Page Alert

Documentación para desarrolladores

Óscar Casais López

PAGE ALERT

01/08/2025

Índice

1 Presentación **2**
1.1 Objetivos . . . 2
1.2 Funcionalidades principales . . . 2

2 Planificación **2**
2.1 Cronograma . . . 3
2.2 Recursos necesarios . . . 3
2.3 Requisitos funcionales generales . . . 3
2.4 Dependencias generales . . . 4
2.5 Requisitos funcionales: alcance según versiones . . . 4
2.5.1 `0.1.0` . . . 4
2.5.2 `0.2.0` Requisitos generales . . . 5
2.6 Requisitos no funcionales . . . 5
2.7 Dependencias . . . 5
2.7.1 `0.1.0` scraper-service . . . 5
2.7.2 `0.2.0` scraper-service . . . 5

3 Análisis/Diseño **5**
3.1 Base de datos . . . 5
3.2 Componentes principales del Sistema . . . 7

4 Implementación **7**
4.1 Springboot . . . 7
4.1.1 Implementa cían de Microservicios . . . 9
4.2 Estado actual . . . 10

5 Testing **10**

6 Despliegue **10**

7 Seguimiento **11**
7.0.1 Plan de acción para la finalización . . . 12

8 Siguientes pasos **12**

1. Presentación

- Objetivos
- Funcionalidades principales

1.1. Objetivos

El proyecto `page-alert` es un proyecto personal que intenta aunar diferentes tecnologías en las que estoy interesado:

- Arquitectura Microservicios
- Docker
- OAuth
- Seguridad

El Objetivo principal de este sistema es el de ofrecer una herramienta al público en general en la que un usuario pueda hacer un seguimiento de páginas web en las que tenga interés. El sistema se encargará de avisar al usuario cuando algún cambio se produzca en las páginas que sigue.

1.2. Funcionalidades principales

- Login de usuario.
- Lista de páginas y reglas de usuario.
- Ayuda 'interactiva' para la creación de reglas.
- CRUD de páginas y reglas.
- Monitoreo recurrente de las páginas.
- Notificación por la vía configurada de los cambios encontrados.

2. Planificación

- Cronograma
- Recursos necesarios
- Requisitos funcionales generales
- Dependencias generales
- Requisitos funcionales: alcance según versiones
- Requisitos no funcionales
- Dependencias

