crea una practica de programación que se ejecuten en local para luego subir a un servidor sobre debian 12 para programarlo necesito implementar una arquitectura de microservicios. Publicarlos con minikube y usar kafka para establecer colas de mensages entre ellos. necesiato aprender la arquitectura de los microservicios .   
El ejercicio es:   
Tengo varias cámaras espcam32 que accediendo por el navegador a la ip y un endpoint puedo visualizar un archivo jpeg o jjpeg (streaming).   
debo programar un sistema usando arquitectura de microservicios que verifique que la camara puede que este o no encendida   
mediante un front el usuario entrar en el sistema logueandose   
entonces vera todas las camaras y si estan o no encendidas.   
la camara puede estar encendiéndose o apagandose debe ser en tiempo real saber si la cámara esta encendia po apagada   
si el usuario quiete ver el streaming de la cámara debe susbcrivirse mediante un botón  
se debe de grabar en una base de datos cada vez que un usuario se subscribe a un streaming y el tiempo que permanece conectado

# Desarrollo en Windows 10 para Arquitectura de Microservicios con Minikube y Kafka

## 1. Configuración Inicial en Windows 10

### Requisitos previos:

1. **Windows 10 Pro, Enterprise o Education** (Home requiere trabajo adicional)
2. **Habilitar Hyper-V** (o usar Docker Desktop con WSL2)
3. **Memoria suficiente** (recomendado mínimo 8GB RAM)

### Pasos de configuración:

powershell

# 1. Habilitar Hyper-V (requiere reinicio)

Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All

# 2. Instalar Docker Desktop

# Descargar desde: https://www.docker.com/products/docker-desktop

# Habilitar WSL2 backend en configuración

# 3. Instalar Windows Subsystem for Linux (WSL2)

wsl --install

wsl --set-default-version 2

# 4. Instalar Minikube para Windows

choco install minikube -y # Usando Chocolatey

# o descargar manualmente desde https://minikube.sigs.k8s.io/docs/start/

# 5. Instalar kubectl

choco install kubernetes-cli -y

# 6. Instalar Helm

choco install kubernetes-helm -y

## 2. Iniciar Minikube en Windows

powershell

# Iniciar Minikube con el driver de Docker (requiere Docker Desktop corriendo)

minikube start --driver=docker --memory=4096 --cpus=2

# Verificar estado

minikube status

# Configurar el contexto de kubectl

kubectl config use-context minikube

## 3. Instalar Kafka en Minikube (Windows)

powershell

# Añadir repositorio Helm

helm repo add bitnami https://charts.bitnami.com/bitnami

helm repo update

# Instalar Kafka (igual que en Linux)

helm install kafka bitnami/kafka

# Verificar instalación

kubectl get pods -w

## 4. Configuración Específica para la Cámara SPcam32 en Windows

1. **Controladores**: Asegurarse de tener los drivers de la cámara para Windows
2. **OpenCV en Windows**:

powershell

pip install opencv-python

pip install opencv-contrib-python

1. **Acceso a cámaras RTSP**: Puede requerir ajustes adicionales en Windows

## 5. Comandos Útiles Específicos para Windows

powershell

# Acceder al dashboard de Minikube

minikube dashboard

# Abrir tunnel para servicios (si usas LoadBalancer)

minikube tunnel

# Conectar a un pod específico

kubectl exec -it <pod-name> -- /bin/bash

# Ver logs de un pod

kubectl logs -f <pod-name>

# Lista de Microservicios Necesarios

## 1. Servicio de Autenticación (Java Spring Boot)

**Responsabilidad**: Manejar login, JWT, validación de sesiones

**Endpoints**:

/auth/login (POST)

/auth/validate (GET)

