

export const AuthorRepository = AppDataSource.getRepository(Author);

```

- DTO Pattern

CreateAuthorDTO.ts

authors-service > src > dtos > CreateAuthorDTO.ts > ...

```ts
export interface CreateAuthorDTO {
  name: string;
  email: string;
  biography?: string;
}

```

AuthorResponseDTO.ts

authors-service > src > dtos > AuthorResponseDTO.ts > ...

```ts
export interface AuthorResponseDTO {
  id: number;
  name: string;
  email: string;
  biography?: string;
}
```

- SRP (SOLID)

author.service.ts

authors-service > src > services > author.service.ts > ...

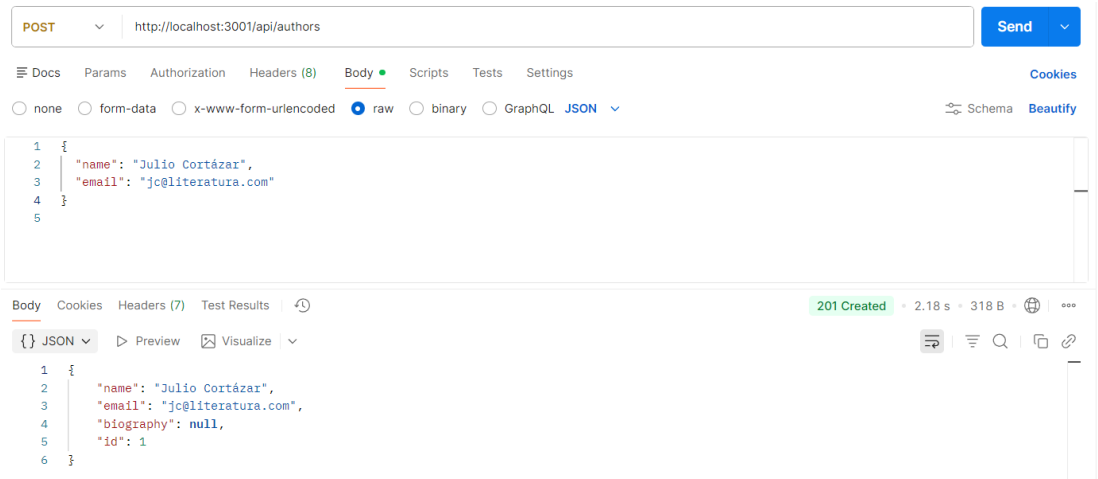
```ts
import { AuthorRepository } from "../repositories/author.repository";
import { CreateAuthorDTO } from "../dtos/CreateAuthorDTO";
import { Author } from "../entities/Author";

export class AuthorService {

  async createAuthor(data: CreateAuthorDTO): Promise<Author> {
    if (!data.name || data.name.trim() === "") {
      throw new Error("Name is required");
    }

    if (!data.email || data.email.trim() === "") {
      throw new Error("Email is required");
    }

    const existingAuthor = await AuthorRepository.findOne({
      where: { email: data.email }
    });

    if (existingAuthor) {
      throw new Error("Author with this email already exists");
    }

    const author = AuthorRepository.create(data);
    return await AuthorRepository.save(author);
  }

```

```
28    async getAuthorById(id: number) {
29      return await AuthorRepository.findOne({
30        where: { id }
31      });
32    }
33
34    async getAllAuthors() {
35      return await AuthorRepository.find();
36    }
37  }
```

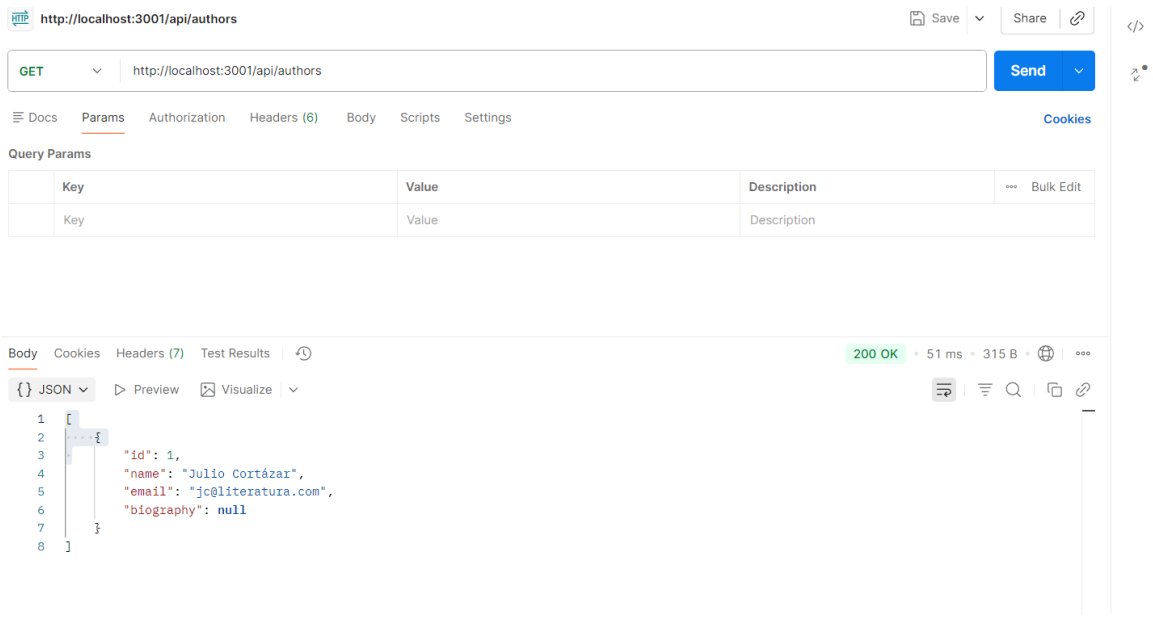
Evidencia:

Endpoints

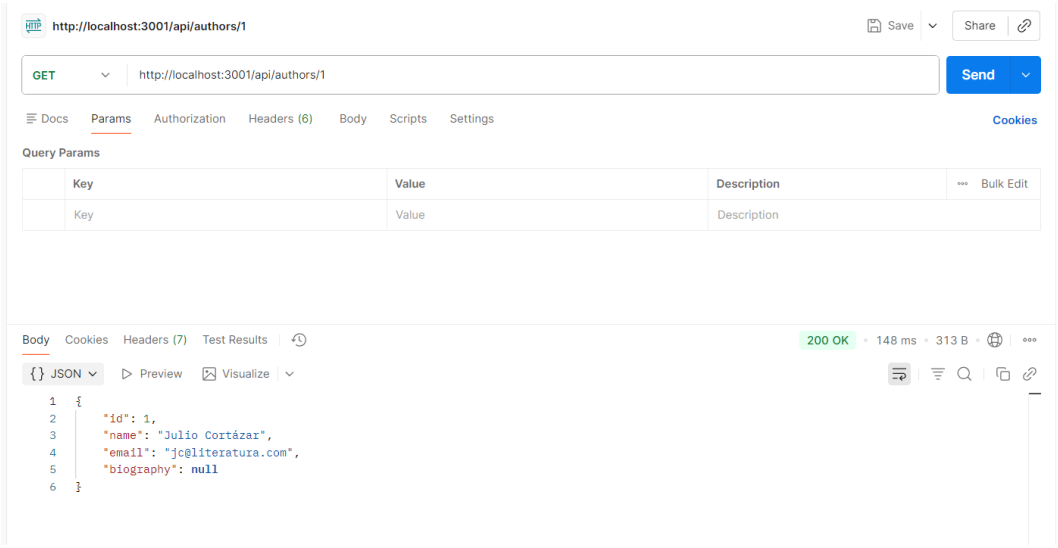
- POST /authors



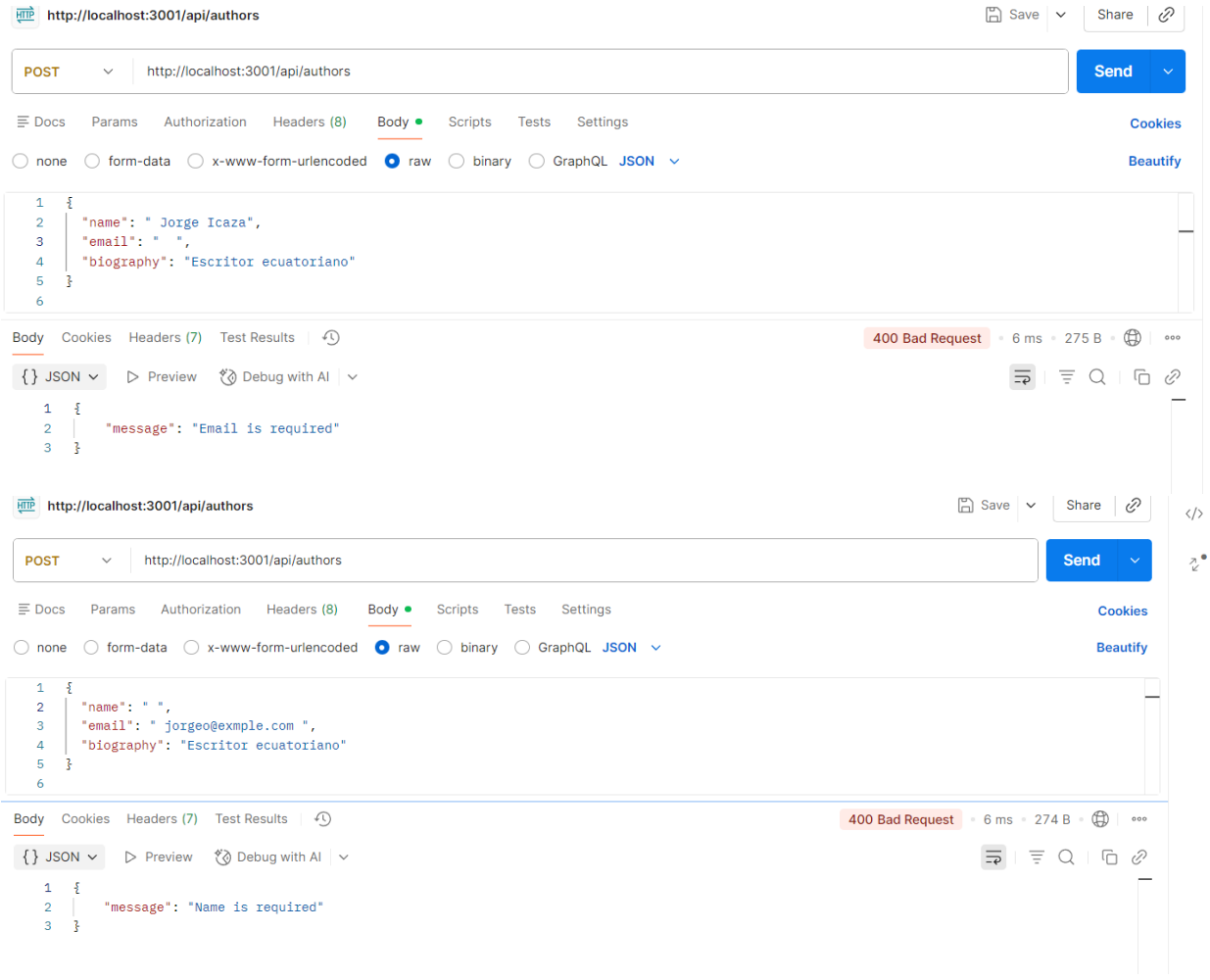
GET /authors

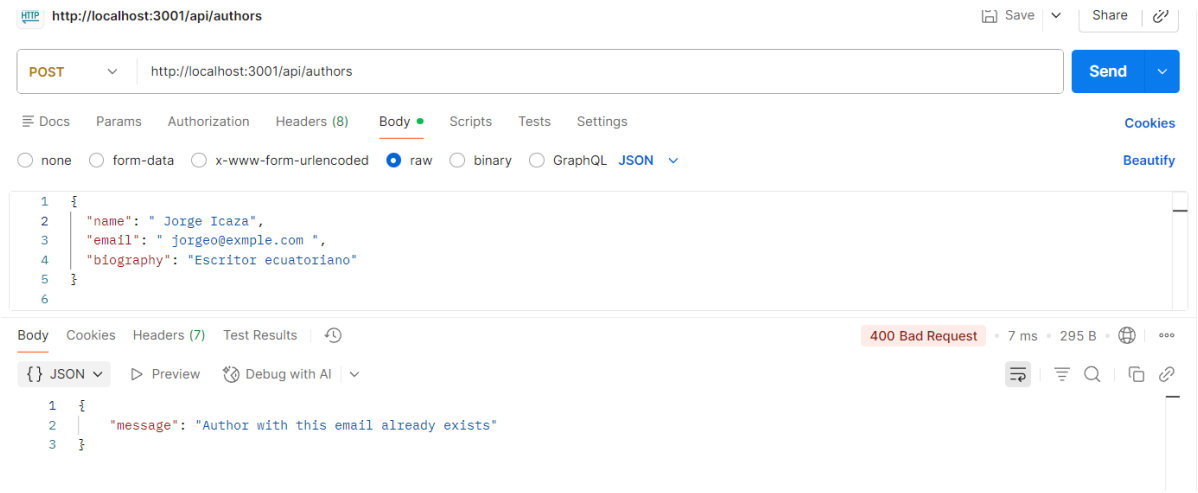


- GET /authors/:id

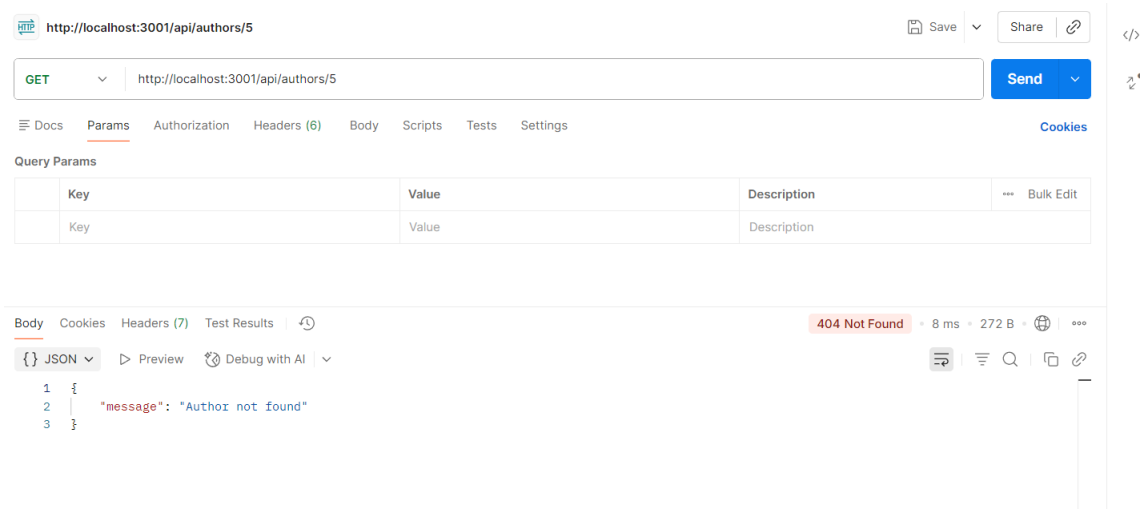


Prueba: ingresar datos incompletos





Prueba: Obtener autor por ID



Publications Service

Creamos el proyecto en nuestra carpeta principal