2.1. Cronograma

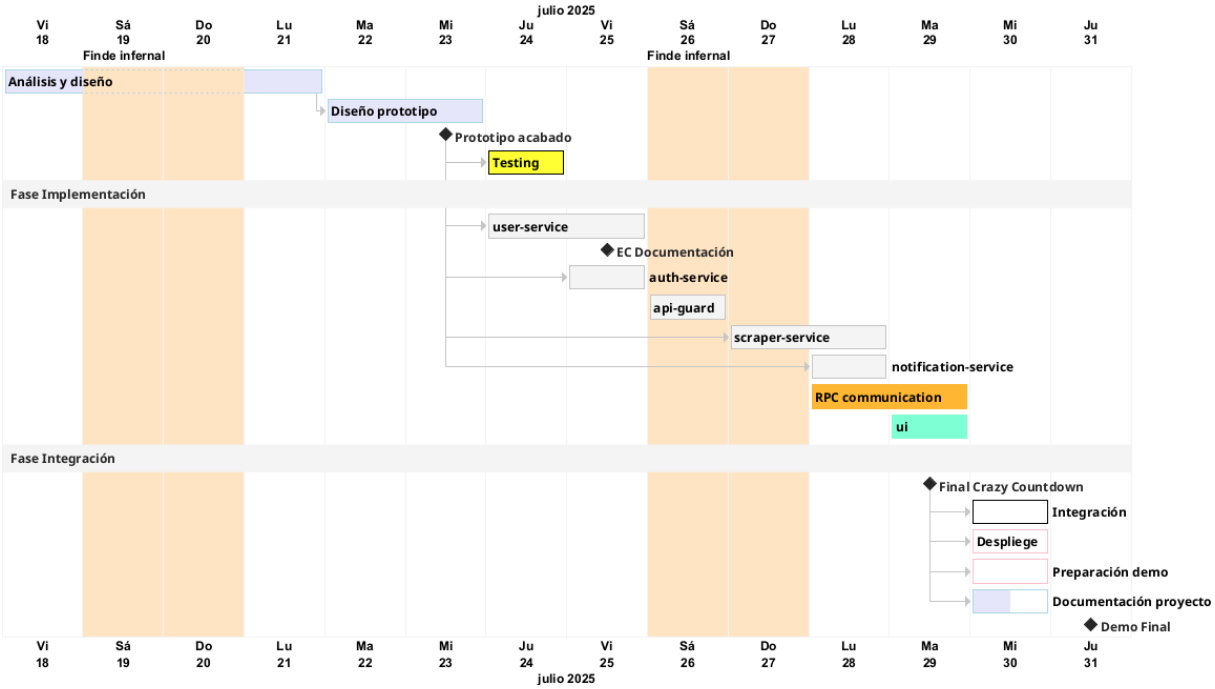

Figura 1: cronograma

2.2. Recursos necesarios

El MVP está planteado para desplegarlo solamente en local. Los recursos de personal son muy limitados (1 persona) y los recursos disponibles son un portátil Intel Core i7 con 16GB de RAM.

A priori los microservicios no deberían de consumir muchos recursos ya que no tienen procesos pesados. El proceso más intensivo sin lugar a dudas es responsabilidad del `scraper-service` pero para la Demo del día 31/072025 solamente trabajará contra el localhost y en intervalos fácilmente asequibles.

A futuro con la versión `0.2.0` nos integraremos con una herramienta externa que hará todo este proceso de manera externa.

2.3. Requisitos funcionales generales

- Login de usuario vía email/pass y SSO:
  - CRUD cuenta de usuario y sus preferencias para las notificaciones.

- Lista de páginas y reglas de usuario.
  - CRUD URLs y reglas de monitoreo.
- Ayuda interactiva del sistema para la creación de reglas.
- Monitoreo recurrente de las páginas.
- Notificación por la vía configurada de los cambios encontrados.

2.4. Dependencias generales

Microservicios

- Spring Security `Security`
- PostgreSQL Driver `SQL`
- Spring Data JPA `SQL`
- H2 Database `SQL` (para desarrollo: base de datos en memoria)
- Spring Web `Web`
- Spring Boot DevTools Developer `Tools` (para desarrollo: reloads, restarts…)
- Validation `I/O`

UI

- Vue3 usando Quasar.

2.5. Requisitos funcionales: alcance según versiones

2.5.1. `0.1.0`

- **Notificaciones**
  - Email
  - Push
- **Reglas de Monitoreo** (localhost):
  - Toda la página: `bodyHash`
  - Elemento/s concretos (sin ayuda interactiva): `['css-selector']`
- **Interfaz** simple para acceso de usuarios y gestión de avisos.

2.5.2. `0.2.0` Requisitos generales

- **Reglas de Monitoreo**:
  - Añadir **ayuda interactiva** para la creación de reglas:
    - Elementos concretos: `['css-selector']`
    - Precio automático
- **Interfaz** con conexión interactiva con el servidor para la validación de reglas.

2.6. Requisitos no funcionales

- El sistema debe operar en todo momento con protocolos seguros:
  - HTTPS para la comunicación con el Front End.
  - RPC entre microservicios.

2.7. Dependencias

2.7.1. `0.1.0` scraper-service

- Playwright - localhost

2.7.2. `0.2.0` scraper-service

- Integración con changedetecion.io

3. Análisis/Diseño

- Base de datos
- Componentes principales del Sistema

3.1. Base de datos

Tras un análisis de las diferentes opciones de gestores de Base de Datos. Me decanto por **PostgreSQL** porque se asemeja mucho a MYSQL pero proporciona funcionalidades extra: colas de mensajes, crons, criptografía …

El concepto principal de los microservicios es que deben de estar tan desacoplados como sea posible, es por esto que cada microservicio tendrá su propia base de datos.

