Red Social con Microservicios

Documentación de Instalación y Uso

Tecnologías: Node.js + TypeScript + React + PostgreSQL

Fecha: 27 de enero de 2026

Autor: Marcel Diaz Granados Robayo

Tabla de Contenidos

1. Descripción del Proyecto

Este proyecto es una red social desarrollada con arquitectura de microservicios que permite a los usuarios:

• Autenticarse con usuario y contraseña usando JWT

• Visualizar publicaciones de otros usuarios en tiempo real

• Crear publicaciones con mensaje y fecha automática

La aplicación está completamente dockerizada y utiliza las mejores prácticas de desarrollo Full Stack, incluyendo TypeScript tanto en el frontend como en el backend, documentación Swagger para las APIs, y un sistema de gestión de estado moderno con Zustand.

Características Principales:

✓ Arquitectura de microservicios escalable

✓ TypeScript en Frontend y Backend

✓ Autenticación JWT segura

✓ Documentación Swagger interactiva

✓ Dockerización completa

✓ ORM Prisma para PostgreSQL

✓ Manejo de estado con Zustand

✓ Seeders automáticos de datos de prueba

2. Arquitectura del Sistema

El sistema está diseñado con una arquitectura de microservicios que se comunican entre sí:

Frontend (React + TypeScript)

• Puerto: Puerto 3000

• Interfaz de usuario moderna y responsiva con gestión de estado Zustand y comunicación con APIs via Axios

Auth Service (Node.js + Express + TypeScript)

• Puerto: Puerto 3001

• Maneja autenticación de usuarios, genera tokens JWT y gestiona usuarios en PostgreSQL

Posts Service (Node.js + Express + TypeScript)

• Puerto: Puerto 3002

• Gestiona las publicaciones, valida tokens JWT y proporciona CRUD completo de posts

PostgreSQL Database

• Puerto: Puerto 5432

• Base de datos relacional con acceso mediante Prisma ORM y migraciones automáticas

3. Tecnologías Utilizadas

Categoría	Tecnologías
Backend	**Node.js, prisma, swagger**
Frontend	**React, typeScript, css3, Zustand**
Base de datos	**PostgreSLQ 15**
DevOps	**Docker Compose**

4. Requisitos Previos

Para ejecutar con Docker (Recomendado):

- Docker versión 20.10 o superior
- Docker Compose versión 2.0 o superior
- 4GB de RAM disponible
- 2GB de espacio en disco

Para desarrollo local:

- Node.js versión 18 o superior
- npm o yarn
- PostgreSQL 15
- Git

5. Instalación con Docker

Paso 1: Clonar el repositorio

```
git clone
<https://github.com/marceld
gr4/Desarrollo-de-Red-
Social-.git>

cd social-network-fullstack
```

Paso 2: Levantar todos los servicios

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

Este comando realiza automáticamente:

- Construye las imágenes Docker de cada servicio
- Inicia PostgreSQL en el puerto 5432
- Ejecuta las migraciones de Prisma
- Crea 5 usuarios de prueba con sus publicaciones
- Levanta Auth Service en el puerto 3001

• Levanta Posts Service en el puerto 3002

• Inicia el Frontend en el puerto 3000

Paso 3: Verificar que todo esté funcionando

Abrir el navegador y acceder a:

• Frontend: http://localhost:3000

• Auth Service: http://localhost:3001/health

• Posts Service: http://localhost:3002/health

• Swagger Auth: http://localhost:3001/api-docs

• Swagger Posts: http://localhost:3002/api-docs

Comandos útiles de Docker:

Ver logs de todos los servicios:

```
docker-compose logs -f
```

Ver logs de un servicio específico:

```
docker-compose logs -f auth-service
```

Detener todos los servicios:

```
docker-compose down
```

Limpiar todo (incluyendo volúmenes):

```
docker-compose down -v
```

Reconstruir un servicio específico:

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

6. Instalación Manual (Desarrollo)

PostgreSQL:

```
createdb social_network
```

Auth Service:

- `cd backend/auth-service npm install`
- `npm run prisma:generate`
- `npm run prisma:migrate npm run seed npm run dev`

Posts Service:

- `cd backend/posts-service`
- `npm install`
- `npm run prisma:generate`
- `npm run prisma:migrate`
- `npm run dev`