```
PS C:\espe1\workspace\microservicios-editorial> mkdir publications-service

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        26/1/2026     22:12                publications-service
```

Instalamos dependencias necesarias

```
PS C:\espe1\workspace\microservicios-editorial\publications-service> npm init -y

PS C:\espe1\workspace\microservicios-editorial\publications-service> npm install express typeorm mysql2 reflect-metadata axios

added 173 packages, and audited 174 packages in 24s

55 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\espe1\workspace\microservicios-editorial\publications-service> npm install -D typescript ts-node-dev @types/node @types/express
```

Configuracion del TypeScript

tsconfig.json

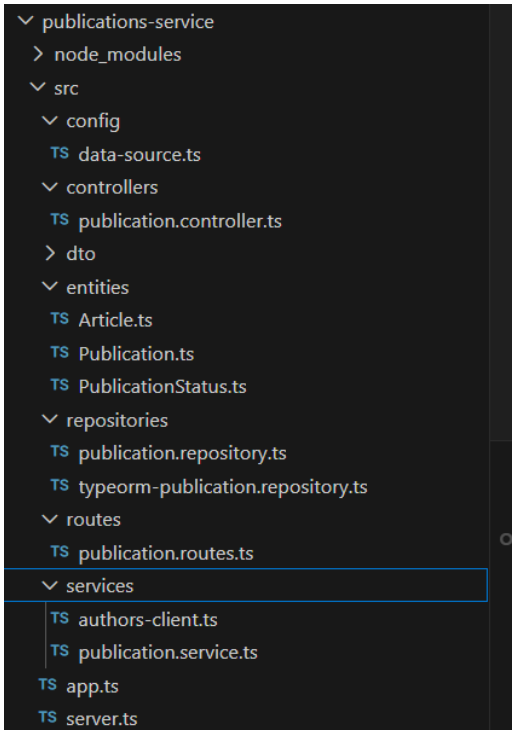
publications-service > tsconfig.json > ...

```json
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "rootDir": "src",
    "outDir": "dist",
    "esModuleInterop": true,
    "strict": true,
    "skipLibCheck": true
  }
}
```

Responsabilidad

Gestiona publicaciones editoriales.

Estructura interna del microservicio



app.ts | server.ts | package.json

publications-service > src > app.ts > ...

```ts
import express from "express";

const app = express();
app.use(express.json());

export default app;
```

Ejecución del servicio

server.ts

publications-service > src > server.ts > ...

```ts
import app from "./app";

const PORT = 3002;

app.listen(PORT, () => {
  console.log(`Publications Service running on port ${PORT}`);
});
```

Dominio implementado

Se modeló:

- Clase abstracta Publication

Publication.ts | Article.ts

publications-service > src > entities > Publication.ts > Publication > updatedAt

```ts
export abstract class Publication {
  @Column()
  authorId!: number;

  @Column({
    type: "enum",
    enum: PublicationStatus,
    default: PublicationStatus.DRAFT
  })
  status!: PublicationStatus;

  @CreateDateColumn()
  createdAt!: Date;

  @UpdateDateColumn()
  updatedAt!: Date;
}
```

- Clase concreta Article

Article.ts

publications-service > src > entities > Article.ts > ...

```ts
import { Entity, Column } from "typeorm";
import { Publication } from "./Publication";

@Entity("articles")
export class Article extends Publication {

  @Column({ default: "GENERAL" })
  category!: string;
}
```

Agregamos esto el archivo: tsconfig.json

```
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

- Enum PublicationStatus

PublicationStatus.ts

publications-service > src > entities > PublicationStatus.ts > ...

```ts
export enum PublicationStatus {
  DRAFT = "DRAFT",
  IN_REVIEW = "IN_REVIEW",
  APPROVED = "APPROVED",
  PUBLISHED = "PUBLISHED",
  REJECTED = "REJECTED"
}

```

Patrones aplicados

- Repository Pattern
- Adapter Pattern (AuthorsClient)
- SOLID (SRP + DIP)

Repository Pattern (TypeORM Repository)

- Aplicar Repository Pattern
- Encapsular el acceso a datos
- Evitar que Services usen TypeORM directamente
- Dejar base sólida para SOLID y pruebas

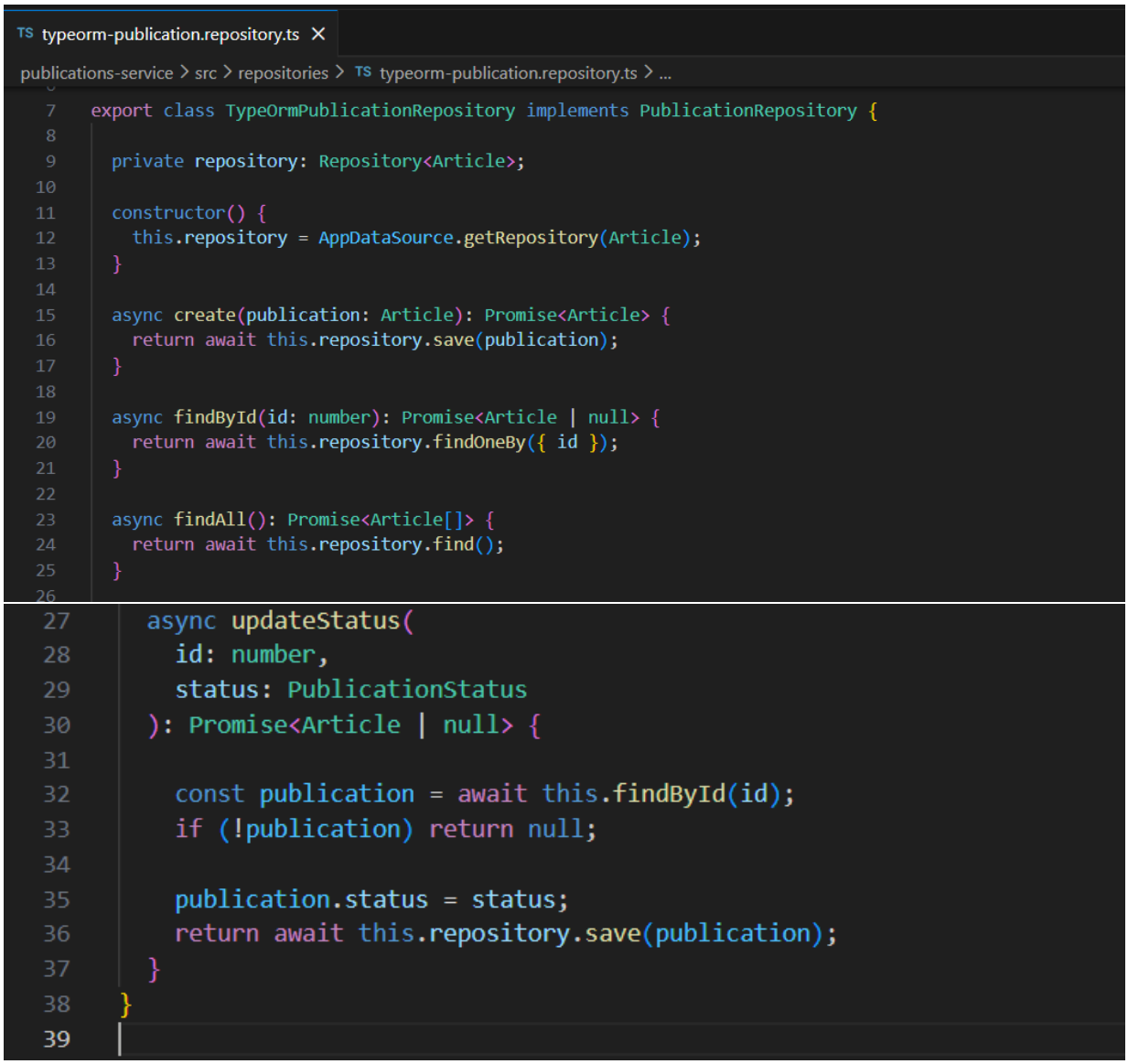
Creamos la interfaz del repositorio

publication.repository.ts | typeorm-publication.repository.ts

publications-service > src > repositories > publication.repository.ts > ...

```ts
import { Article } from "../entities/Article";
import { PublicationStatus } from "../entities/PublicationStatus";

export interface PublicationRepository {
  create(publication: Article): Promise<Article>;
  findById(id: number): Promise<Article | null>;
  findAll(): Promise<Article[]>;
  updateStatus(id: number, status: PublicationStatus): Promise<Article | null>;
}

```

Implementación con TypeORM



Se aplicó el Repository Pattern para encapsular el acceso a datos y desacoplar la lógica de negocio del ORM utilizado

Lógica de Negocio más SOLID

Crear el servicio de publicaciones

Qué SOLID estamos usando:

- SRP: El service solo maneja reglas de negocio
- DIP: Depende de PublicationRepository (interfaz), no de TypeORM
- OCP: Puedes cambiar la implementación del repositorio sin tocar el service

Validación entre servicios

Antes de crear una publicación Publications Service consulta Authors Service vía HTTP usando Axios.

Creamos el Adapter HTTP "cliente de Authors"

TS authors-client.ts

publications-service > src > services > TS authors-client.ts > ...

```ts
import axios from "axios";

export class AuthorsClient {
  private baseUrl = "http://localhost:3001/api/authors";

  async authorExists(authorId: number): Promise<boolean> {
    try {
      await axios.get(`${this.baseUrl}/${authorId}`);
      return true;
    } catch (error) {
      return false;
    }
  }
}

```

Integramos el Adapter en PublicationService

TS publication.service.ts

publications-service > src > services > TS publication.service.ts > PublicationService > constructor

```ts

export class PublicationService {

  private repository: PublicationRepository;
  private authorsClient: AuthorsClient;

  constructor() {
    this.repository = new TypeOrmPublicationRepository();
    this.authorsClient = new AuthorsClient();
  }

async createPublication(data: {
  title: string;
  content: string;
  authorId: number;
  category?: string;
}): Promise<Article> {

  const exists = await this.authorsClient.authorExists(data.authorId);
  if (!exists) {
    throw new Error("Author does not exist");
  }
```

```
30      const article = new Article();
31      article.title = data.title;
32      article.content = data.content;
33      article.authorId = data.authorId;
34      article.category = data.category || "GENERAL";
35      article.status = PublicationStatus.DRAFT;
36
37      return await this.repository.create(article);
38    }
39
```

Como se mencionó estamos usando el patrón Adapter Pattern. Donde AuthorsClient adapta HTTP y si cambia Authors, no rompes el service

Controller + Routes

Exponer endpoints REST para:

- Crear publicaciones
- Listar publicaciones
- Obtener una publicación por ID
- Cambiar estado de una publicación

Controller

publication.controller.ts | publication.routes.ts | app.ts | server.ts

publications-service > src > controllers > publication.controller.ts > PublicationController

```
1  import { Request, Response } from "express";
2  import { PublicationService } from "../services/publication.service";
3  import { PublicationStatus } from "../entities/PublicationStatus";
4
5  const service = new PublicationService();
6
7  export class PublicationController {
8
9    async create(req: Request, res: Response) {
10     try {
11       const publication = await service.createPublication(req.body);
12       res.status(201).json(publication);
13     } catch (error: any) {
14       res.status(400).json({ message: error.message });
15     }
16   }
17
18   async getAll(req: Request, res: Response) {
19     const publications = await service.getAllPublications();
20     res.json(publications);
21   }
22
23   async getById(req: Request, res: Response) {
24     const id = Number(req.params.id);
25
26     if (isNaN(id)) {
27       return res.status(400).json({ message: "Invalid ID" });
28     }
29
```

```
    const publication = await service.getPublicationById(id);

    if (!publication) {
      return res.status(404).json({ message: "Publication not found" });
    }

    res.json(publication);
  }

  async changeStatus(req: Request, res: Response) {
    const id = Number(req.params.id);
    const { status } = req.body;

    if (!Object.values(PublicationStatus).includes(status)) {
      return res.status(400).json({ message: "Invalid status" });
    }

    const updated = await service.changeStatus(id, status);

    if (!updated) {
      return res.status(404).json({ message: "Publication not found" });
    }

    res.json(updated);
  }
}
```

Routes

publication.routes.ts | app.ts | server.ts

publications-service > src > routes > publication.routes.ts > ...