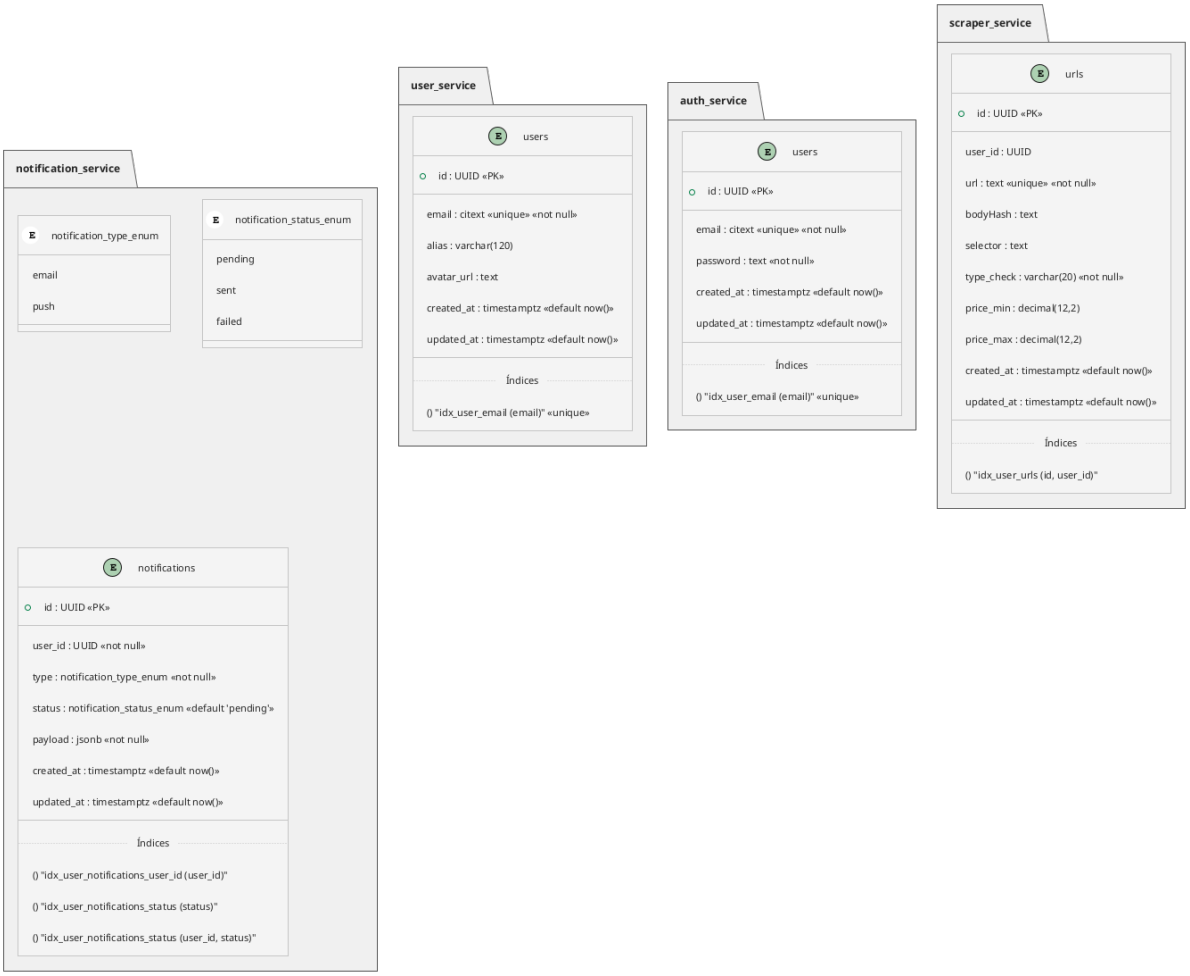

Figura 2: Esquema DB

3.2. Componentes principales del Sistema

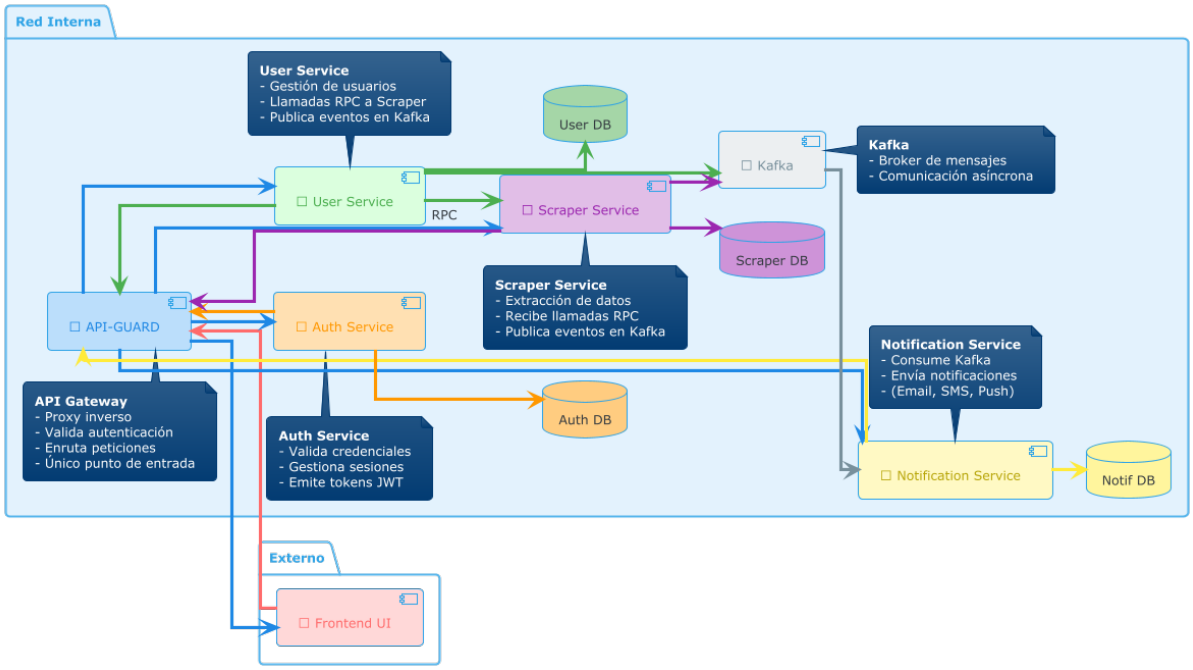

Figura 3: Vista general del sistema

4. Implementación

- Springboot
  - Implementación de Microservicios
- Estado actual

4.1. Springboot

Vamos a usar la librería de Springboot. La cual nos proporciona una serie de Plugins que nos simplifica enormemente el proceso de desarrollo de nuestros microservicios.

Utilizaremos una arquitectura típica y fiable como la siguiente.