**Base de datos**: PostgreSQL para usuarios/credenciales

## 2. Servicio de Gestión de Cámaras (Python FastAPI)

**Responsabilidad**: Verificar estado de cámaras, manejar streaming

**Endpoints**:

/cameras/status (GET) - Lista todas las cámaras y su estado

/cameras/{id}/stream (WebSocket) - Streaming en tiempo real

**Tecnologías especiales**: OpenCV, asyncio, WebSockets

## 3. Servicio de Monitoreo (Python)

**Responsabilidad**: Verificar constantemente estado de cámaras

**Funcionamiento**: Background worker que chequea cámaras periódicamente

**Comunicación**: Publica eventos a Kafka (cámara online/offline)

## 4. Servicio de Registro de Visualizaciones (Node.js)

**Responsabilidad**: Registrar conexiones de usuarios a streams

**Endpoints**:

/viewing/start (POST) - Inicia registro

/viewing/end (POST) - Finaliza registro

**Base de datos**: MongoDB para registros temporales

## 5. Servicio de Notificaciones (PHP Laravel)

**Responsabilidad**: Manejar alertas y notificaciones

**Comunicación**: Consume eventos de Kafka para notificar cambios de estado

## 6. API Gateway (Node.js Express)

**Responsabilidad**: Enrutamiento, rate limiting, autenticación centralizada

**Tecnologías**: Express Gateway o Kong

## 7. Frontend (React + TypeScript)

**Responsabilidad**: Interfaz de usuario

**Características**:

Listado de cámaras con estado en tiempo real

Conexión via WebSocket para streaming

Botones de suscripción/desuscripción

## Flujo de Comunicación

Copy

[Frontend] → [API Gateway] → [Autenticación]

↓

[Servicio Cámaras] ←→ [Kafka] ←→ [Monitoreo]

↓ ↓

[Registro Visualizaciones] [Notificaciones]

### Tecnologías Clave por Capa

| **Capa** | **Tecnologías Principales** |
| --- | --- |
| Frontend | React, TypeScript, Material-UI, WebSocket |
| Gateway | Node.js, Express Gateway |
| Autenticación | Java Spring Boot, JWT, PostgreSQL |
| Servicios de Cámara | Python, FastAPI, OpenCV, asyncio, WebSockets |
| Monitoreo | Python, Kafka Producer |
| Registros | Node.js, MongoDB |
| Notificaciones | PHP Laravel, Kafka Consumer |
| Infraestructura | Docker, Kubernetes (Minikube), Kafka, Helm |

### Estructura de Proyecto Recomendada

Copy

/proyecto-camaras

│

├── /frontend # Aplicación React

├── /api-gateway # Configuración del gateway

├── /auth-service # Servicio Java Spring Boot

├── /camera-service # Servicio Python FastAPI

├── /monitoring-service # Worker Python

├── /tracking-service # Servicio Node.js

├── /notification-service # Servicio PHP

├── /k8s # Manifiestos Kubernetes

├── /helm-charts # Charts Helm para cada servicio

└── docker-compose.yml # Para desarrollo local

Pasos Inmediatos para Comenzar

## Estructura de Directorios Completa para el Proyecto de Cámaras

Voy a crear una estructura detallada con los archivos esenciales para cada microservicio. Aquí está el árbol completo:

