Guía Completa: Dos Microservicios NestJS con CRUD, Autenticación JWT y API Gateway Ocelot

Objetivo: Construir tres componentes que trabajen en conjunto sin exponer código en la guía, pero con instrucciones detalladas:

- **Microservicio A ("users-service")**: gestión de usuarios con CRUD y autenticación JWT.
- **Microservicio B ("items-service")**: CRUD completo sobre un recurso "items".
- **API Gateway ("gateway")**: usa Ocelot para unificar rutas, validar tokens, hacer cache y balanceo.

Al terminar, tendrás un sistema funcional donde:

1. El cliente se autentica contra users-service y recibe un JWT.
2. El gateway valida el JWT y enruta peticiones protegidas a users-service o items-service.
3. Ambos microservicios exponen operaciones CRUD completas.

---

1. Preparación del Entorno

1. **Node.js** (versión LTS recomendada, p. ej. 18.x).
2. **Nest CLI**: instalar globalmente y verificar con `nest --version`.
3. **.NET 6+ SDK**: instalar y confirmar con `dotnet --version`.
4. **Git**: configurar usuario y correo, clonar un repo vacío.
5. **Postman o Insomnia**: para pruebas de APIs.

> 💡 Usa WSL o Terminal de Mac si trabajas en Windows para mejor experiencia.

---

2. Estructura de Carpetas

Dentro de tu repositorio local crea:

```
mi-proyecto/
├── users-service/      ← Servicio A (CRUD usuarios + autenticación)
├── items-service/      ← Servicio B (CRUD items)
└── api-gateway/        ← Proyecto Ocelot (.NET Web)
```

Cada carpeta es un proyecto independiente, versionado y desplegable por separado.

---

3. Microservicio A: `users-service`

Funcionalidad: Registro, login (genera JWT), y gestión completa de usuarios (crear, leer, actualizar, borrar).

3.1 Inicialización del proyecto

1. Navega a `users-service/` y arranca Nest CLI.
2. Nómbralo `users-service` y selecciona opciones de configuración: TypeScript, ESLint, etc.
3. Instala paquetes esenciales:
4. **@nestjs/jwt** y **passport-jwt** para autenticación.
5. **@nestjs/passport** y **passport**.
6. **@nestjs/config** para variables de entorno.
7. **class-validator** y **class-transformer** para validar DTOs.

3.2 Diseño de la base de datos (in memory o SQLite)

1. Usa un módulo ORM ligero (p. ej. TypeORM o Prisma).
2. Crea una entidad `User` con campos básicos: id (UUID), nombre, correo, password (hash).
3. Configura conexión a BD desde variables de entorno.

3.3 Implementar Auth y CRUD (sin mostrar código)

1. **Módulo Auth**:
2. Define estrategia JWT que extrae el token del header `Authorization: Bearer <token>`.
3. Crea guardas (Guards) que protejan rutas.
4. **Controlador AuthController**:
5. `POST /auth/register`: recibe nombre, correo y contraseña; guarda usuario; devuelve datos sin el hash.
6. `POST /auth/login`: valida credenciales; si son correctas, genera y devuelve un JWT.
7. **Controlador UsersController** (protegido con JWT):
8. `GET /users`: lista todos los usuarios.
9. `GET /users/:id`: obtiene usuario por id.
10. `PUT /users/:id`: actualiza datos de un usuario.
11. `DELETE /users/:id`: borra un usuario.
12. **Validaciones y seguridad**:
13. Valida DTOs con decoradores de clase.
14. Hash de contraseñas con bcrypt.
15. Nunca devuelvas el campo `password` en respuestas.

3.4 Pruebas locales

1. Arranca `users-service` en `localhost:3001`.
2. En Postman:
3. Registra un usuario y verifica JWT.
4. Usa JWT para acceder a rutas CRUD de `/users`.
5. Prueba errores: token inválido, datos faltantes.

---

4. Microservicio B: `items-service`

Funcionalidad: CRUD básico sobre “items” con campos id (UUID), nombre, descripción.