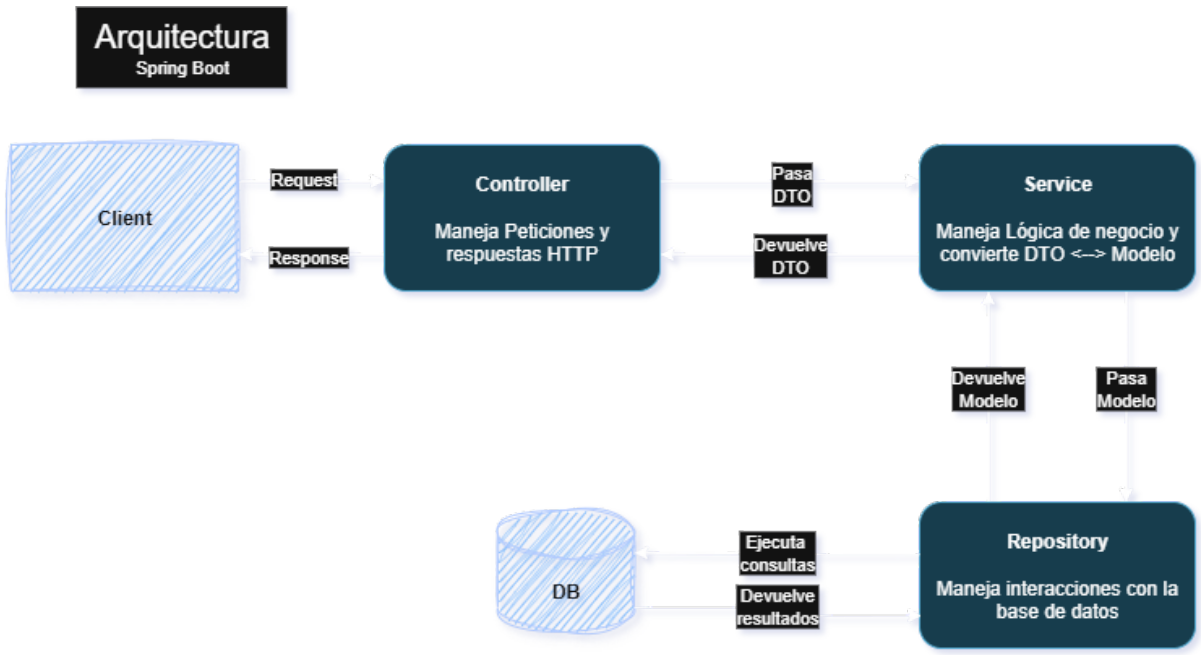

Figura 4: Arquitectura de Springboot

> La capa de Servicio es la única que conoce tanto al Modelo como al DTO, esto es clave en este diseño.

Los plugins que más vamos a usar son:

Spring Boot Security

- Autenticación/autorización básica.
- Protege endpoints con config mínima (ej: `@PreAuthorize`).

Spring Boot JPA

- Abstracción de bases de datos (ORM).
- Elimina boilerplate SQL con repositorios (`CrudRepository`).

Spring Boot Testing

- Pruebas integradas (mock MVC, JUnit, etc.).
- Anotaciones como `@SpringBootTest` simplifican testing E2E.

4.1.1. Implementa cían de Microservicios

Sirva de ejemplo el componente `user-service`. La estructura siguiente la seguiremos para todos los componentes del sistema.

```
C:\USERS\OSCAR\PAGE-ALERT\USER-SERVICE
|   .gitattributes
|   .gitignore
|   docker-compose.yml
|   Dockerfile
|   pom.xml
\---src
    +---main
    |   +---java
    |   |   \---es
    |   |       \---oscasais
    |   |           \---pa
    |   |               \---userService
    |   |                   |   UserServiceApplication.java
    |   |                   |
    |   |                   +---controller
    |   |                   |       UserController.java
    |   |                   |
    |   |                   +---dto
    |   |                   |       UserRequestDTO.java
    |   |                   |       UserResponseDTO.java
    |   |                   |
    |   |                   +---exception
    |   |                   |       EmailAlreadyExistsException.java
    |   |                   |       GlobalExceptionHandler.java
    |   |                   |       UserNotFoundException.java
    |   |                   |
    |   |                   +---mapper
    |   |                   |       UserMapper.java
    |   |                   |
    |   |                   +---model
    |   |                   |       User.java
    |   |                   |
    |   |                   +---repository
    |   |                   |       UserRepository.java
    |   |                   |
    |   |                   \---service
    |   |                           UserService.java
    |   |
    |   \---resources
    |           data.sql
    |
    \---test
        \---java
            \---es
```

```
                \---oscasais
                    \---pa
                        \---userService
                                PageAlertServerApplicationTests.java
```

4.2. Estado actual

> Repositorio Github del proyecto

A fecha 01/08/2025 el desarrollo está en una fase muy embrionaria. La `0.0.1` no está ni mucho menos acabada ni completa. Queda mucho trabajo por delante. La buena noticia es que ya tenemos todos los microservicios (a excepción de `notification-service`) y la integración con `changedetection.io`. Hemos mejorado mucho en tan solo 4 días.

5. Testing

El plan inicial es que cada componente tenga sus propia Suites de test los cuales se probaran con una base de datos en memoria (H2).

Estos test tendrán como objetivo el confirmar tanto que las operaciones CRUD se hagan de manera satisfactoria como que todas las reglas de negocio definidas en el modelo se cumplan.