Copy

/proyecto-camaras

│

├── /frontend

│ ├── public/

│ │ ├── index.html

│ │ └── favicon.ico

│ ├── src/

│ │ ├── components/

│ │ │ ├── CameraList.tsx

│ │ │ ├── CameraView.tsx

│ │ │ └── LoginForm.tsx

│ │ ├── pages/

│ │ │ ├── Dashboard.tsx

│ │ │ └── Login.tsx

│ │ ├── services/

│ │ │ ├── auth.ts

│ │ │ └── camera.ts

│ │ ├── App.tsx

│ │ ├── index.tsx

│ │ └── types.ts

│ ├── package.json

│ ├── tsconfig.json

│ └── Dockerfile

│

├── /api-gateway

│ ├── config/

│ │ └── gateway.config.yml

│ ├── package.json

│ └── Dockerfile

│

├── /auth-service

│ ├── src/

│ │ ├── main/

│ │ │ ├── java/com/proyectocamaras/auth/

│ │ │ │ ├── config/

│ │ │ │ │ └── SecurityConfig.java

│ │ │ │ ├── controller/

│ │ │ │ │ └── AuthController.java

│ │ │ │ ├── model/

│ │ │ │ │ ├── User.java

│ │ │ │ │ └── Role.java

│ │ │ │ ├── repository/

│ │ │ │ │ └── UserRepository.java

│ │ │ │ ├── service/

│ │ │ │ │ └── UserDetailsServiceImpl.java

│ │ │ │ └── AuthApplication.java

│ │ │ └── resources/

│ │ │ ├── application.properties

│ │ │ └── data.sql

│ ├── pom.xml

│ └── Dockerfile

│

├── /camera-service

│ ├── app/

│ │ ├── core/

│ │ │ ├── camera\_manager.py

│ │ │ └── camera\_driver.py

│ │ ├── api/

│ │ │ ├── endpoints.py

│ │ │ └── websockets.py

│ │ ├── models/

│ │ │ └── camera.py

│ │ ├── main.py

│ │ └── config.py

│ ├── requirements.txt

│ ├── Dockerfile

│ └── .env.example

│

├── /monitoring-service

│ ├── app/

│ │ ├── services/

│ │ │ ├── camera\_checker.py

│ │ │ └── kafka\_producer.py

│ │ ├── models/

│ │ │ └── camera\_status.py

│ │ ├── main.py

│ │ └── config.py

│ ├── requirements.txt

│ ├── Dockerfile

│ └── .env.example

│

├── /tracking-service

│ ├── src/

│ │ ├── controllers/

│ │ │ └── trackingController.js

│ │ ├── models/

│ │ │ └── viewingRecord.js

│ │ ├── routes/

│ │ │ └── trackingRoutes.js

│ │ ├── services/

│ │ │ └── kafkaConsumer.js

│ │ ├── app.js

│ │ └── config.js

│ ├── package.json

│ ├── Dockerfile

│ └── .env.example

│

├── /notification-service

│ ├── app/

│ │ ├── Console/

│ │ │ └── KafkaConsumerCommand.php

│ │ ├── Listeners/

│ │ │ └── CameraStatusListener.php

│ │ ├── Services/

│ │ │ └── NotificationService.php

│ │ ├── config/

│ │ │ └── kafka.php

│ │ └── bootstrap.php

│ ├── composer.json

│ ├── Dockerfile

│ └── .env.example

│

├── /k8s

│ ├── auth-deployment.yaml

│ ├── auth-service.yaml

│ ├── camera-deployment.yaml

│ ├── camera-service.yaml

│ ├── monitoring-deployment.yaml

│ ├── tracking-deployment.yaml

│ ├── tracking-service.yaml

│ ├── notification-deployment.yaml

│ ├── gateway-deployment.yaml

│ ├── gateway-service.yaml

│ └── ingress.yaml

│

├── /helm-charts

│ ├── auth/

│ │ ├── Chart.yaml

│ │ ├── values.yaml

│ │ └── templates/

│ │ ├── deployment.yaml

│ │ └── service.yaml

│ ├── camera/

│ │ ├── Chart.yaml

│ │ ├── values.yaml

│ │ └── templates/

│ │ ├── deployment.yaml

│ │ └── service.yaml

│ └── ... (similar para otros servicios)

│

├── docker-compose.yml

├── .gitignore

└── README.md

Archivos Clave con Contenido Inicial

1. docker-compose.yml (versión básica para desarrollo)

yaml

Copy

version: '3.8'

services:

zookeeper:

image: bitnami/zookeeper:latest

ports:

- "2181:2181"

environment:

- ALLOW\_ANONYMOUS\_LOGIN=yes

kafka:

image: bitnami/kafka:latest

ports:

- "9092:9092"

environment:

- KAFKA\_CFG\_ZOOKEEPER\_CONNECT=zookeeper:2181

- ALLOW\_PLAINTEXT\_LISTENER=yes

depends\_on:

- zookeeper

auth-service:

build: ./auth-service

ports:

- "8080:8080"

environment:

- DB\_URL=jdbc:postgresql://db-auth:5432/authdb

- DB\_USER=admin

- DB\_PASSWORD=secret

db-auth:

image: postgres:13

environment:

- POSTGRES\_DB=authdb

- POSTGRES\_USER=admin

- POSTGRES\_PASSWORD=secret

volumes:

- auth-data:/var/lib/postgresql/data

volumes:

auth-data:

2. camera-service/app/main.py (estructura inicial)

python

Copy

from fastapi import FastAPI, WebSocket

from fastapi.middleware.cors import CORSMiddleware

import uvicorn

from .api import endpoints, websockets

app = FastAPI(title="Camera Service")

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_methods=["\*"],

allow\_headers=["\*"],

)

app.include\_router(endpoints.router)

app.add\_websocket\_route("/ws/camera/{camera\_id}", websockets.camera\_stream)

if \_\_name\_\_ == "\_\_main\_\_":

uvicorn.run(app, host="0.0.0.0", port=8000)