Frontend:

```
cd frontend npm install npm start
```

7. Uso de la Aplicación

Credenciales de Prueba:

Usuario	Contraseña	Email
user1	password123	user1@example.com
user2	password123	user2@example.com
user3	password123	user3@example.com

Flujo de Uso:

1. Login

Acceder a http://localhost:3000, ingresar usuario y contraseña de prueba, presionar botón 'Login'

2. Visualizar Publicaciones

Automáticamente se carga el feed con todas las publicaciones ordenadas de más reciente a más antigua

3. Crear Publicación

En la sección 'Create Post' escribir un mensaje (máximo 500 caracteres) y presionar botón 'Post'. La publicación aparece inmediatamente en el feed

4. Cerrar Sesión

Presionar botón 'Logout' en el header. El token JWT es eliminado del localStorage

8. Endpoints de la API

Auth Service - POST /api/auth/login

Descripción: Autentica un usuario y retorna un token JWT

URL: http://localhost:3001/api/auth/login

Método: POST

Request Body:

```
{ "username": "user1", "password": "password123" }
```

Response (200 OK):

```
{ "message": "Login successful",

 "token": "eyJhbGciOiJIUzI1NiIs...",

"user":
{ "id": 1, "username": "user1", "email": "user1@example.com" } }
```

Posts Service - GET /api/posts

Descripción: Obtiene todas las publicaciones

URL: http://localhost:3002/api/posts

Método: GET

Headers: Authorization: Bearer <token>

Response (200 OK):

```
[ { "id": 1, "message": "Hello! This is my first post...", "createdAt":
"2024-01-26T10:00:00.000Z", "user": { "id": 1, "username": "user1", "email":
"user1@example.com" } } ]
```

Posts Service - POST /api/posts

Descripción: Crea una nueva publicación

URL: http://localhost:3002/api/posts

Método: POST

Headers: Authorization: Bearer <token>

Request Body:

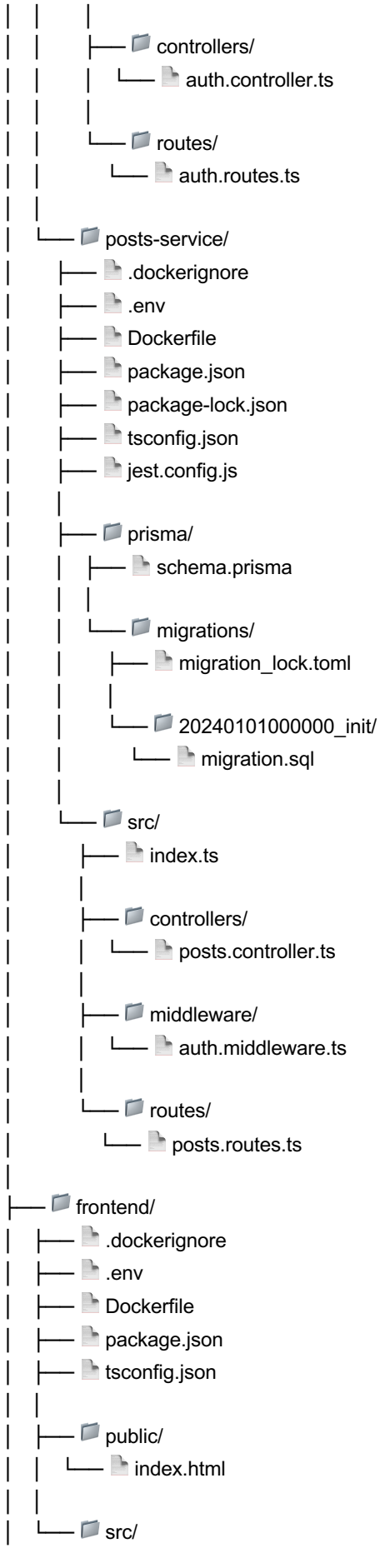
```
{ "message": "This is my new post!" }
```