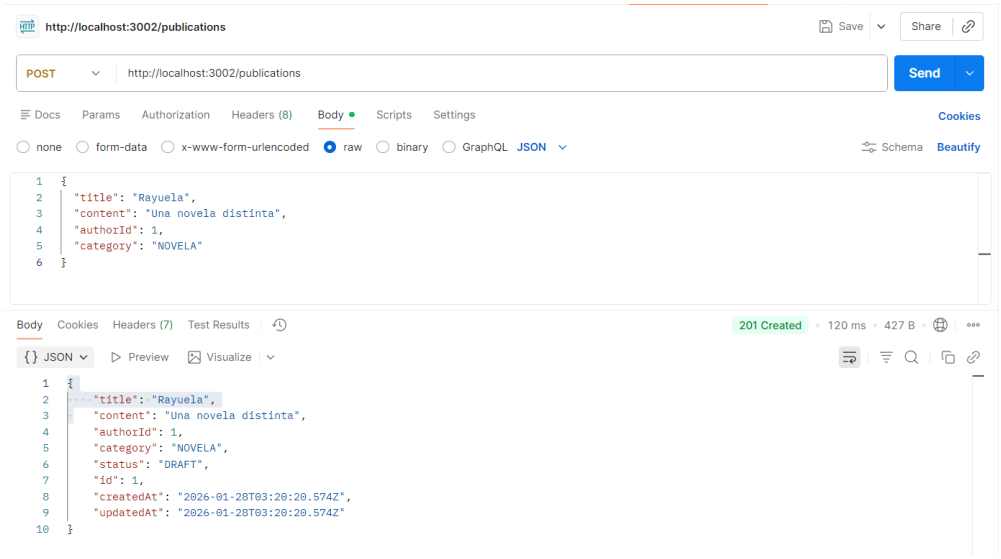
```
import { Router } from "express";
import { PublicationController } from "../controllers/publication.controller";

const router = Router();
const controller = new PublicationController();

router.post("/", controller.create);
router.get("/", controller.getAll);
router.get("/:id", controller.getById);
router.patch("/:id/status", controller.changeStatus);

export default router;
```

Registrar rutas en app.ts

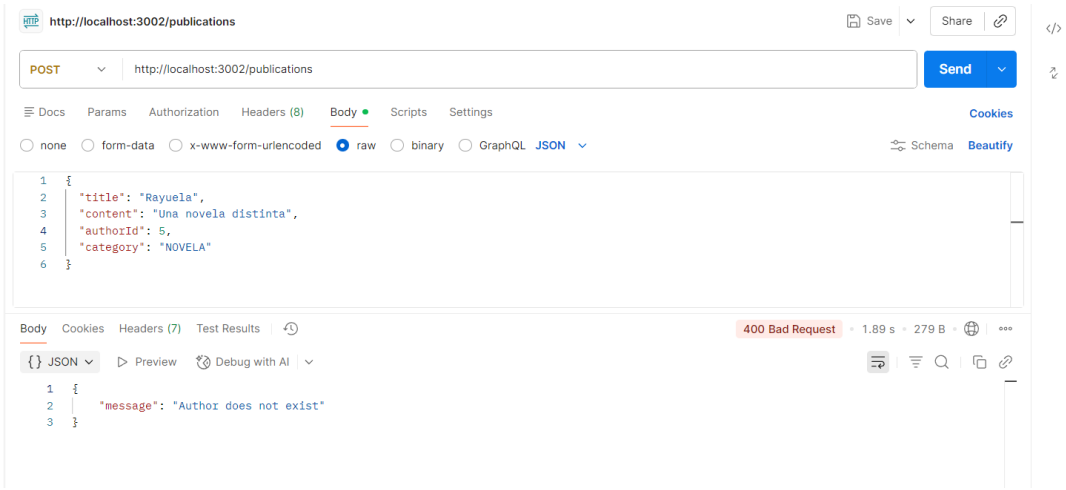
```
app.use("/publications", publicationRoutes);
```

Evidencia:

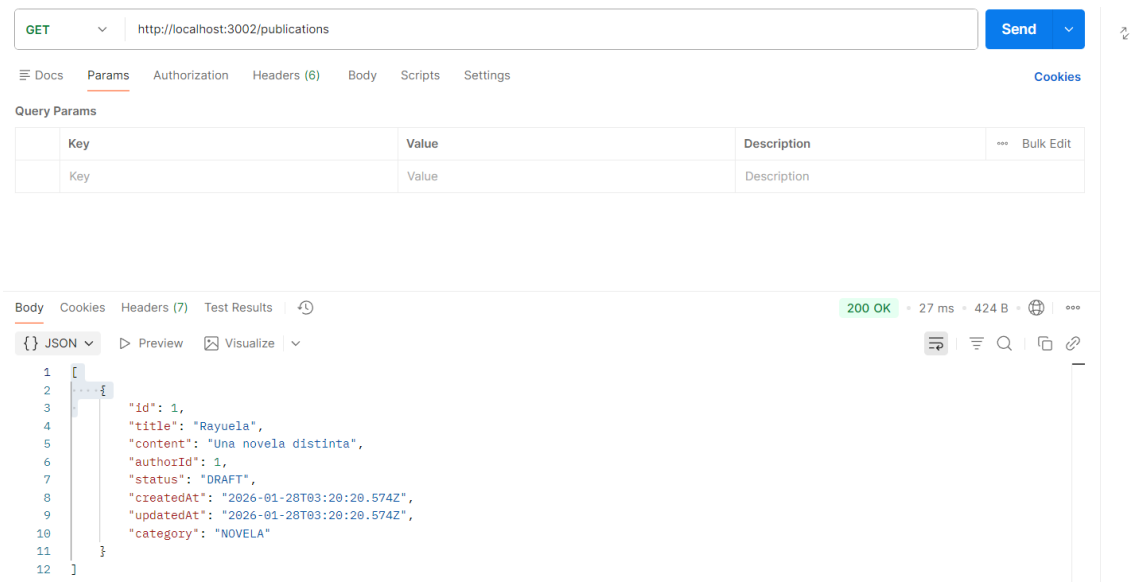
Se realizaron pruebas mediante Postman para validar el funcionamiento de los microservicios. El servicio de publicaciones verifica la existencia del autor consultando al Authors Service antes de crear una publicación.



A su vez, si se rechaza la operación por que el autor no existe, tenemos que la publicación no se realizó ya que se intenta conectar a la id de un autor inexistente.



Podemos ver posteriormente el “Listar publicaciones”



Patrones de diseño aplicados

Repository Pattern

Encapsula acceso a base de datos.

Adapter Pattern

Implementado en AuthorsClient.

Adapta HTTP: método simple authorExists().

Persistencia

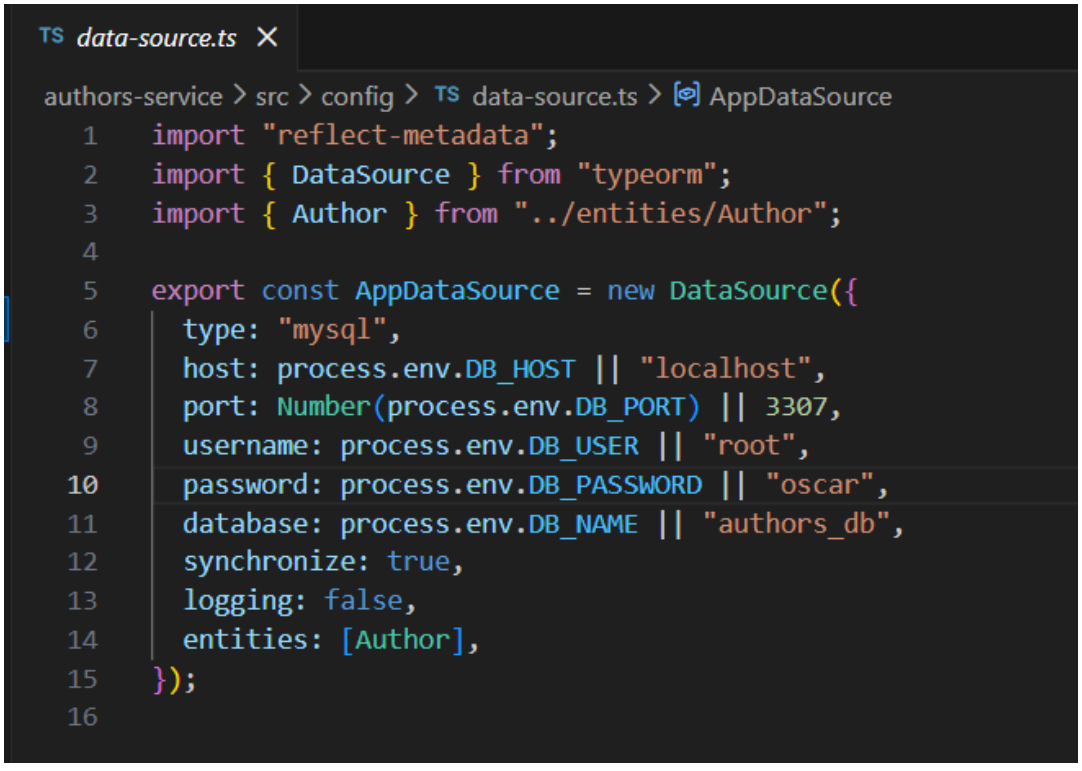
Cada microservicio tiene su propia base de datos MySQL.

- authors_db
- publications_db

Configuración de Base de Datos (TypeORM + MySQL)

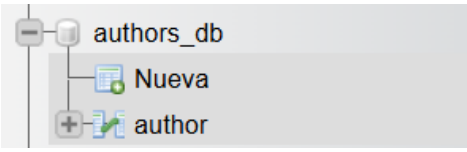
- Configuramos TypeORM con DataSource
- Conectamos Authors Service a su propia base de datos

Tenemos el archivo: data-source.ts de "Authors-service" para la configuración de la BD



Inicializamos la conexión al arrancar el servidor

```ts
// server.ts
// authors-service > src > server.ts > ...
import app from "./app";
import { AppDataSource } from "./config/data-source";

const PORT = 3001;

AppDataSource.initialize()
  .then(() => {
    console.log(" Authors DB connected");
    app.listen(PORT, () => {
      console.log(` Authors Service running on port ${PORT}`);
    });
  })
  .catch((error) => console.error("DB connection error:", error));
```



Configuración de Base de Datos para el servicio de "Publications-Service***"***

- Configuramos TypeORM con DataSource
- Conectamos Publications Service a su propia base de datos

Tenemos el archivo: data-source.ts para la configuración de la BD

```ts
// data-source.ts
// publications-service > src > config > data-source.ts > AppDataSource > logging
import "reflect-metadata";
import { DataSource } from "typeorm";
import { Article } from "../entities/Article";

export const AppDataSource = new DataSource({
  type: "mysql",
  host: process.env.DB_HOST || "localhost",
  port: Number(process.env.DB_PORT) || 3307,
  username: process.env.DB_USER || "root",
  password: process.env.DB_PASSWORD || "oscar",
  database: process.env.DB_NAME || "publications_db",

  entities: [Article],
  synchronize: true,
  logging: false
});
```

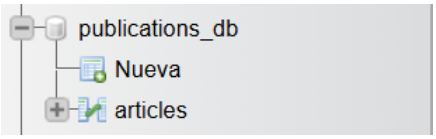
Inicializamos la conexión al arrancar el servidor

TS server.ts

publications-service > src > TS server.ts > ...

```ts
import "reflect-metadata";
import app from "./app";
import { AppDataSource } from "./config/data-source";

const PORT = 3002;

AppDataSource.initialize()
  .then(() => {
    console.log("Publications DB connected");

    app.listen(PORT, () => {
      console.log(`Publications Service running on port ${PORT}`);
    });
  })
  .catch((error) => {
    console.error("DB connection error:", error);
  });

```

- publications_db



Los servicios están configurados con TypeORM + DataSource. Ademas servirán para usar variables de entorno usadas en Docker.

Contenerización con Docker

Dockerfile para cada microservicio

- Authors Service

Dockerfile

authors-service > Dockerfile > ...

```dockerfile
FROM node:18-alpine

# Directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiamos dependencias
COPY package*.json ./

RUN npm install

# Copiamos el código
COPY . .

# Compilamos TypeScript (si no usas build, se queda igual)
# RUN npm run build

EXPOSE 3001

CMD ["npm", "run", "dev"]

```

- Publications Service

Dockerfile

publications-service > Dockerfile > ...

```
1   FROM node:18-alpine
2
3   WORKDIR /app
4
5   COPY package*.json ./
6
7   RUN npm install
8
9   COPY . .
10
11  EXPOSE 3002
12
13  CMD ["npm", "run", "dev"]
14
```

Evidencia:

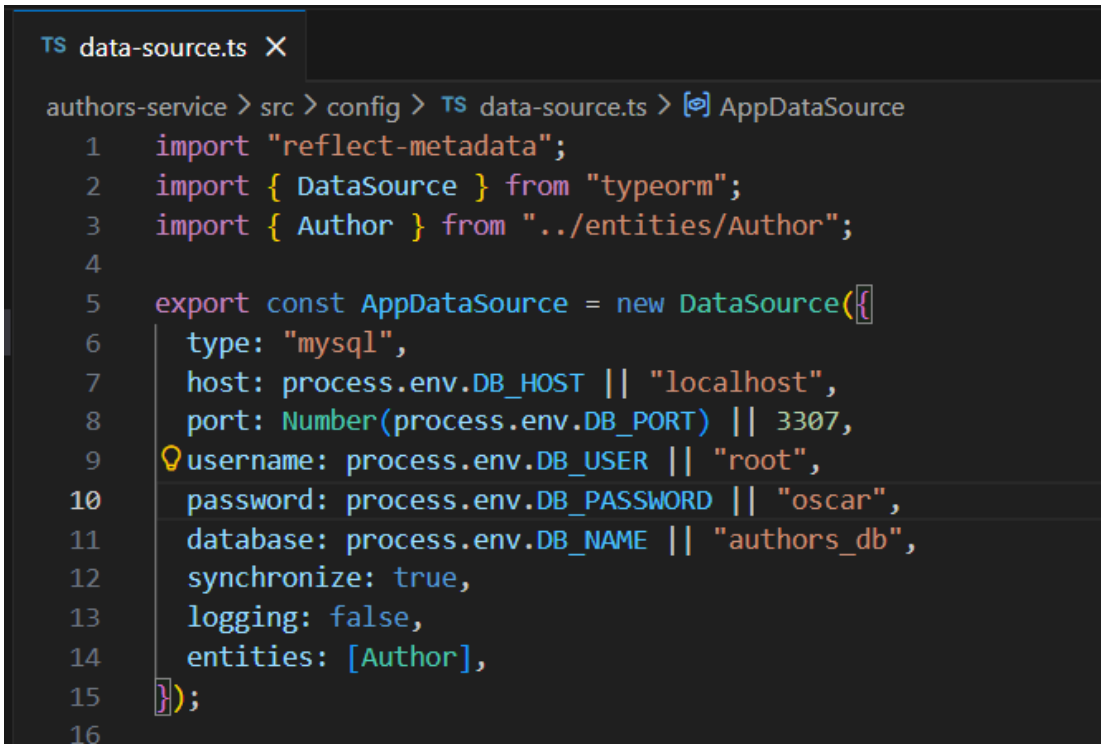
Creamos el archivo docker-compose.yml

```
5   # ========================
6   # Authors Database
7   # ========================
    ▷Run Service
8   db-authors:
9     image: mysql:8.0
10    container_name: db-authors
11    restart: always
12    environment:
13      MYSQL_ROOT_PASSWORD: root
14      MYSQL_DATABASE: authors_db
15    ports:
16      - "3307:3306"
17    volumes:
18      - db_authors_data:/var/lib/mysql
19    networks:
20      - editorial-network
21
```