3. frontend/src/components/CameraList.tsx (estructura inicial)

typescript

Copy

import React, { useEffect, useState } from 'react';

import { Camera } from '../types';

const CameraList: React.FC = () => {

const [cameras, setCameras] = useState<Camera[]>([]);

const [loading, setLoading] = useState(true);

useEffect(() => {

const fetchCameras = async () => {

try {

const response = await fetch('/api/cameras');

const data = await response.json();

setCameras(data);

} catch (error) {

console.error('Error fetching cameras:', error);

} finally {

setLoading(false);

}

};

fetchCameras();

}, []);

return (

<div>

<h2>Available Cameras</h2>

{loading ? (

<p>Loading cameras...</p>

) : (

<ul>

{cameras.map(camera => (

<li key={camera.id}>

{camera.name} - {camera.status}

</li>

))}

</ul>

)}

</div>

);

};

export default CameraList;

Próximos Pasos Recomendados

1. **Configurar entornos de desarrollo**:

bash

Copy

# Para cada servicio

cd camera-service && pip install -r requirements.txt

cd ../auth-service && mvn clean install

cd ../frontend && npm install

1. **Implementar el servicio de monitoreo básico**:
   * Crear el script que verifique el estado de las cámaras
   * Configurar el productor de Kafka
2. **Configurar la comunicación entre servicios**:
   * Definir los topics de Kafka
   * Establecer contratos de API entre servicios

¿Quieres que empecemos a desarrollar algún servicio en particular? Podríamos comenzar por:

1. El servicio de cámaras (Python) para conectar con las SPcam32
2. El frontend básico para visualizar el estado
3. El sistema de autenticación inicial

¿Cuál prefieres que desarrollemos primero?

# Archivo docker-compose.yml Completo para el Proyecto de Cámaras

## Ubicación del Archivo

El archivo docker-compose.yml debe grabarse en el **directorio raíz del proyecto** (/proyecto-camaras/), al mismo nivel que las carpetas de los microservicios.

## Código Completo

yaml

Copy

version: '3.8'

services:

# Infraestructura

zookeeper:

image: bitnami/zookeeper:latest

ports:

- "2181:2181"

environment:

- ALLOW\_ANONYMOUS\_LOGIN=yes

networks:

- camaras-network

volumes:

- zookeeper\_data:/bitnami/zookeeper

kafka:

image: bitnami/kafka:latest

ports:

- "9092:9092"

- "9093:9093"

environment:

- KAFKA\_CFG\_ZOOKEEPER\_CONNECT=zookeeper:2181

- ALLOW\_PLAINTEXT\_LISTENER=yes

- KAFKA\_CFG\_LISTENER\_SECURITY\_PROTOCOL\_MAP=PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT

- KAFKA\_CFG\_LISTENERS=PLAINTEXT://:9092,PLAINTEXT\_HOST://:9093

- KAFKA\_CFG\_ADVERTISED\_LISTENERS=PLAINTEXT://kafka:9092,PLAINTEXT\_HOST://localhost:9093

- KAFKA\_AUTO\_CREATE\_TOPICS\_ENABLE=true

depends\_on:

- zookeeper

networks:

- camaras-network

volumes:

- kafka\_data:/bitnami/kafka

# Bases de datos

db-auth:

image: postgres:13-alpine

environment:

- POSTGRES\_DB=authdb

- POSTGRES\_USER=admin

- POSTGRES\_PASSWORD=secret

networks:

- camaras-network

volumes:

- auth-data:/var/lib/postgresql/data

healthcheck:

test: ["CMD-SHELL", "pg\_isready -U admin -d authdb"]

interval: 5s

timeout: 5s

retries: 5

db-tracking:

image: mongo:5.0

environment:

- MONGO\_INITDB\_ROOT\_USERNAME=admin

- MONGO\_INITDB\_ROOT\_PASSWORD=secret

networks:

- camaras-network

volumes:

- tracking-data:/data/db

ports:

- "27017:27017"

# Microservicios

auth-service:

build: ./auth-service

ports:

- "8080:8080"

environment:

- DB\_URL=jdbc:postgresql://db-auth:5432/authdb

- DB\_USER=admin

- DB\_PASSWORD=secret

- JWT\_SECRET=mysecretkey

- JWT\_EXPIRATION=86400

depends\_on:

db-auth:

condition: service\_healthy

networks:

