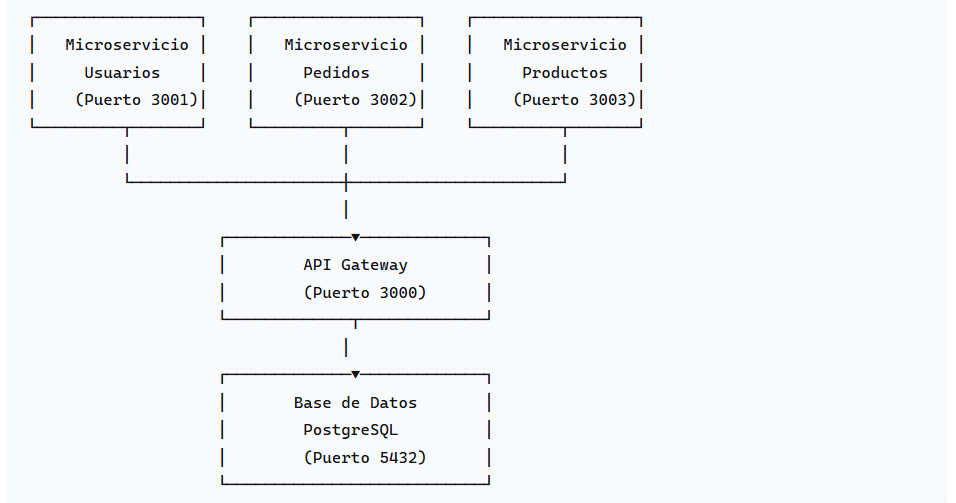
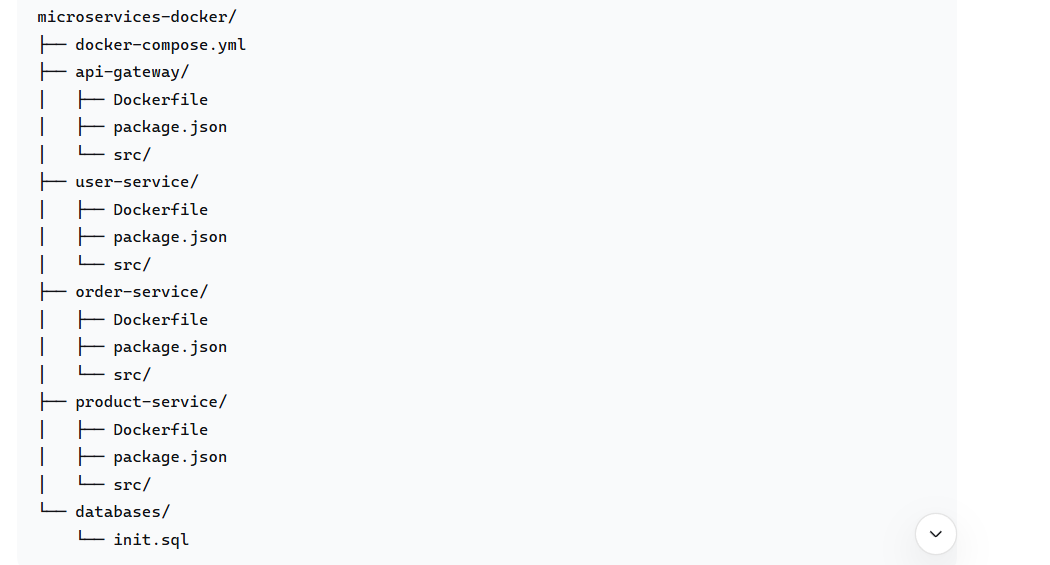
**Dockerización de Microservicios para Escalabilidad - Guía Técnica**

Basándome en los artículos proporcionados, te presento una guía completa para dockerizar microservicios y probar su comunicación con bases de datos.