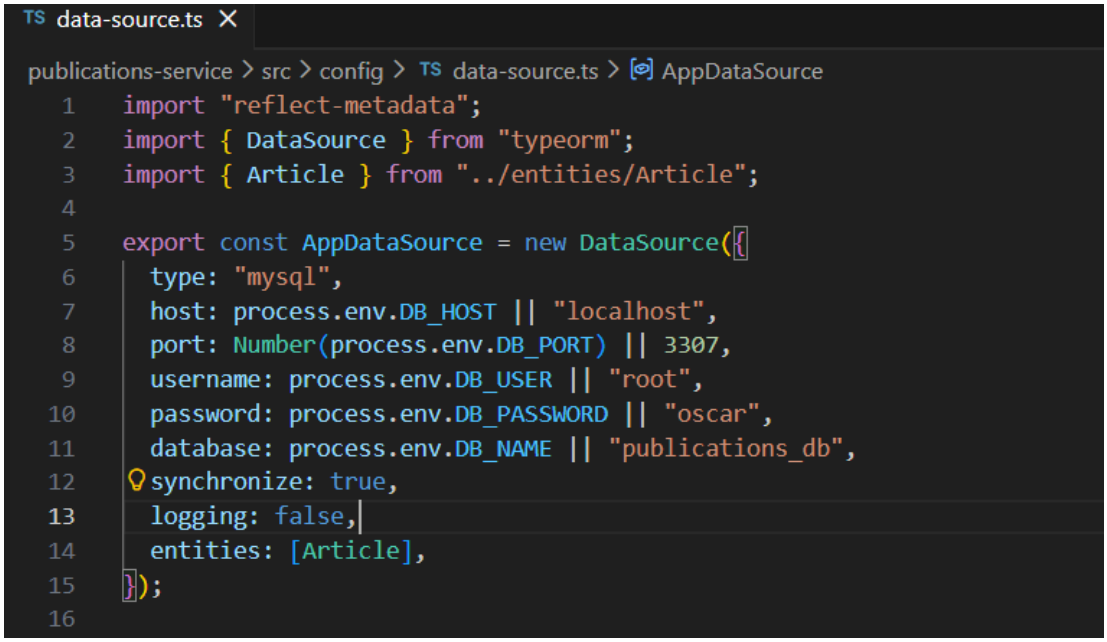
```yaml
# docker-compose.yml
services:
  # Publications Database
  # ========================
  ▷Run Service
  db-publications:
    image: mysql:8.0
    container_name: db-publications
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: publications_db
    ports:
      - "3308:3306"
    volumes:
      - db_publications_data:/var/lib/mysql
    networks:
      - editorial-network

  # ========================
  # Authors Service
  # ========================
  ▷Run Service
  authors-service:
    build: ./authors-service
    container_name: authors-service
    restart: always
    ports:
      - "3001:3001"
    environment:
      DB_HOST: db-authors
      DB_PORT: 3306
      DB_USER: root
      DB_PASSWORD: root
      DB_NAME: authors_db
    depends_on:
      - db-authors
    networks:

  # ========================
  # Publications Service
  # ========================
  ▷Run Service
  publications-service:
    build: ./publications-service
    container_name: publications-service
    restart: always
    ports:
      - "3002:3002"
    environment:
      DB_HOST: db-publications
      DB_PORT: 3306
      DB_USER: root
      DB_PASSWORD: root
      DB_NAME: publications_db
      AUTHORS_SERVICE_URL: http://authors-service:3001/api/authors
    depends_on:
      - db-publications
      - authors-service
    networks:
      - editorial-network
```

Ajustar data-source.ts para Docker donde tendremos env + host + puerto

Authors Service



Publications Service



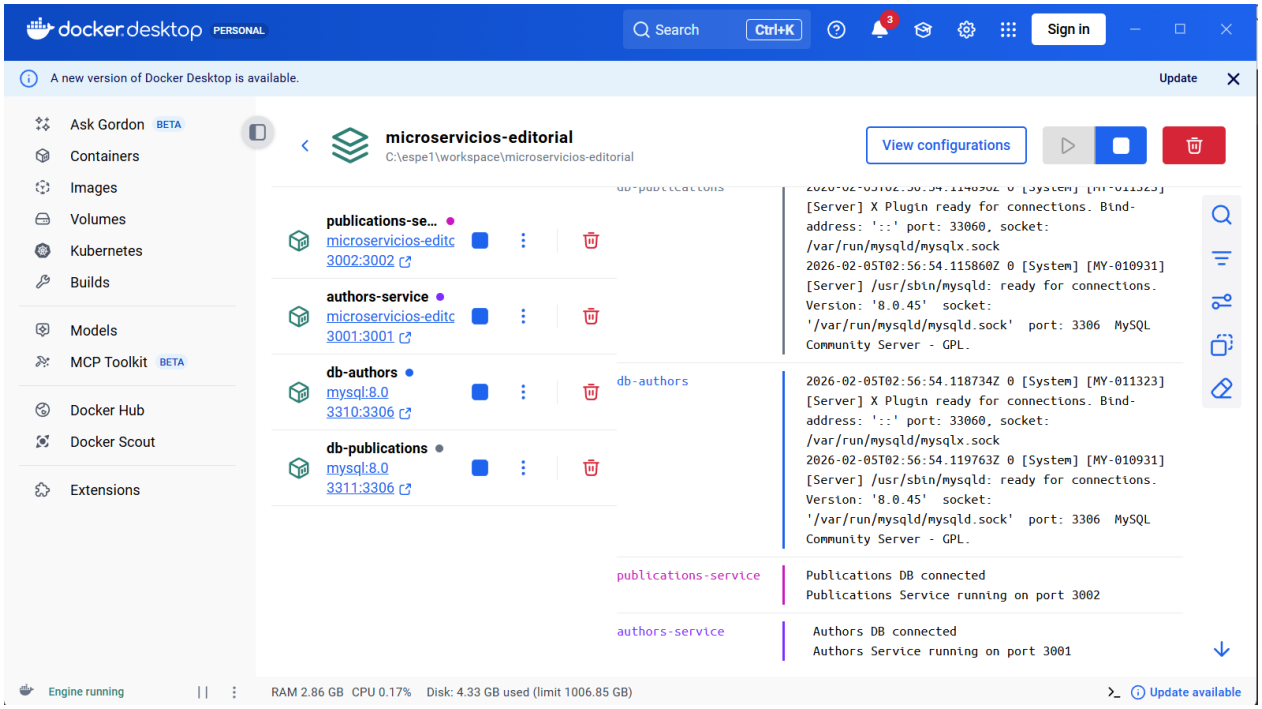
La configuración de la base de datos se realizó mediante variables de entorno, permitiendo que los microservicios funcionen tanto en entorno local como en contenedores Docker.

Levantar todo con Docker Compose: docker compose up –build

```
PS C:\espe1\workspace\microservicios-editorial> docker compose up --build
publications-service  | Publications DB connected
publications-service  | Publications Service running on port 3002
```

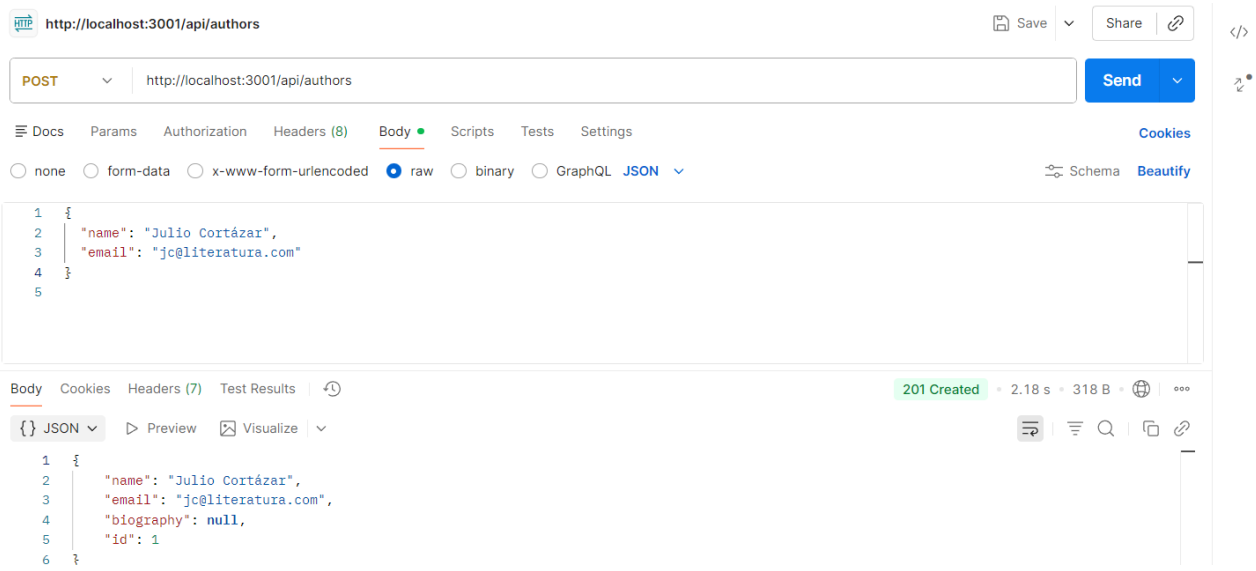
```
authors-service       |
authors-service       | [INFO] 02:57:55 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.2, typescript ver. 5.9.3)
authors-service       | 📦 Authors DB connected
authors-service       | 🚀 Authors Service running on port 3001
```

