

UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Tecnologías de la Información y la Comunicación



Tema

Actividad 3

ESTUDIANTE

Tenemaza Parra Alanis Valeria

Asignatura

Arquitectura de software

Período

Octubre – Marzo 2025-2026

2. Arquitectura del sistema

2.1 Descripción general

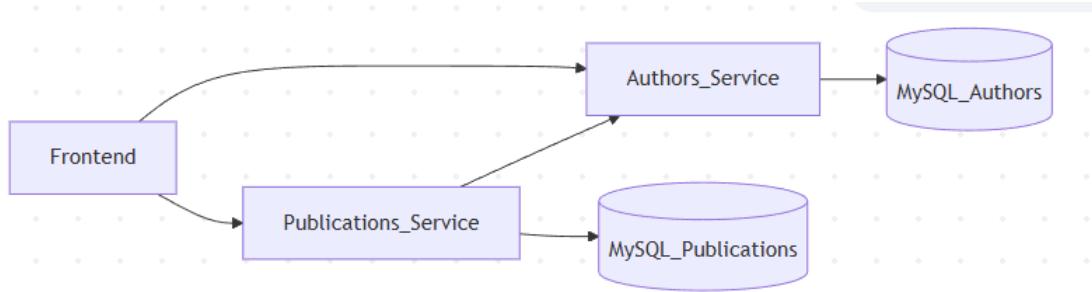
La solución implementada corresponde a un sistema editorial moderno basado en una **arquitectura de microservicios**, cuyo objetivo es administrar autores y publicaciones de forma desacoplada, escalable y mantenable.

El sistema está compuesto por:

- **Microservicio Authors**: encargado de la gestión del ciclo de vida de los autores.
- **Microservicio Publications**: encargado de la gestión de publicaciones y su estado editorial.
- **Frontend web**: interfaz gráfica para la interacción del usuario.
- **Bases de datos independientes MySQL**, una por cada microservicio.
- **Docker Compose** para el despliegue y orquestación de todos los componentes.

La comunicación entre microservicios se realiza mediante **HTTP REST**, evitando dependencias directas a nivel de base de datos.

2.2 Diagrama de arquitectura



Cada microservicio posee su propia base de datos, cumpliendo el principio de independencia y evitando acoplamiento fuerte.

3. Patrones de diseño aplicados

3.1 Repository Pattern

Dónde se aplicó:

En ambos microservicios, utilizando TypeORM.

Descripción:

El acceso a datos se realiza a través de repositorios (`Repository<Entity>`), evitando que la lógica de negocio acceda directamente a la base de datos.

Evidencia en código:

```

import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';
import { Person } from './person.entity';

@Entity()
export class Author extends Person {

    @PrimaryGeneratedColumn()
    declare id: number;

    @Column()
    declare name: string;

    @Column()
    declare email: string;

    @Column()
    declare nationality: string;
}

```

3.2

Adapter Pattern

Dónde se aplicó:

En el microservicio de Publications.

Descripción:

Se utiliza un cliente HTTP (axios) como adaptador para comunicarse con el microservicio Authors y validar la existencia de un autor antes de crear una publicación.

Evidencia en código:

```

try {
    await axios.get(`http://authors-api:3000/authors/${publication.authorId}`);
} catch (error) {
    throw new BadRequestException('El autor no existe');
}

return this.repo.save(publication);
}

```

3.3 Strategy Pattern

Dónde se aplicó:

En la gestión de estados editoriales de una publicación.

Descripción:

Los estados (DRAFT, REVIEW, APPROVED, PUBLISHED, REJECTED) permiten aplicar diferentes comportamientos según el estado editorial, facilitando la extensión futura del sistema.