- camaras-network

camera-service:

build: ./camera-service

ports:

- "8000:8000"

environment:

- KAFKA\_BOOTSTRAP\_SERVERS=kafka:9092

- CAMERA\_CHECK\_INTERVAL=30

volumes:

- ./camera-service/app:/app

depends\_on:

- kafka

networks:

- camaras-network

monitoring-service:

build: ./monitoring-service

environment:

- KAFKA\_BOOTSTRAP\_SERVERS=kafka:9092

- CAMERA\_LIST=spcam32\_1:192.168.1.100,spcam32\_2:192.168.1.101

depends\_on:

- kafka

networks:

- camaras-network

tracking-service:

build: ./tracking-service

ports:

- "3001:3001"

environment:

- MONGODB\_URI=mongodb://admin:secret@db-tracking:27017

- KAFKA\_BOOTSTRAP\_SERVERS=kafka:9092

depends\_on:

- db-tracking

- kafka

networks:

- camaras-network

notification-service:

build: ./notification-service

environment:

- KAFKA\_BOOTSTRAP\_SERVERS=kafka:9092

- SMTP\_HOST=smtp.example.com

- SMTP\_PORT=587

depends\_on:

- kafka

networks:

- camaras-network

api-gateway:

build: ./api-gateway

ports:

- "3000:3000"

depends\_on:

- auth-service

- camera-service

networks:

- camaras-network

frontend:

build: ./frontend

ports:

- "80:80"

depends\_on:

- api-gateway

networks:

- camaras-network

# Volúmenes para persistencia de datos

volumes:

zookeeper\_data:

kafka\_data:

auth-data:

tracking-data:

# Red para comunicación entre servicios

networks:

camaras-network:

driver: bridge

## ¿Para qué sirve este archivo?

Este archivo docker-compose.yml sirve para:

1. **Orquestar todos los servicios** de la aplicación en un entorno de desarrollo local
2. **Definir las dependencias** entre los distintos microservicios
3. **Configurar la red** para que los servicios puedan comunicarse entre sí
4. **Gestionar volúmenes** para persistencia de datos
5. **Establecer variables de entorno** específicas para cada servicio

## ¿Cómo se ejecuta?

1. **Requisitos previos**:
   * Tener instalado Docker y Docker Compose
   * Tener todos los servicios en sus respectivas carpetas con sus Dockerfiles
2. **Comandos para ejecutar**:

bash

Copy

# Desde el directorio raíz del proyecto (/proyecto-camaras/)

# Construir y levantar todos los servicios (modo detached)

docker-compose up -d --build

# Ver los logs combinados de todos los servicios

docker-compose logs -f

# Detener todos los servicios

docker-compose down

# Reconstruir y reiniciar un servicio específico (ej: camera-service)

docker-compose up -d --build camera-service

# Ver el estado de los contenedores

docker-compose ps

1. **Acceso a los servicios**:
   * Frontend: [http://localhost](http://localhost/)
   * API Gateway: [http://localhost:3000](http://localhost:3000/)
   * Auth Service: [http://localhost:8080](http://localhost:8080/)
   * Camera Service: [http://localhost:8000](http://localhost:8000/)

## Estructura Clave Explicada

1. **Servicios de Infraestructura**:
   * Zookeeper y Kafka para la mensajería
   * PostgreSQL para autenticación
   * MongoDB para tracking de visualizaciones
2. **Microservicios**:
   * Cada uno construido desde su Dockerfile
   * Configurados con sus variables de entorno
   * Conectados a través de una red común
3. **Volúmenes**:
   * Persistencia de datos entre reinicios
   * Datos de Kafka, Zookeeper y bases de datos
4. **Red**:
   * Red bridge aislada para todos los servicios
   * Comunicación interna por nombres de servicio

## Personalización Requerida

Debes actualizar:

1. Las IPs de las cámaras en monitoring-service (variable CAMERA\_LIST)
2. Las credenciales de las bases de datos (mejor usar secrets en producción)
3. Los puertos según tus necesidades
4. Las configuraciones específicas de cada servicio

¿Quieres que explique con más detalle alguna parte específica del docker-compose o cómo configurar algún servicio en particular?