Vemos en Docker los servicios conectados

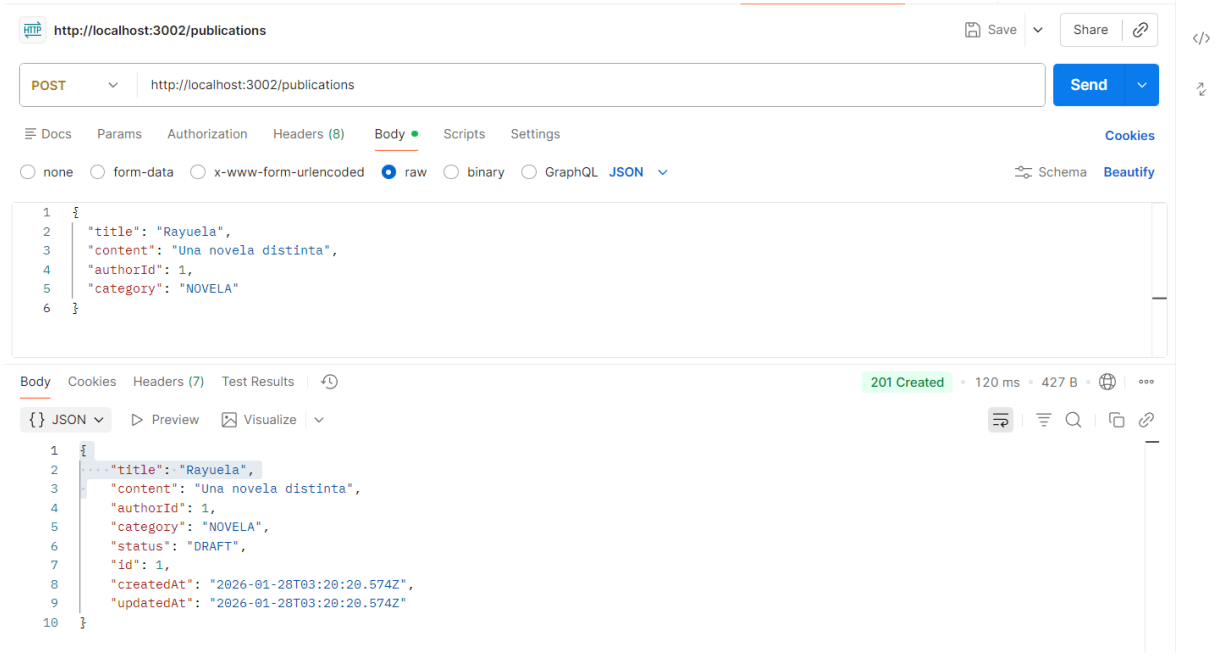


Pruebas finales con Postman

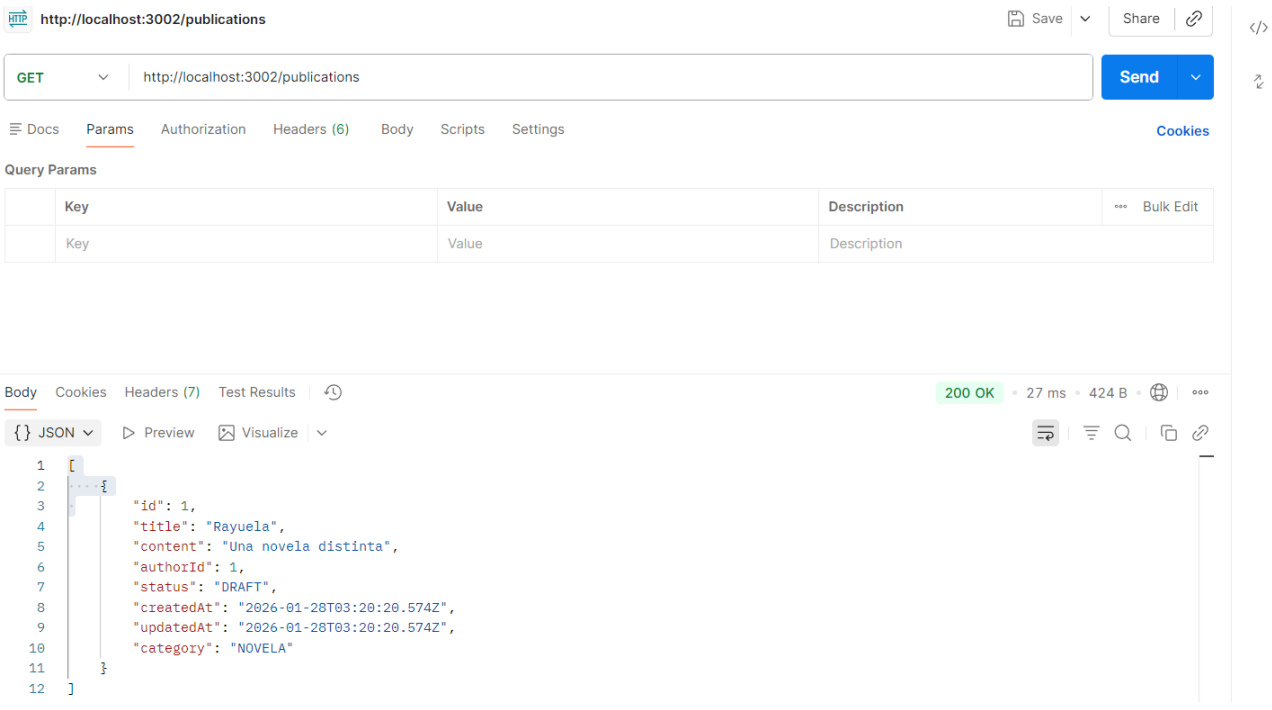
Crear autor



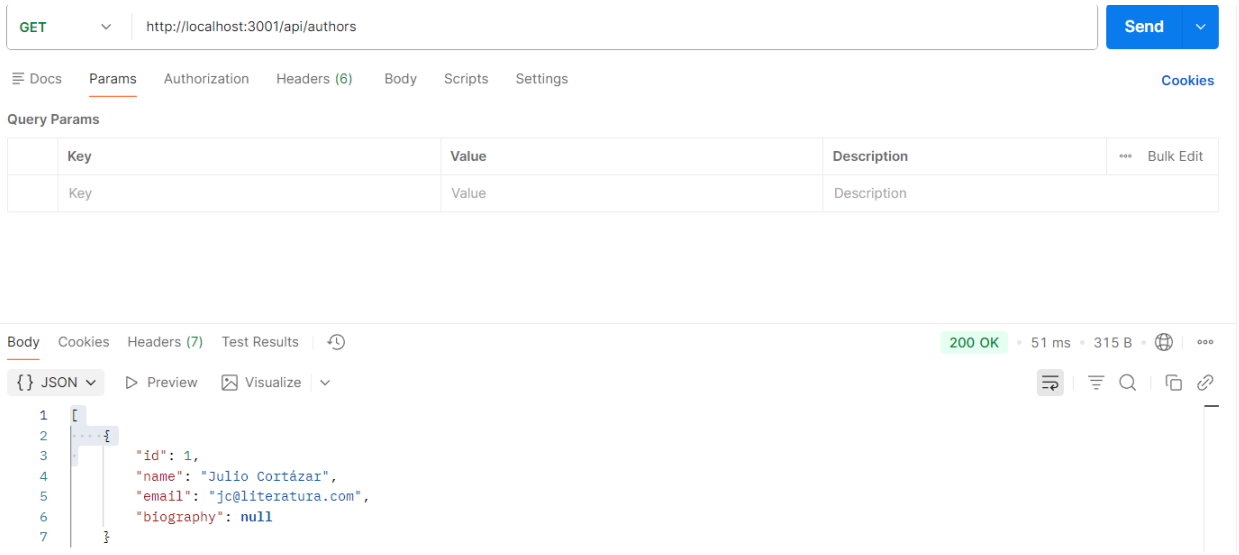
Crear publicación



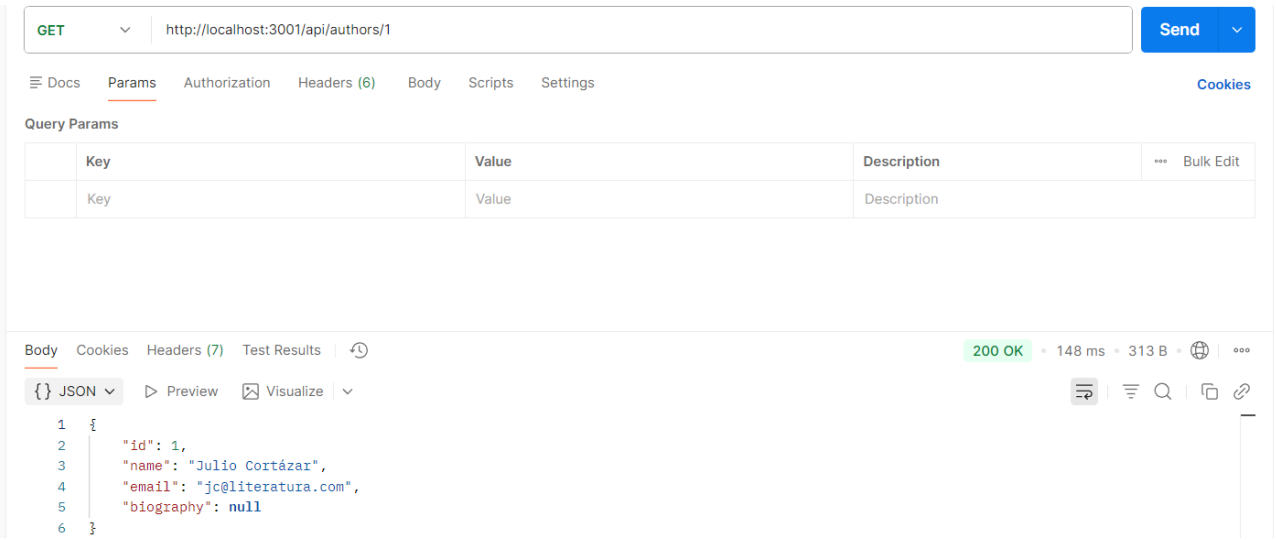
Listar publicaciones



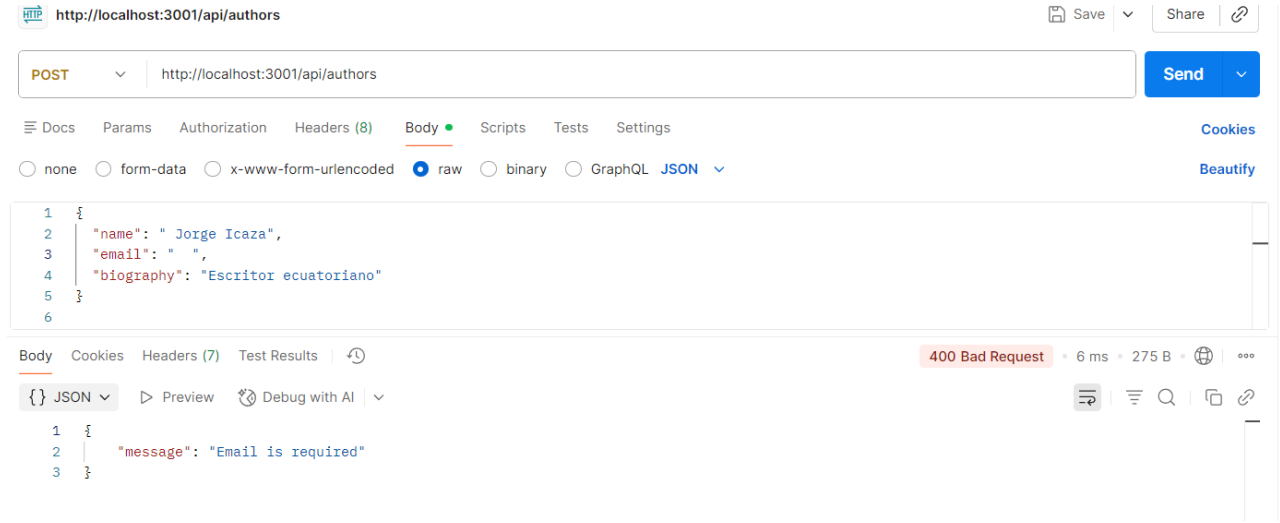
Listar autores



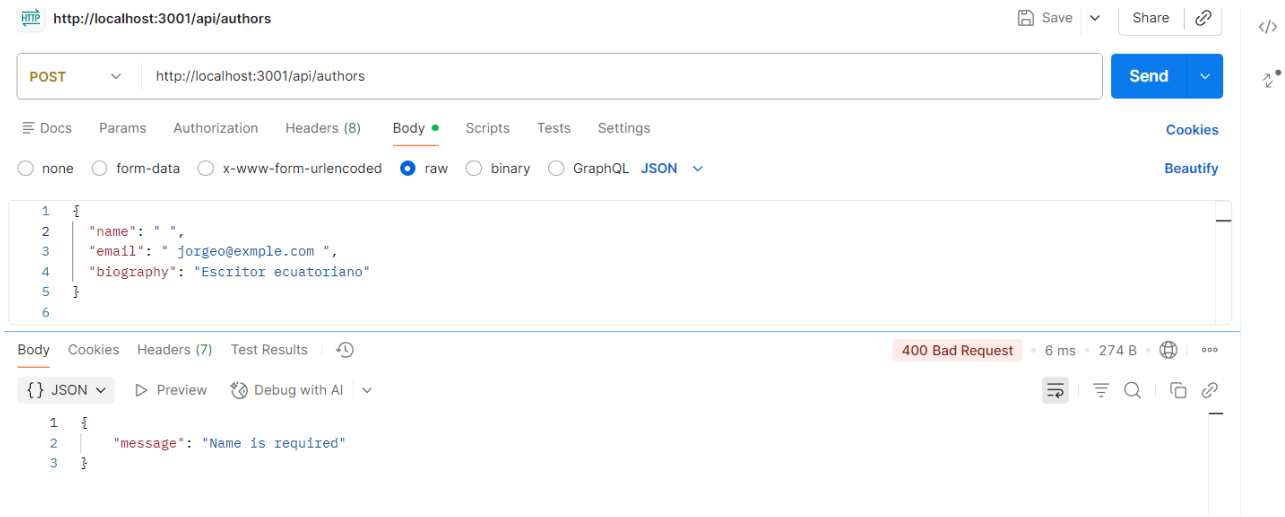
Buscar autor con authorId correcto



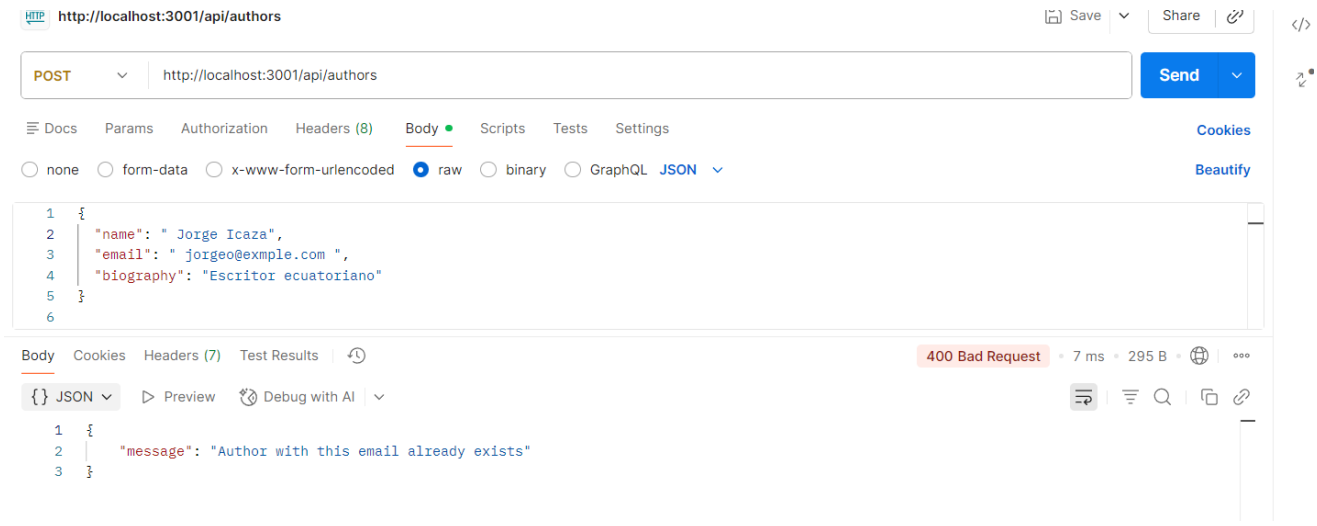
Prueba: ingresar datos incompletos



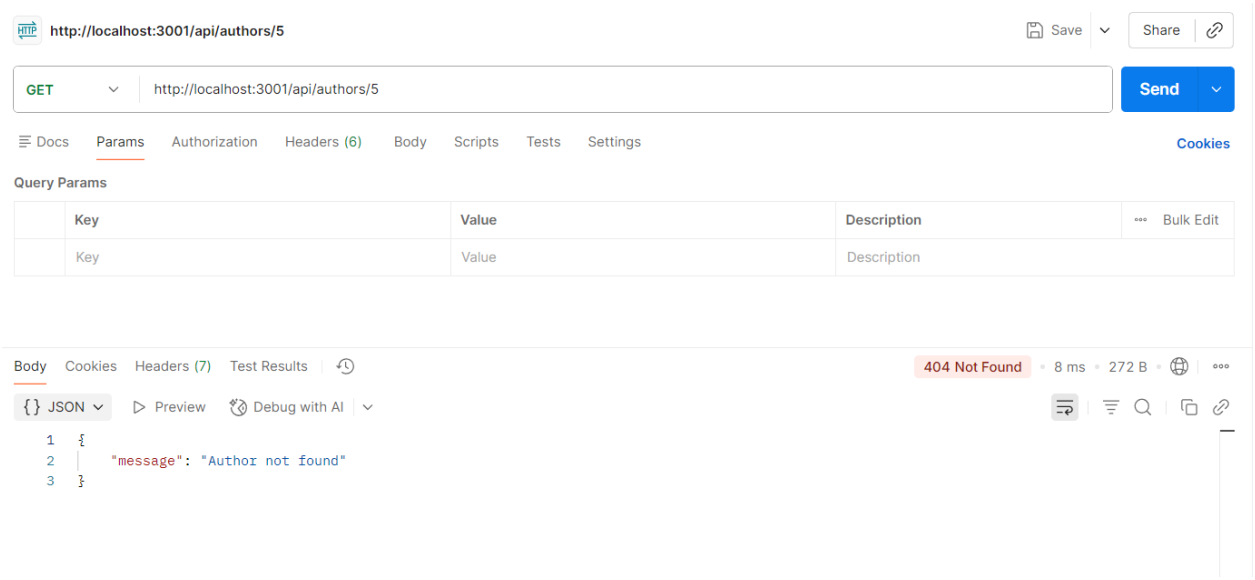
Prueba: Cuando falta el nombre del autor