Evidencia en código:

```

@Controller('authors')
export class AuthorsController {
  constructor(private readonly service: AuthorsService) {}

  @Post()
  create(@Body() author: Author) {
    return this.service.create(author);
  }

  > @Get(':id')
  findOne(@Param('id') id: number) { ... }

  @Get()
  findAll() {
    return this.service.findAll();
  }
}

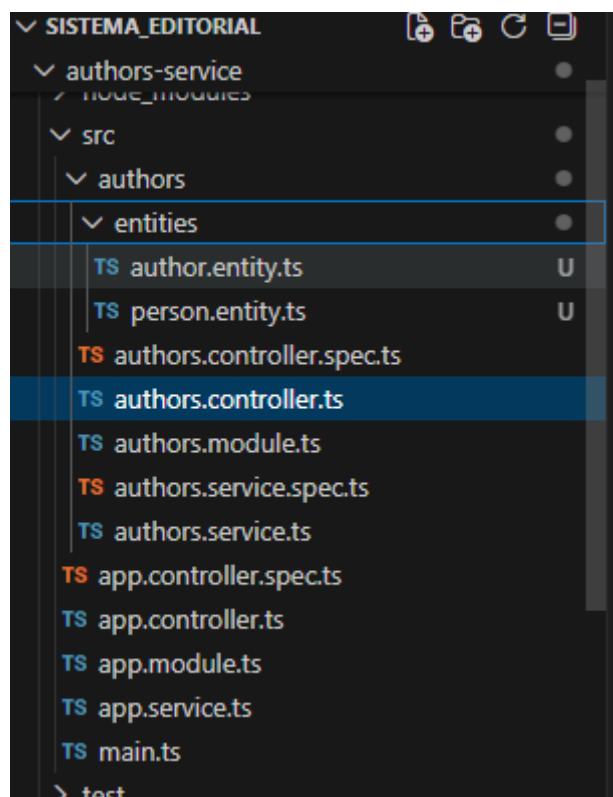
```

4. Evidencia de principios SOLID

4.1 Single Responsibility Principle (SRP)

Cada clase tiene una única responsabilidad:

- Controllers: manejo de HTTP
- Services: lógica de negocio
- Entities: modelo de datos



4.2 Open/Closed Principle (OCP)

El uso de clases abstractas permite extender el sistema sin modificar el código base.

Ejemplo:

```
export abstract class Content {  
    id: number;  
    title: string;  
}
```

4.3 Dependency Inversion Principle (DIP)

Los servicios dependen de abstracciones (repositorios) y no de implementaciones concretas.

5. Instrucciones de despliegue y ejecución

5.1 Requisitos previos

- Docker Desktop
- Node.js
- Docker Compose
- Visual Studio Code

5.2 Ejecución del sistema

Desde la raíz del proyecto:

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

Este comando levanta:

- Authors Service
- Publications Service
- Frontend
- MySQL Authors
- MySQL Publications

6. Endpoints principales

6.1 Authors Service

Crear autor

POST /authors

```
{  
  "name": "Juan Pérez",  
  "email": "juan@email.com"  
}
```

Obtener autor

GET /authors/1

6.2 Publications Service

Crear publicación

POST /publications

```
{  
  "title": "Microservicios con NestJS",  
  "authorId": 1  
}
```

Cambiar estado

PATCH /publications/1/status

```
{  
  "status": "PUBLISHED"  
}
```

7. Guía de ejecución

7.1 Comando principal

docker compose up –build

```

authors-service > 🐳 Dockerfile > ...
1   FROM node:18-alpine
2
3   WORKDIR /app
4
5   COPY package*.json ./
6   RUN npm install
7
8   COPY . .
9
10  RUN npm run build
11
12  EXPOSE 3000
13
14  CMD ["npm", "run", "start:dev"]
15

```

The screenshot shows a code editor interface with several tabs open:

- Dockerfile U**: Contains the Dockerfile code shown above.
- docker-compose.yml**: Shows a configuration for services like authors-service and publications-service.
- authors.module.ts**, **authors.service.ts**, and **authors.controller.ts**: These files are part of a Nest.js application.
- PROBLEMS**: A panel showing no errors or warnings.
- OUTPUT**: A panel showing the command `docker-compose up --build` being run in the terminal.
- DEBUG CONSOLE**: A panel showing the results of the build command, including layer statistics and container creation details.
- TERMINAL**: The command `PS C:\Sistema_editorial\authors-service> docker-compose up --build` is visible.
- PORTS**: A panel showing port mappings for the services.
- EXPLORE**: A sidebar showing the project structure, including `SISTEMA EDITORIAL` and `src` directories containing various files like `app.controller.specs`, `app.controller.ts`, etc.
- OUTLINE**: A panel showing the outline of the current file.

7.2 URLs del sistema

- **Frontend:**
<http://localhost:3000>
- **Authors API:**
<http://authors-api:3000/authors>
- **Publications API:**
<http://publications-api:3000/authors>

8. Conclusiones

El proyecto permitió aplicar conceptos fundamentales de arquitectura de microservicios, principios SOLID y patrones de diseño, además de herramientas modernas como Docker y NestJS.

La solución desarrollada es escalable, mantenible y cumple con los requisitos funcionales y técnicos solicitados.