# ¿Cómo probar el servicio? de Autenticación (Java Spring Boot)

1. **Construir y ejecutar**:

bash

Copy

docker-compose up -d db-auth

mvn spring-boot:run

1. **Endpoints disponibles**:
   * POST /auth/login (con cuerpo JSON: {"username":"admin","password":"password"})
   * GET /auth/validate (con header Authorization: Bearer <token>)
2. **Usuarios iniciales**:
   * Necesitarás crear al menos un usuario admin (puedes añadirlo en data.sql o crear un endpoint de registro)

¿Quieres que implementemos algún componente adicional o que expliquemos con más detalle alguna parte del código?

Consultas SQL para Crear las Tablas Necesarias

Para resolver el error "Unknown table 'sequences' in information\_schema" en MySQL, necesitamos crear manualmente las tablas que tu aplicación requiere. Aquí están las consultas SQL necesarias:

1. Primero, crea la base de datos (si no existe)

sql

Copy

CREATE DATABASE IF NOT EXISTS authdb CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci;

2. Tabla de roles

sql

CREATE TABLE IF NOT EXISTS roles (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(20) NOT NULL UNIQUE

);

3. Tabla de usuarios

sql

Copy

CREATE TABLE IF NOT EXISTS users (

id BIGINT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(20) NOT NULL UNIQUE,

email VARCHAR(50) NOT NULL UNIQUE,

password VARCHAR(120) NOT NULL

);

4. Tabla de relación usuario-rol

sql

Copy

CREATE TABLE IF NOT EXISTS user\_roles (

user\_id BIGINT NOT NULL,

role\_id INT NOT NULL,

PRIMARY KEY (user\_id, role\_id),

FOREIGN KEY (user\_id) REFERENCES users(id) ON DELETE CASCADE,

FOREIGN KEY (role\_id) REFERENCES roles(id) ON DELETE CASCADE

);

## 5. Datos iniciales

sql

Copy

-- Insertar roles básicos

INSERT IGNORE INTO roles(name) VALUES('ROLE\_USER');

INSERT IGNORE INTO roles(name) VALUES('ROLE\_ADMIN');

-- Insertar usuario admin contraseña admin123

INSERT INTO users (username, email, password) VALUES (

'admin',

'admin@example.com',

'$2a$10$Nn0W4r5LZBLo84Rf4yhrAOhEy8q0whHnCW97yISXz7XjwZ3fzUPzK'

);

-- Asignar rol de admin al usuario admin

INSERT IGNORE INTO user\_roles(user\_id, role\_id) VALUES(

(SELECT id FROM users WHERE username = 'admin'),

(SELECT id FROM roles WHERE name = 'ROLE\_ADMIN')

);

6. Consultas adicionales útiles

Para verificar la estructura creada:

sql

Copy

-- Ver tablas existentes

SHOW TABLES;

-- Ver estructura de una tabla

DESCRIBE users;

DESCRIBE roles;

DESCRIBE user\_roles;

-- Ver datos

SELECT \* FROM users;

SELECT \* FROM roles;

SELECT \* FROM user\_roles;

Cómo ejecutar estas consultas

1. Accede a la consola de MySQL:

bash

Copy

docker exec -it [nombre\_contenedor\_mysql] mysql -u admin -p

1. Selecciona la base de datos:

sql

Copy

USE authdb;

1. Ejecuta las consultas SQL anteriores una por una.

# Cómo Probar tu Servicio de Autenticación Spring con MySQL

Para verificar que tu auth-service funciona correctamente, puedes realizar las siguientes pruebas:

## 1. Pruebas Básicas con Postman

Registrar un Usuario (si tienes este endpoint)

Copy

POST http://localhost:8080/auth/login

Content-Type: application/json

{

"username": "admin",

"email": "test@example.com",

"password": "admin123",

"roles": ["ROLE\_USER"]

}

4. Pruebas Automatizadas con JUnit

Crea una clase de prueba en Eclipse:

java

Copy

@SpringBootTest

@AutoConfigureMockMvc

class AuthControllerTests {

@Autowired

private MockMvc mockMvc;

@Test

void testLogin() throws Exception {

String requestBody = "{\"username\":\"testuser\",\"password\":\"testpassword\"}";

mockMvc.perform(MockMvcRequestBuilders

.post("/api/auth/login")

.content(requestBody)

.contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(jsonPath("$.token").exists());

}

}