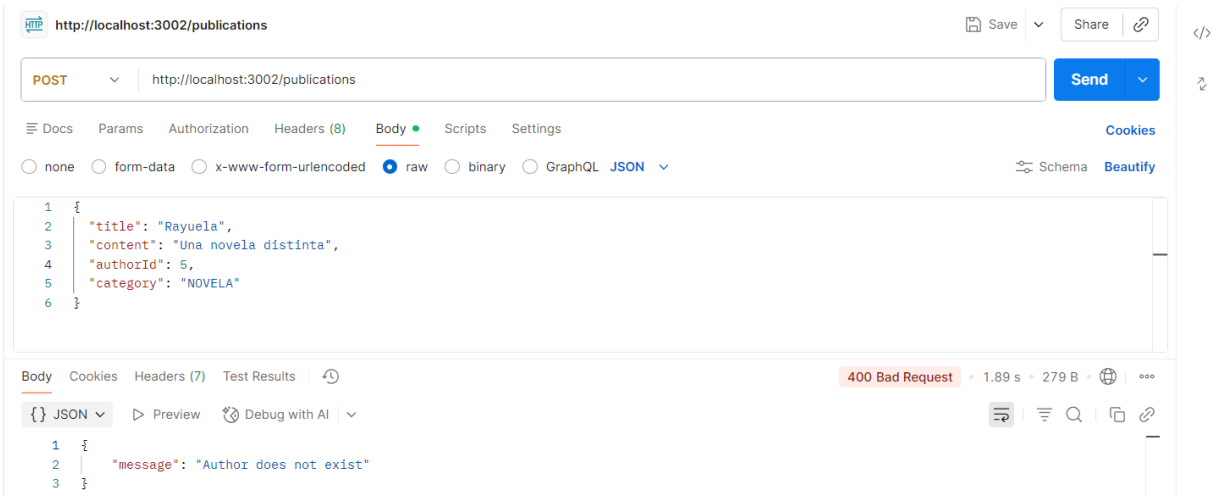
Cuando falta el nombre y el email del autor



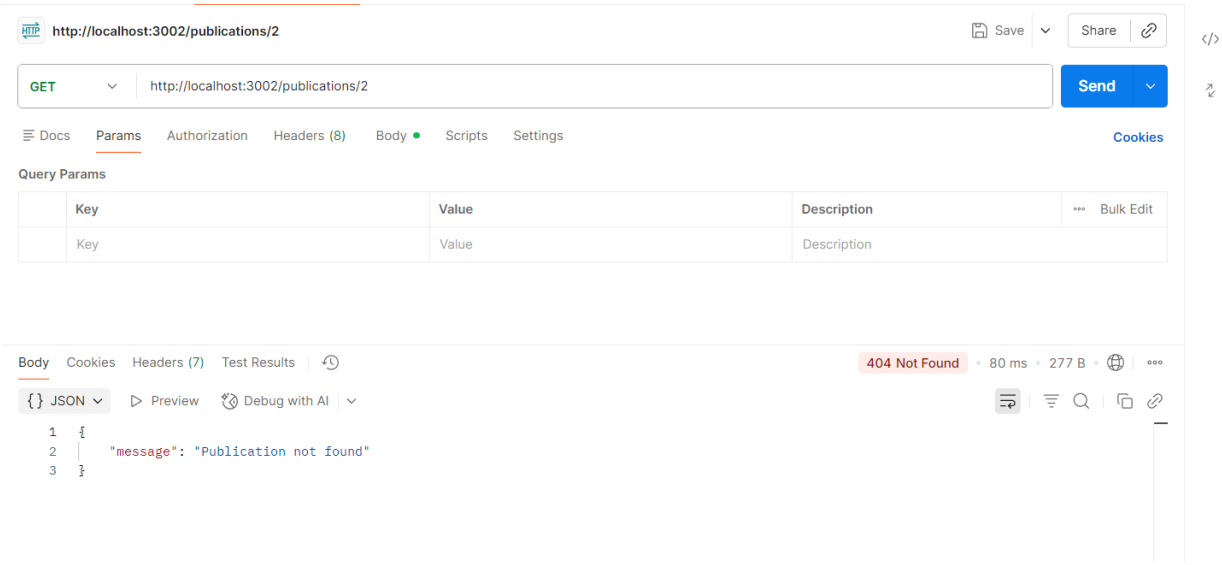
Prueba: Obtener autor por ID inexistente



Crear publicación con la id de un autor inexistente.



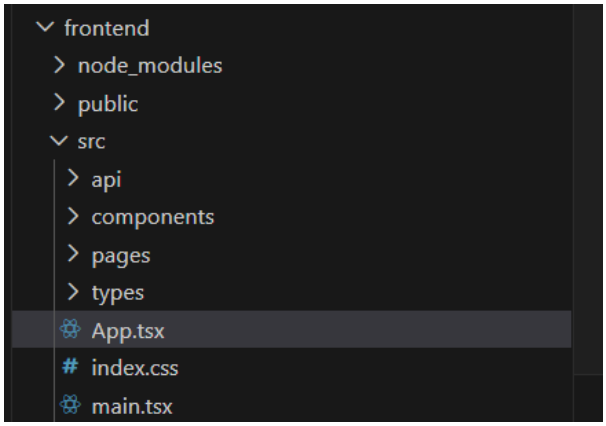
Buscar Endpoint inexistente en publicaciones



Frontend

Se desarrolló un frontend en React + TypeScript que consume los microservicios Authors y Publications, donde se puede crear y listar autores y publicaciones, validando dependencias entre microservicios en tiempo real.

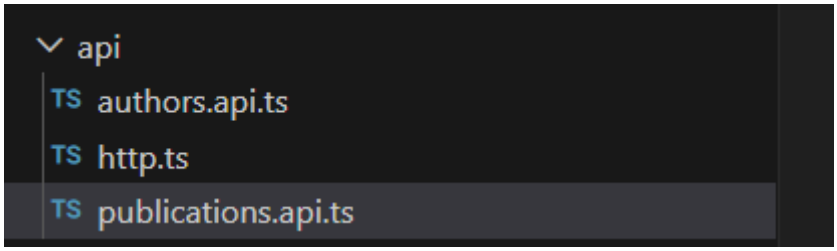
Estructura base



Configuración de conexión con los microservicios

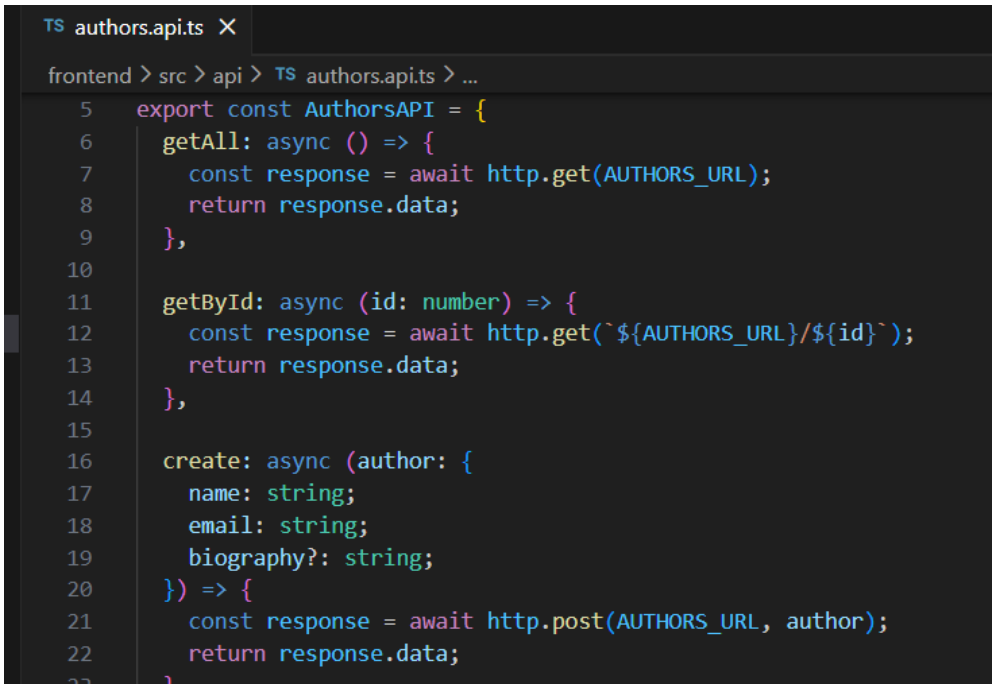
Creamos una capa api/ que tambien centralizamos las llamadas HTTP y sepparamos el frontend dle backend

Estructura del API layer

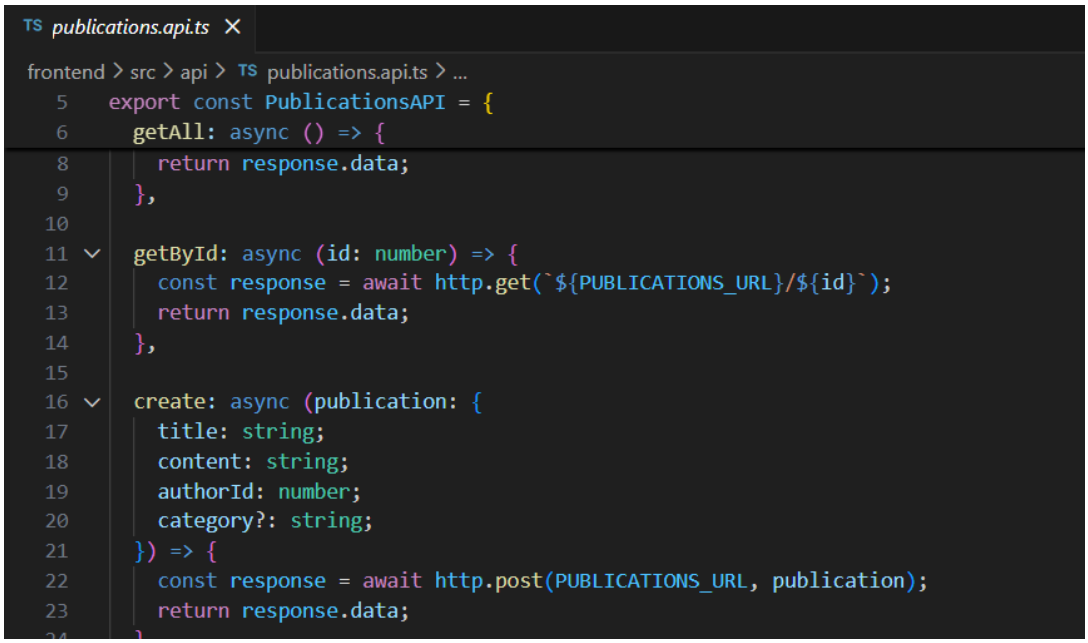


Configuración del cliente HTTP

Api de Authors



Api de Publications



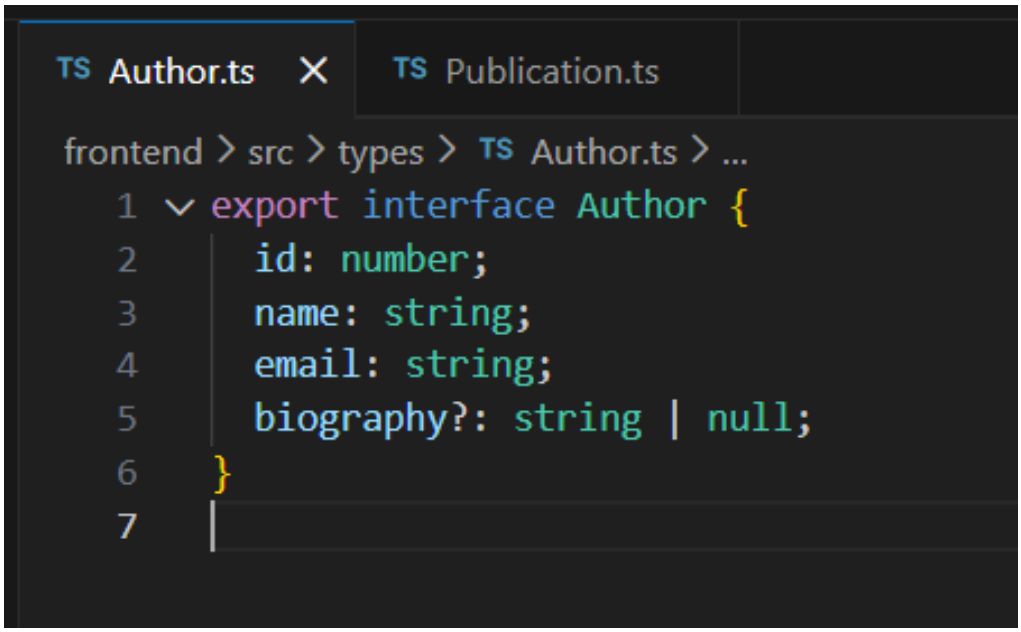
Se implementó una capa de acceso a datosen el frontend, el cual se encargó de la comunicación con los microservicios Authors y Publications mediante Axios

Definición de tipos TypeScript (models)

Aquí definimos loas interfaces TypeScript que representen exactamente:

- Lo que devuelve Authors Service
- Lo que envía y devuelve Publications Service

Crear tipos para Authors



Crear tipos para Publications

Aquí definimos los modelos TypeScript en el frontend para representar las entidades Author y Publication.