**Procedimiento Paso a Paso**

**1. Estructura del Proyecto**



**2. Configuración de Docker Compose**

**docker-compose.yml :**version: '3.8'

services:

# Base de Datos PostgreSQL

postgres:

image: postgres:15

container\_name: microservices\_db

environment:

POSTGRES\_DB: microservices

POSTGRES\_USER: admin

POSTGRES\_PASSWORD: password

ports:

- "5432:5432"

volumes:

- postgres\_data:/var/lib/postgresql/data

- ./databases/init.sql:/docker-entrypoint-initdb.d/init.sql

networks:

- microservices\_network

# Microservicio de Usuarios

user-service:

build: ./user-service

container\_name: user\_service

environment:

DB\_HOST: postgres

DB\_PORT: 5432

DB\_NAME: microservices

DB\_USER: admin

DB\_PASSWORD: password

PORT: 3001

ports:

- "3001:3001"

depends\_on:

- postgres

networks:

- microservices\_network

restart: unless-stopped

# Microservicio de Pedidos

order-service:

build: ./order-service

container\_name: order\_service

environment:

DB\_HOST: postgres

DB\_PORT: 5432

DB\_NAME: microservices

DB\_USER: admin

DB\_PASSWORD: password

PORT: 3002

ports:

- "3002:3002"

depends\_on:

- postgres

networks:

- microservices\_network

restart: unless-stopped

# Microservicio de Productos

product-service:

build: ./product-service

container\_name: product\_service

environment:

DB\_HOST: postgres

DB\_PORT: 5432

DB\_NAME: microservices

DB\_USER: admin

DB\_PASSWORD: password

PORT: 3003

ports:

- "3003:3003"

depends\_on:

- postgres

networks:

- microservices\_network

restart: unless-stopped

# API Gateway

api-gateway:

build: ./api-gateway

container\_name: api\_gateway

environment:

USER\_SERVICE\_URL: http://user-service:3001

ORDER\_SERVICE\_URL: http://order-service:3002

PRODUCT\_SERVICE\_URL: http://product-service:3003

PORT: 3000

ports:

- "3000:3000"

depends\_on:

- user-service

- order-service

- product-service

networks:

- microservices\_network

restart: unless-stopped

volumes:

postgres\_data:

networks:

microservices\_network:

driver: bridge

**3. Dockerfile para Microservicios (Ejemplo User Service)**

**user-service/Dockerfile :**

# Usar imagen base de Node.js

FROM node:18-alpine

# Establecer directorio de trabajo

WORKDIR /app

# Copiar package.json y package-lock.json

COPY package\*.json ./

# Instalar dependencias

RUN npm ci --only=production

# Copiar código fuente

COPY . .

# Exponer puerto

EXPOSE 3001

# Variable de entorno para el puerto

ENV PORT=3001

# Comando para iniciar la aplicación

CMD ["node", "src/server.js"]

**4. Script de Inicialización de Base de Datos**

**databases/init.sql**

-- Crear esquemas separados para cada microservicio

CREATE SCHEMA IF NOT EXISTS users;

CREATE SCHEMA IF NOT EXISTS orders;

CREATE SCHEMA IF NOT EXISTS products;

-- Tabla para usuarios

CREATE TABLE IF NOT EXISTS users.users (

id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Tabla para productos

CREATE TABLE IF NOT EXISTS products.products (

id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

price DECIMAL(10,2) NOT NULL,

stock INTEGER NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Tabla para pedidos

CREATE TABLE IF NOT EXISTS orders.orders (

id SERIAL PRIMARY KEY,

user\_id INTEGER NOT NULL,

product\_id INTEGER NOT NULL,

quantity INTEGER NOT NULL,

total\_price DECIMAL(10,2) NOT NULL,

status VARCHAR(50) DEFAULT 'pending',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Datos de ejemplo

INSERT INTO users.users (name, email) VALUES

('Juan Pérez', 'juan@email.com'),

('María García', 'maria@email.com');

INSERT INTO products.products (name, price, stock) VALUES

('Laptop', 999.99, 10),

('Mouse', 29.99, 50);

**5. Código de Ejemplo para Microservicio**

**user-service/src/server.js**

const express = require('express');

const { Pool } = require('pg');

const app = express();

const port = process.env.PORT || 3001;

// Configuración de la base de datos

const pool = new Pool({

host: process.env.DB\_HOST,

port: process.env.DB\_PORT,

database: process.env.DB\_NAME,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

});

app.use(express.json());

// Health check

app.get('/health', (req, res) => {

res.status(200).json({ status: 'OK', service: 'user-service' });

});

// Obtener todos los usuarios

app.get('/users', async (req, res) => {

try {

const result = await pool.query('SELECT \* FROM users.users');

res.json(result.rows);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

// Crear usuario

app.post('/users', async (req, res) => {

try {

const { name, email } = req.body;

const result = await pool.query(

'INSERT INTO users.users (name, email) VALUES ($1, $2) RETURNING \*',

[name, email]

);

res.status(201).json(result.rows[0]);