4.1 Inicialización del proyecto

1. Entra a `items-service/` y lanza Nest CLI.
2. Nómbralo `items-service`.
3. Instala condensados:
4. **@nestjs/config**
5. **class-validator** y **class-transformer**

4.2 Diseño de datos

1. Elige memoria o mismo ORM que users (SQLite/TypeORM).
2. Define entidad `Item` con id, nombre y descripción.

4.3 Implementar CRUD

1. **ItemsController**:
2. `GET /items`: lista todos.
3. `GET /items/:id`: obtiene uno.
4. `POST /items`: crea item.
5. `PUT /items/:id`: actualiza.
6. `DELETE /items/:id`: borra.
7. Asegúrate de validar DTOs y manejar errores (404 cuando no exista).

4.4 Pruebas locales

1. Arranca `items-service` en `localhost:3002`.
2. En Postman prueba todas las rutas, tanto con datos correctos como con errores.

---

5. API Gateway: `api-gateway` con Ocelot

Objetivo: Unificar rutas, validar JWT, activar cache y balanceo.

5.1 Inicializar proyecto .NET

1. Entra a `api-gateway/` y crea un proyecto web vacío (`dotnet new web`).
2. Agrega paquete NuGet `Ocelot`.

5.2 Configuración de `ocelot.json`

Crea `ocelot.json` y define:

- **GlobalConfiguration**:

- Tiempo de espera, log level.

- **ReRoutes**:

- **Auth público**:

    - UpstreamPath: `/api/auth/{everything}`
    - DownstreamHostAndPorts: `localhost:3001,/auth/{everything}`
    - No requiere autenticación.
- **Users protegido**:
    - UpstreamPath: `/api/users/{everything}`
    - DownstreamHostAndPorts: `localhost:3001,/users/{everything}`
    - Activa JwtBearerTokenMiddleware para validar JWT.

- **Items protegido**:

    - UpstreamPath: `/api/items/{everything}`
    - DownstreamHostAndPorts: `localhost:3002,/items/{everything}`
    - También validado con JWT.

- **CacheOptions** (por ejemplo para GET `/api/items` con TTL 60s).

- **LoadBalancerOptions** (si levantas múltiples instancias de algún servicio).

5.3 Habilitar Ocelot en Startup

1. Lee `ocelot.json` desde configuración.
2. Registra middleware de Ocelot.
3. Configura JWT Validation Parameters (mira clave secreta y alg).
4. Define puerto de escucha (p. ej. 5000).

5.4 Correr y probar

1. Levanta `gateway` en `localhost:5000`.
2. Pruebas:
3. **Registro y login**: POST a `/api/auth/register` y `/login` → obtén JWT.
4. **Rutas protegidas**: usa JWT en header `Authorization: Bearer <token>` para:
    - GET/PUT/DELETE en `/api/users`.
    - GET/POST/PUT/DELETE en `/api/items`.
5. Sin JWT o token inválido: comprueba 401.

---

6. Escenarios de Falla y Debug

1. **Servicio caído**: apaga uno de los microservicios y prueba ruta correspondiente en el gateway → recibe 502.
2. **Token expirado**: vuelve a usar un token viejo → recibe 401.

3. **Error en ruta**: GET a `/api/users/nonexistent-id` → recibe 404.
4. **Logs avanzados**: sube nivel de logs a DEBUG en cada proyecto y rastrea la petición completa.

---

7. Extensiones Avanzadas

- **Balanceo**: lanza dos instancias de items-service en puertos 3002 y 3003, configura load-balancer.
- **Swagger**: expón documentación en cada microservicio y en el gateway, si deseas.
- **Circuit Breaker**: usa políticas de resiliencia (p. ej. App Metrics + Polly).

---

8. Buenas Prácticas

- Centraliza configuración sensible en variables de entorno.
- Versiona tu `ocelot.json` y ten un diff claro cuando actualices rutas.
- Documenta cada servicio y actualiza README con comandos CLI usados.
- Integra tests e2e que lancen peticiones al gateway y verifiquen comportamientos.

¡Con esta guía detallada, tendrás un sistema robusto de microservicios con autenticación y un gateway profesional usando Ocelot! Pilas pues, practica cada paso y revisa bien los logs para entender el flujo completo.