Vista: listar autores desde el backend

Creamos la página AuthorsPage

AuthorsPage.tsx

frontend > src > pages > AuthorsPage.tsx > AuthorsPage > useEffect() callback > loadAuthors

```tsx
export const AuthorsPage = () => {
  const [authors, setAuthors] = useState<Author[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState("");

  useEffect(() => {
    const loadAuthors = async () => {
      try {
        const data = await AuthorsAPI.getAll();
        setAuthors(data);
      } catch (err) {
        setError("Error al cargar autores");
      } finally {
        setLoading(false);
      }
    };

    loadAuthors();
  }, []);

  if (loading) return <p>Cargando autores...</p>;
  if (error) return <p>{error}</p>;

  return (
    <div>
      <h2>Autores</h2>
      <ul>
        {authors.map((author) => (
          <li key={author.id}>
            <strong>{author.name}</strong> — {author.email}
          </li>
        ))}
      </ul>
    </div>
  );
};
```

Verificamos el tipo Author

TS author.ts | App.tsx

frontend > src > pages > TS author.ts > ...

```ts
export interface Author {
  id: number;
  name: string;
  email: string;
  biography?: string | null;
}
```

Conectamos la vista en App.tsx

```
TS author.ts    App.tsx
frontend > src > App.tsx > ...
1   import { AuthorsPage } from "./pages/AuthorsPage";
2
3   function App() {
```

Instala CORS

Dentro de authors-service instalamos CORS

```
PS C:\espe1\workspace\microservicios-editorial\authors-service> npm install cors

added 3 packages, and audited 246 packages in 3s

64 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\espe1\workspace\microservicios-editorial\authors-service>
```

Tambien sus dependencias:

```
PS C:\espe1\workspace\microservicios-editorial\authors-service> npm install --save-dev @types/cors

added 1 package, and audited 247 packages in 2s

64 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Dentro de publications-service instalamos CORS y sus dependencias

```
PS C:\espe1\workspace\microservicios-editorial\publications-service> npm install cors

added 2 packages, and audited 246 packages in 2s

64 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\espe1\workspace\microservicios-editorial\publications-service> npm install --save-dev @types/cors

added 1 package, and audited 247 packages in 1s

64 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\espe1\workspace\microservicios-editorial\publications-service>
```

Y agregamos en app.ts

```
2   import cors from "cors";
```

Se instalaron los tipos de TypeScript para el middleware CORS (@types/cors) en cada microservicio, garantizando compatibilidad con TypeScript y correcta compilación del proyecto.

Vista para crear Autor en el formulario

Crear una pantalla en el frontend que permita crear autores usando el authors-service.

Creamos la página CreateAuthorPage.tsx

app.ts | CreateAuthorPage.tsx

frontend > src > pages > CreateAuthorPage.tsx > CreateAuthorPage > handleSubmit

```tsx
import { useState } from "react";
import { AuthorsAPI } from "../api/authors.api";

export default function CreateAuthorPage() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [biography, setBiography] = useState("");
  const [message, setMessage] = useState("");

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();

    try {
      await AuthorsAPI.create({
        name,
        email,
        biography,
      });

      setMessage("Autor creado correctamente ");
      setName("");
      setEmail("");
      setBiography("");
    } catch (error) {
      setMessage("Error al crear autor ");
    }
  };

  return (
    <div>
      <h2>Crear Autor</h2>

      <form onSubmit={handleSubmit}>
        <div>
          <input
            type="text"
            placeholder="Nombre"
            value={name}
            onChange={(e) => setName(e.target.value)}
            required
          />
        </div>

        <div>
          <input
            type="email"
            placeholder="Email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
            required
          />
        </div>
```

Registramos la ruta en App.tsx

App.tsx

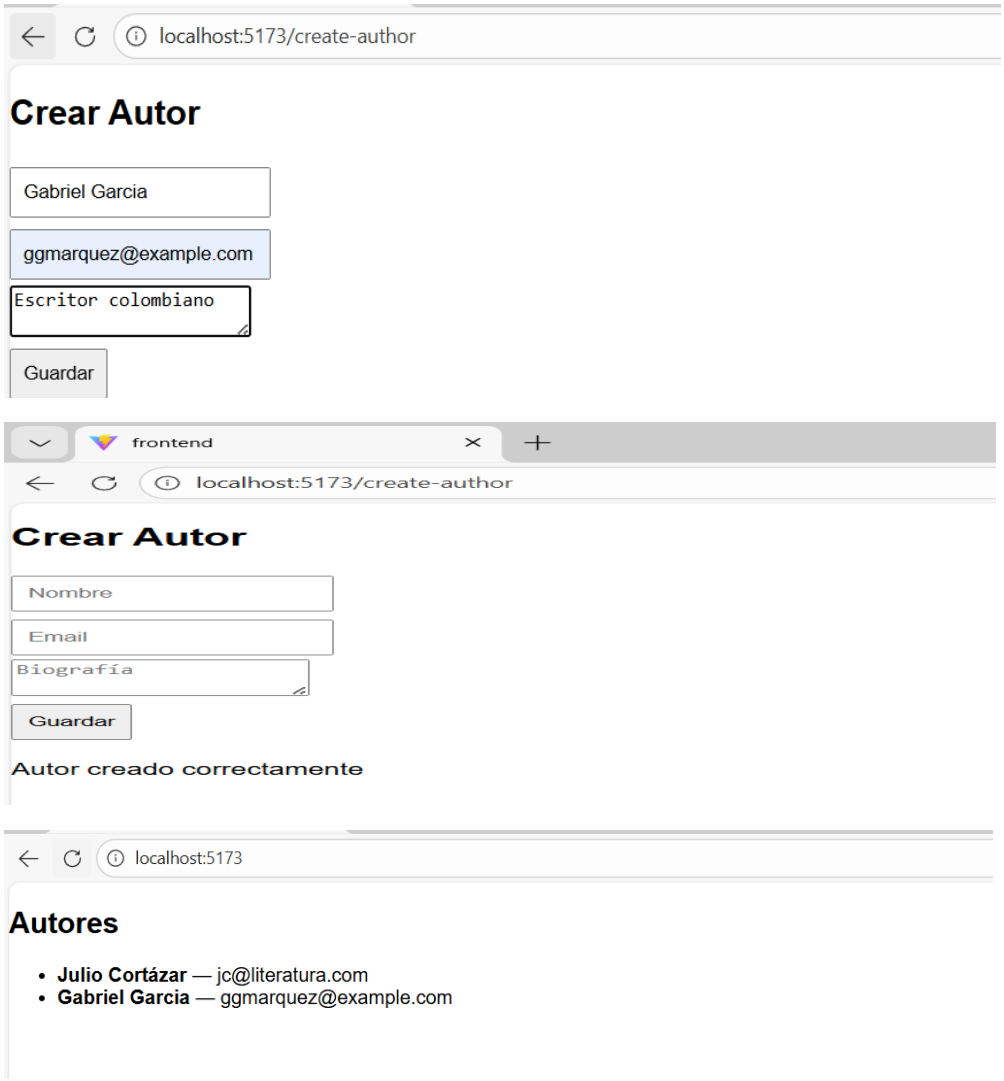
frontend > src > App.tsx > ...

```tsx
import { AuthorsPage } from "./pages/AuthorsPage";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import CreateAuthorPage from "./pages/CreateAuthorPage";
```

Se implementó un formulario en React para la creación de autores, consumiendo el microservicio Authors mediante llamadas HTTP usando Axios. El frontend valida y envía datos al backend, comprobando la integración completa.

EVIDENCIA WEB

Evidenciamos en el navegador nuestro "crear autor":



Crear Publicaciones desde el Frontend

Definimos el modelo TypeScript de Publication

TS publication.ts

frontend > src > types > TS publication.ts > ...

```ts
export type Category = "NOVELA" | "POESIA" | "ENSAYO";

export interface Publication {
  title: string;
  content: string;
  authorId: number;
  category: Category | "";
}
```

Crear la página CreatePublicationPage.tsx

```tsx
return (
  <div>
    <h2>Crear Publicación</h2>

    <form onSubmit={handleSubmit}>
      <input
        name="title"
        placeholder="Título"
        value={form.title}
        onChange={handleChange}
        required
      />

      <input
        name="content"
        placeholder="Contenido"
        value={form.content}
        onChange={handleChange}
        required
```

```tsx
      <select
        name="authorId"
        value={form.authorId}
        onChange={handleChange}
        required
      >
        <option value="">Seleccione autor</option>
        {authors.map((author) => (
          <option key={author.id} value={author.id}>
            {author.name}
          </option>
        ))}
      </select>

      <select
        name="category"
        value={form.category}
        onChange={handleChange}
        required
      >
        <option value="">Seleccione categoría</option>
        <option value="NOVELA">Novela</option>
        <option value="CUENTO">Cuento</option>
        <option value="POESIA">Poesía</option>
      </select>


      <button type="submit">Crear</button>
    </form>
```

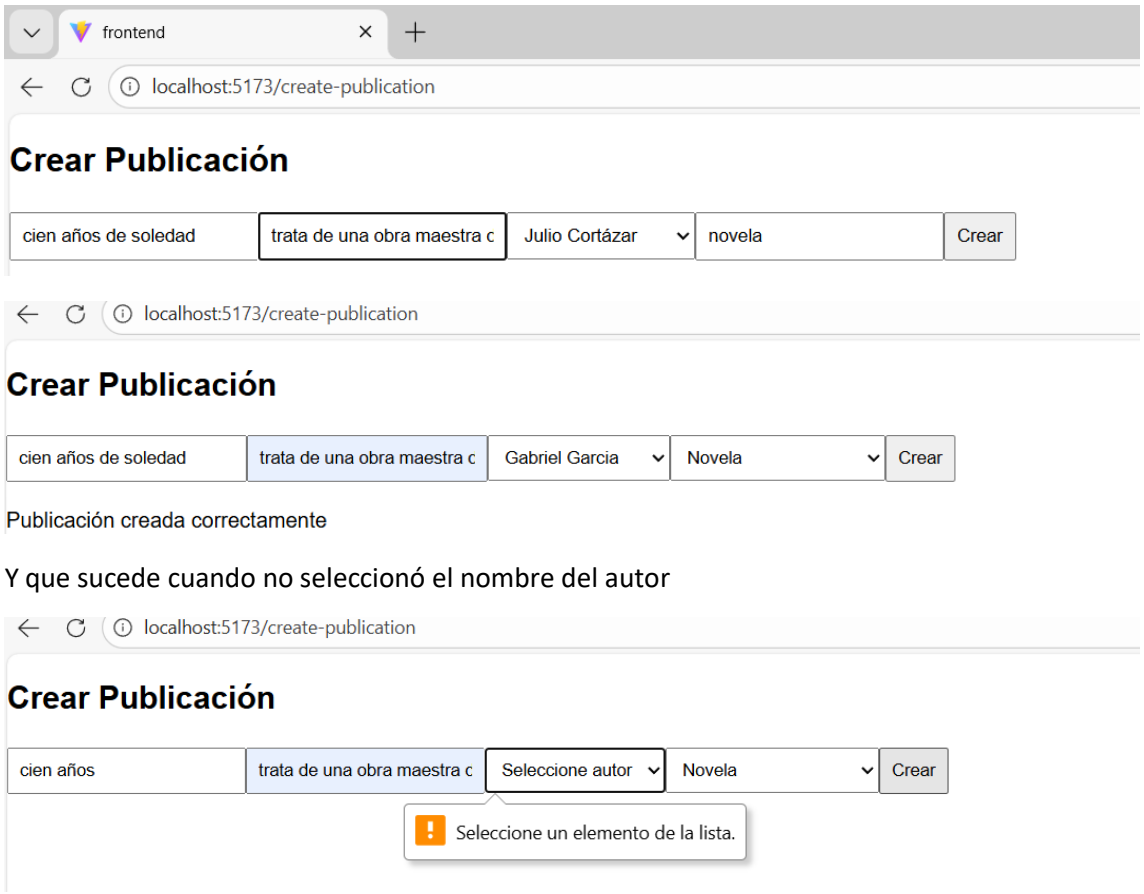
Verificamos nuestra API de Publications

publications.api.ts

frontend > src > api > publications.api.ts > PublicationsAPI

```ts
import { http } from "./http";

const PUBLICATIONS_URL = "http://localhost:3002/publications";

export const PublicationsAPI = {
  getAll: async () => {
    const response = await http.get(PUBLICATIONS_URL);
    return response.data;
  },

  getById: async (id: number) => {
    const response = await http.get(`${PUBLICATIONS_URL}/${id}`);
    return response.data;
  },

  create: async (publication: {
    title: string;
    content: string;
    authorId: number;
    category?: string;
  }) => {
    const response = await http.post(PUBLICATIONS_URL, publication);
    return response.data;
  },
};
```

Probamos en el navegador: localhost:5173/create-publication



Y que sucede cuando no seleccionó el nombre del autor

Listar publicaciones en el frontend

Crear vista para listar publicaciones

```
15    return (
16      <div>
17        <h2>Listado de Publicaciones</h2>
18
19        {error && <p>{error}</p>}
20
21        <table border={1} cellPadding={8}>
22          <thead>
23            <tr>
24              <th>ID</th>
25              <th>Título</th>
26              <th>Categoría</th>
27              <th>Autor ID</th>
28            </tr>
29          </thead>
30          <tbody>
31            {publications.map((pub) => (
32              <tr key={pub.id}>
33                <td>{pub.id}</td>
34                <td>{pub.title}</td>
35                <td>{pub.category}</td>
36                <td>{pub.authorId}</td>
37              </tr>
38            ))}
39          </tbody>
40        </table>
41      </div>
42    );
```

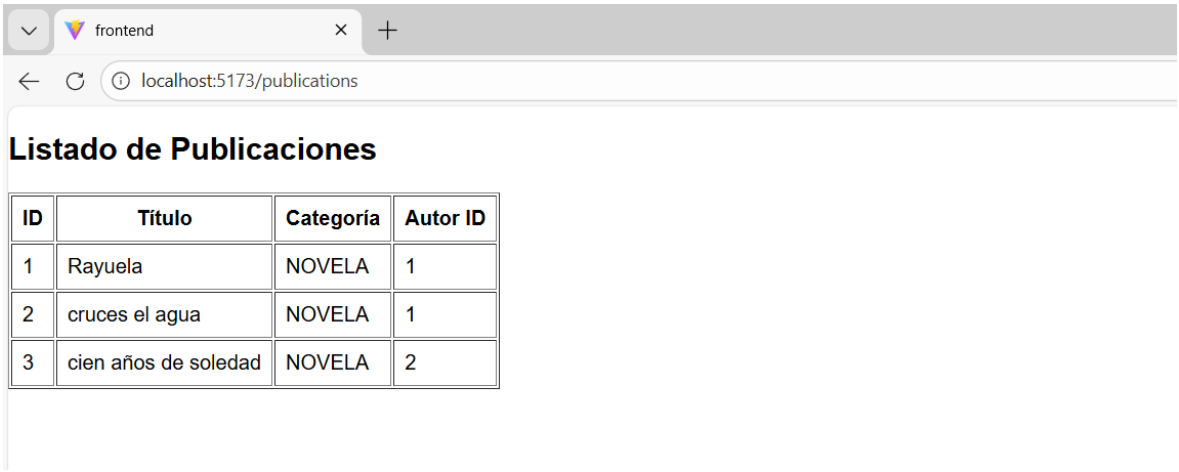
Registrar la ruta en App.ts