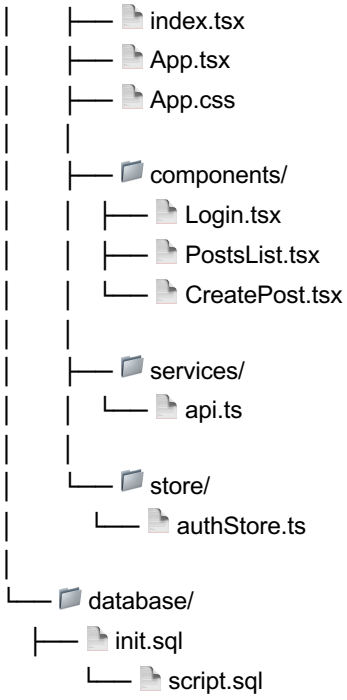
Response (201 Created):

```
{ "message": "Post created successfully", "post": { "id": 6, "message": "This
is my new post!", "userId": 1, "createdAt": "2024-01-26T10:30:00.000Z",
"user": { ... } } }
```

9. Estructura del Proyecto

```
social-network-fullstack/
│
├── README.md
├── INSTALACION_Y_DOCUMENTACION.pdf
├── docker-compose.yml
├── start.sh
├── .gitattributes
│
├── backend/
│   │
│   ├── auth-service/
│   │   ├── .dockerignore
│   │   ├── .env
│   │   ├── Dockerfile
│   │   ├── package.json
│   │   ├── package-lock.json
│   │   ├── tsconfig.json
│   │   ├── jest.config.js
│   │   │
│   │   ├── prisma/
│   │   │   ├── schema.prisma
│   │   │   │
│   │   │   └── migrations/
│   │   │       ├── migration_lock.toml
│   │   │       │
│   │   │       └── 20260126230505_init/
│   │   │           └── migration.sql
│   │   │
│   │   └── src/
│   │       ├── index.ts
│   │       ├── seed.ts
```


controllers/
auth.controller.ts
routes/
auth.routes.ts
posts-service/
.dockerignore
.env
Dockerfile
package.json
package-lock.json
tsconfig.json
jest.config.js
prisma/
schema.prisma
migrations/
migration_lock.toml
20240101000000_init/
migration.sql
src/
index.ts
controllers/
posts.controller.ts
middleware/
auth.middleware.ts
routes/
posts.routes.ts
frontend/
.dockerignore
.env
Dockerfile
package.json
tsconfig.json
public/
index.html
src/



10. Pruebas y Testing

El proyecto está configurado para ejecutar pruebas unitarias con Jest.

Ejecutar pruebas:

Auth Service:

```
cd backend/auth-service npm test
```

Posts Service:

```
cd backend/posts-service npm test
```

Frontend:

```
cd frontend npm test
```

Áreas cubiertas por tests:

• Autenticación de usuarios
• Validación de tokens JWT
• Creación de publicaciones
• Validación de datos de entrada
• Manejo de errores

11. Solución de Problemas

Problema	Solución
Puerto ya en uso	Detener el proceso que usa el puerto o cambiar el puerto en docker-compose.yml
Error de conexión a DB	Verificar que PostgreSQL esté corriendo: docker-compose ps
Migraciones fallidas	Eliminar volúmenes y recrear: docker-compose down -v && docker-compose up
Frontend no conecta	Verificar que las variables de entorno apunten a localhost:3001 y localhost:3002
Token inválido	Verificar que JWT_SECRET sea el mismo en ambos servicios
CORS error	Verificar que CORS esté habilitado en los servicios backend

Comandos de diagnóstico:

Ver estado de contenedores:

```
docker-compose ps
```

Ver logs detallados:

```
docker-compose logs -f --tail=100
```

Reiniciar un servicio específico:

```
docker-compose restart auth-service
```

Acceder a un contenedor:

```
docker-compose exec auth-service sh
```

Verificar conectividad a base de datos:

```
docker-compose exec postgres psql -U postgres -d social_network -c "\dt"
```

Conclusión

Este proyecto demuestra una implementación completa de una red social con arquitectura de microservicios, utilizando las mejores prácticas de desarrollo Full Stack moderno.

Puntos destacados:

• Arquitectura escalable y mantenible

• Código limpio y bien estructurado

• TypeScript para mayor seguridad de tipos

• Documentación completa con Swagger

• Dockerización para fácil despliegue

• Seeders automáticos para pruebas rápidas

Para cualquier duda o problema, consultar el README.md del proyecto o revisar la documentación de Swagger en los endpoints correspondientes.

Enlaces útiles:

• Swagger Auth Service: http://localhost:3001/api-docs

• Swagger Posts Service: http://localhost:3002/api-docs

• Aplicación Frontend: http://localhost:3000