} catch (error) {

res.status(500).json({ error: error.message });

}

});

app.listen(port, () => {

console.log(`User service running on port ${port}`);

});

**6. API Gateway Básico**

**api-gateway/src/server.js**

const express = require('express');

const axios = require('axios');

const app = express();

const port = process.env.PORT || 3000;

app.use(express.json());

// Proxy para el servicio de usuarios

app.use('/users', async (req, res) => {

try {

const response = await axios({

method: req.method,

url: `${process.env.USER\_SERVICE\_URL}${req.path}`,

data: req.body,

headers: { 'Content-Type': 'application/json' }

});

res.status(response.status).json(response.data);

} catch (error) {

res.status(500).json({ error: 'Error communicating with user service' });

}

});

// Health check general

app.get('/health', async (req, res) => {

try {

const services = [

{ name: 'user-service', url: process.env.USER\_SERVICE\_URL },

{ name: 'order-service', url: process.env.ORDER\_SERVICE\_URL },

{ name: 'product-service', url: process.env.PRODUCT\_SERVICE\_URL }

];

const healthChecks = await Promise.all(

services.map(async (service) => {

try {

const response = await axios.get(`${service.url}/health`);

return { service: service.name, status: response.data.status };

} catch (error) {

return { service: service.name, status: 'DOWN' };

}

})

);

res.json({ status: 'OK', services: healthChecks });

} catch (error) {

res.status(500).json({ error: error.message });

}

});

app.listen(port, () => {

console.log(`API Gateway running on port ${port}`);

});

**Procedimiento de Pruebas**

**1. Construir y Ejecutar los Contenedores**

# Clonar o crear la estructura de directorios

mkdir microservices-docker

cd microservices-docker

# Construir y levantar todos los servicios

docker-compose up –build  
  
  
**2. Verificar el Estado de los Servicios**

# Ver contenedores en ejecución

docker-compose ps

# Ver logs de todos los servicios

docker-compose logs -f

# Ver logs de un servicio específico

docker-compose logs user-service

**3. Probar Comunicación con la Base de Datos**

# Conectarse a la base de datos PostgreSQL

docker exec -it microservices\_db psql -U admin -d microservices

# Ejecutar consultas de prueba

SELECT \* FROM users.users;

SELECT schema\_name FROM information\_schema.schemata;

**4. Probar los Endpoints de los Microservicios**

# Health check del API Gateway

curl http://localhost:3000/health

# Probar servicio de usuarios

curl http://localhost:3000/users

curl -X POST http://localhost:3000/users \

-H "Content-Type: application/json" \

-d '{"name":"Test User","email":"test@email.com"}'

# Probar servicios individuales directamente

curl http://localhost:3001/health # User service

curl http://localhost:3002/health # Order service

curl http://localhost:3003/health # Product service

**5. Monitorear la Comunicación**

# Ver tráfico de red entre contenedores

docker network inspect microservices-docker\_microservices\_network

# Estadísticas de uso de recursos

docker stats

**Mejores Prácticas Implementadas**

**1. Seguridad**

* Variables de entorno para credenciales
* Imágenes Alpine para reducir superficie de ataque
* Usuarios no-root en contenedores

**2. Performance**

* Multi-stage builds (para aplicaciones más complejas)
* Cache de dependencias de npm
* Volúmenes para persistencia de datos

**3. Escalabilidad**

* Servicios independientes
* Balance de carga nativo de Docker
* Health checks integrados

**📊 Comandos Útiles para Debugging**

bash

*# Inspeccionar un contenedor específico*

docker exec -it user\_service sh

*# Ver variables de entorno de un contenedor*

docker exec user\_service env

*# Probar conectividad entre servicios*

docker exec user\_service ping order-service

*# Backup de base de datos*

docker exec microservices\_db pg\_dump -U admin microservices > backup.sql

**🎯 Resultados Esperados**

Al seguir este procedimiento, deberías obtener:

1. **4 contenedores Docker ejecutándose:**
   * PostgreSQL database
   * User service (puerto 3001)
   * Order service (puerto 3002)
   * Product service (puerto 3003)
   * API Gateway (puerto 3000)
2. **Comunicación exitosa entre:**
   * Microservicios ↔ Base de datos
   * API Gateway ↔ Microservicios
   * Microservicios entre sí
3. **Persistencia de datos** en la base de datos PostgreSQL
4. **Capacidad de escalar** individualmente cada microservicio