```
5   import PublicationsPage from "./pages/PublicationsPage";
```

```
14          <Route path="/publications" element={<PublicationsPage />} />
```

Evidenciamos en el navegador

http://localhost:5173/publications



Agregamos un Link de react-router-dom

```jsx
<nav style={{ marginBottom: "20px" }}>
  <Link to="/">Autores</Link> |{" "}
  <Link to="/publications">Publicaciones</Link> |{" "}
  <Link to="/create-author">Crear Autor</Link> |{" "}
  <Link to="/create-publication">Crear Publicación</Link>
</nav>
```

Lo que hicimos fue agregar en la parte de arriba, fuera de <Routes> la navegación hacia las rutas indicadas.

Evidenciamos nuestro listar autores con los links hacia las demás rutas

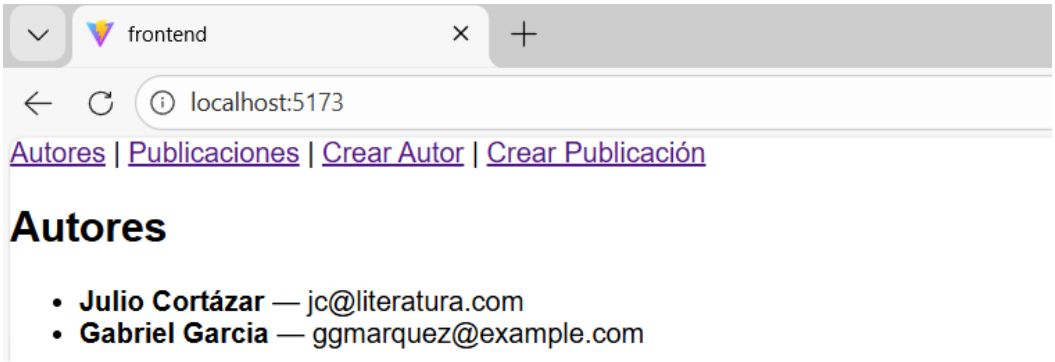
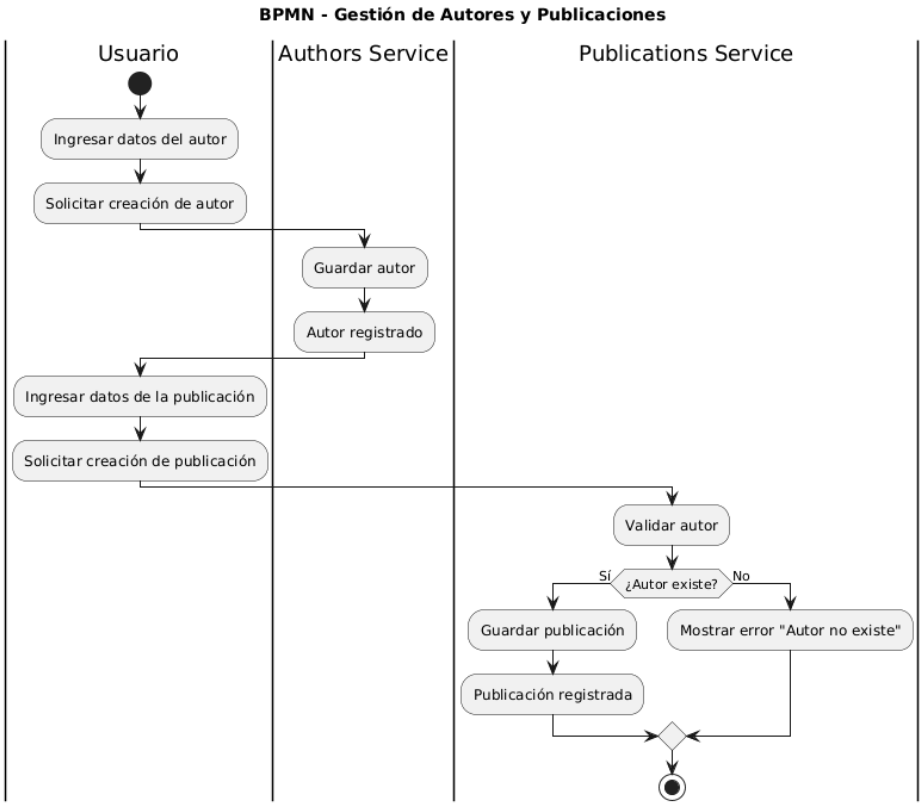
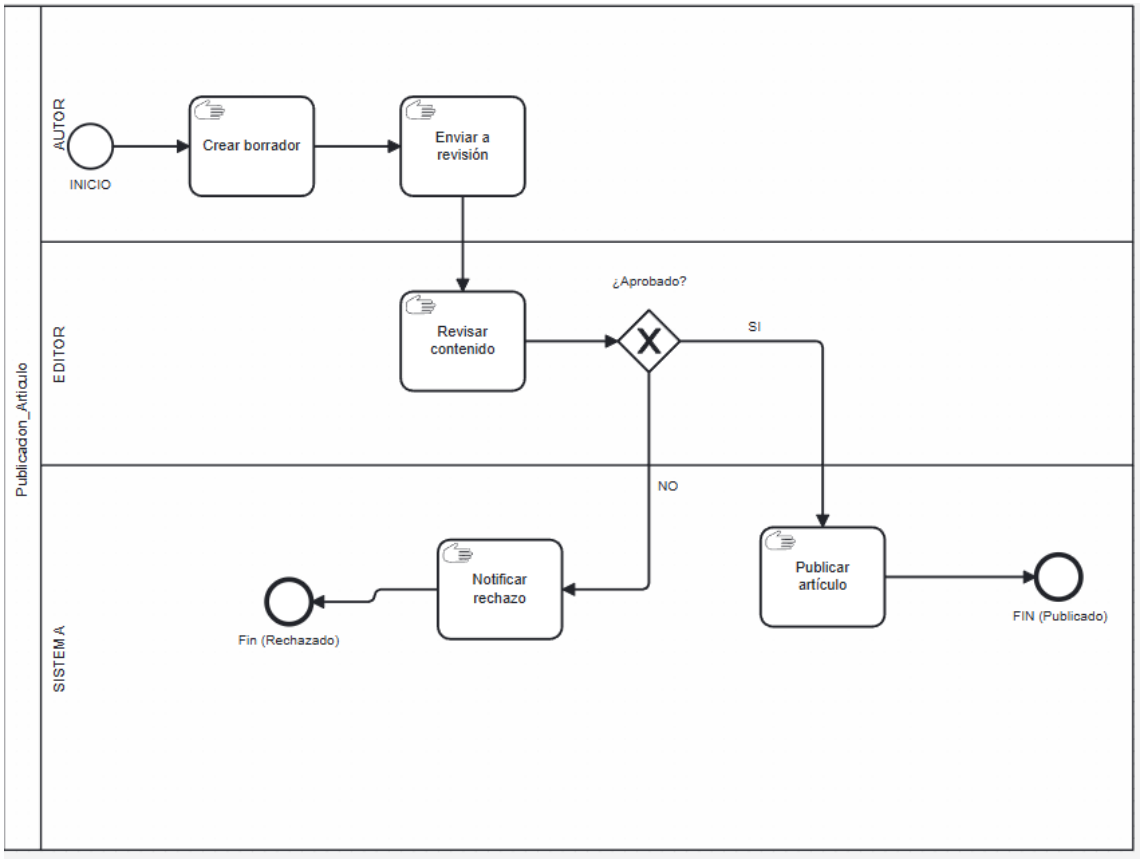


Diagrama BPMN



Modelado BPMN en Camunda

Este modelo permite representar el proceso del microservicion donde tenemos un artículo desde su creación hasta su publicación o rechazo. El proceso inicia cuando se crea un borrador del artículo, el cual es enviado a revisión y luego, un revisor analiza el contenido del artículo y toma una decisión mediante un gateway de decisión, donde se evalúa si el artículo es aprobado o rechazado. Si el artículo es aprobado, se procede a preparar y publicar el contenido, finalizando el proceso con éxito.

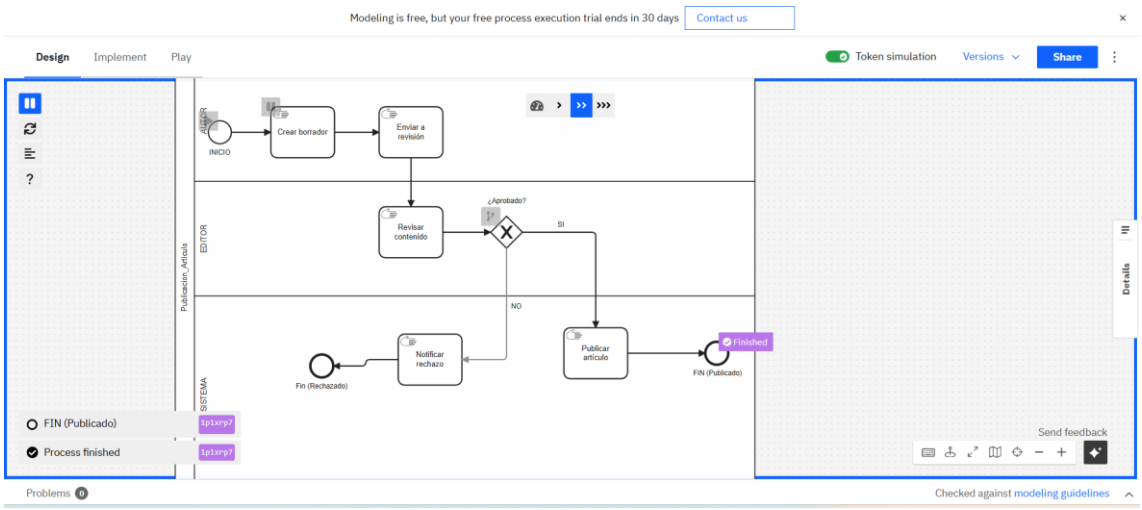


Este modelo nos ayuda a visualizar las actividades, decisiones y resultados del proceso. Además, mediante la **simulación de tokens**, es posible validar el flujo del proceso y comprobar que las decisiones se ejecutan de forma correcta o no, asegurando que el proceso está bien.

Decisión de aprobación (Gateway exclusivo XOR):

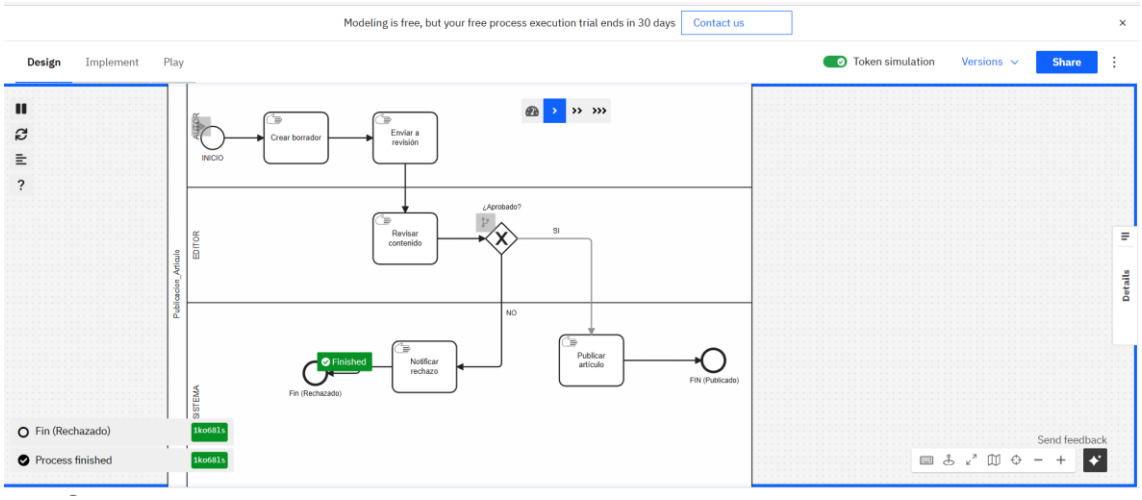
Si aprueba: preparar publicación → publicar → fin (publicado)

Cuando el contenido es correcto (aprobado), el flujo continúa su camino hacia la actividad preparar publicación, seguida de la tarea publicar artículo, y finalmente el proceso en el evento da por finalizado.



Si rechaza: notificar rechazo → fin (rechazado)

A su vez, cuando el contenido es rechazado, el flujo se dirige a la tarea notificar rechazo, concluyendo el proceso en el evento con un mensaje que dice; fin rechazado.



El modelo BPMN representa el proceso de negocio del sistema editorial, desde la creación de un artículo por parte del autor hasta su publicación o rechazo. Aunque la arquitectura del sistema está implementada mediante microservicios (Authors y Publications), el BPMN describe el flujo funcional global del dominio editorial, no un microservicio específico.

CONCLUSIONES

- La arquitectura está basada en microservicios que permitieron desacoplar las funcionalidades del sistema editorial, separando claramente la gestión de autores y publicaciones.
- El uso de Docker y Docker Compose simplificó el despliegue y la ejecución del sistema, permitiendo levantar todos los servicios y bases de datos de forma consistente en distintos entornos.

- La modelación del proceso mediante BPMN y su simulación con Token Simulation permitió visualizar y validar el flujo de negocio de publicación de artículos, asegurando que las decisiones de aprobación y rechazo se ejecuten correctamente antes de una implementación real.

RECOMENDACIONES

- Se recomienda más pruebas automatizadas para cada microservicio, con el fin de detectar errores de forma temprana y asegurar la estabilidad del sistema ante futuros cambios.
- También para una versión productiva del sistema, sería conveniente integrar un API Gateway que centralice el acceso a los microservicios, mejore la seguridad, gestione autenticación y permita un mejor control del tráfico entre cliente y servicios.
- Por último, se recomienda extender el modelo BPMN incorporando métricas de tiempo y carga en la simulación, lo que permitiría analizar cuellos de botella y optimizar el proceso editorial antes de su despliegue en un entorno real.