En el estado actual no hemos creado todavía estos Tests Unitarios. Pero hemos realizado diferentes pruebas manuales confirmando que la API de nuestro único (de momento) componente: `user-service` actúa como esperamos.

Se incluye junto con la documentación un video con pruebas de la API del servicio `user-service`

6. Despliegue

El despliegue lo haremos utilizando `docker-compose`. Cada uno de los Componentes a desplegar tendrá un archivo `Docker` que se encargará de levantar el servicio por nosotros.

Esto nos permite fácilmente desplegarlo tanto en entornos de desarrollo como a producción.

En la implementación actual cada uno de los Componentes tendrá un archivo `.env` que es de donde cargará los ***secrets***. A futuro implementaremos esta lógica mediante archivos secrets que Docker no persiste en disco sino en memoria.

```
# Serve only as an example
# docker-compose.yml
secrets:
   db_password:
     file: db_password.txt
   db_root_password:
     file: db_root_password.txt
```

7. Seguimiento

Progreso realizado

- Configuración inicial del entorno Docker con PostgreSQL y Spring Boot.
- Creación del contenedor `user-service-db` y mapeo de puerto 5000→5432.
- Creación del microservicio `user-service` y mapeo de puerto 4000→4000.
- Integración exitosa de la aplicación con la base de datos. También memoria para los futuros test unitarios.
- Desarrollo del script `data.sql` con 20 registros de prueba.

Desafíos encontrados

- **Fallos en Configuración inicial**:
  - El IDE no me reconocía los módulos dentro del proyecto padre.
    Solución: ponerlos como dependencia en el `pom.xml` del padre.
  - Evitar nombre `user` como tabla (da error con H2 database (la que se levanta en memoria), en el script de generación evitar uso de doble comilla “” para nombre de atributos y de tablas.
  - **Conflicto de Schema entre H2 y PostgreSQL**: Que hacía que el servicio no se levantara. El problema estaba en cómo definíamos la PK en el `data.sql`. **Solución**: Hacer el script inicial mucho más sencillo (es solo un script para testing)
  - **Error de inicialización de Docker**: Fallo al iniciar `user-service` por conexión prematura a `user-service-db`.
    Solución: añadir `healthcheck` en PostgreSQL y `depends_on condition: service_healthy`.

7.0.1. Plan de acción para la finalización

Es todavía pronto, pero diría que va a ser harto difícil acabar el proyecto tal cual lo hemos planteado con fecha **31/07/2025**.

A día de hoy tan solo hemos acabado 1/5 componentes de BackEnd y falta el FrontEnd. Debemos intentar dar un enfoque más práctico.

Acciones que vamos a tomar

- Vamos a pasar de puntillas por el tema de la seguridad: Al final, el día de la entrega vamos a estar corriendo todo en local y es con el objetivo de ofrecer una prueba de concepto.
- No vamos a implementar la comunicación RPC entre `user-service` y `scraper-service`
- El `notification-service` podríamos abandonarlo y ver la posibilidad de usar una herramienta externa como apprise

```
docker run -d -p 8000:8000 --name apprise caronc/apprise-api
```

```
record AprisePayload(String urls, String title, String body) {}

RestTemplate rest = new RestTemplate();
AprisePayload payload = new AprisePayload(
        "discord://webhook_id/webhook_token",
        "Alerta",
        "¡Todo está arriba!"
);
rest.postForLocation("http://localhost:8000/notify", payload);
```

La prioridad es tener un sistema funcional que sirva a modo de Demostración.

8. Siguientes pasos

Debemos de sortear el problema de CORS que estamos teniendo para comunicar los servicios o bien implementar correctamente Kafka y comunicarnos internamente sin hacer uso de HTTP.

La seguridad de los datos es muy importante, así pues, debemos de introducir Mutual/TLS para las conexiones a la base de datos.

La implementación del `notification-service` queda toda pendiente.

Debemos de empezar a hacer Suite de tests unitarios para hacer el sistema robusto a posibles errores.

La interfaz de usuario necesita serias mejoras.

Queda mucho camino por delante.