Fecha:
Octubre 2024

Documentación de URL Shortener para Promociones

Propuesta de software

Realizado por:
Augusto Occhiuzzi

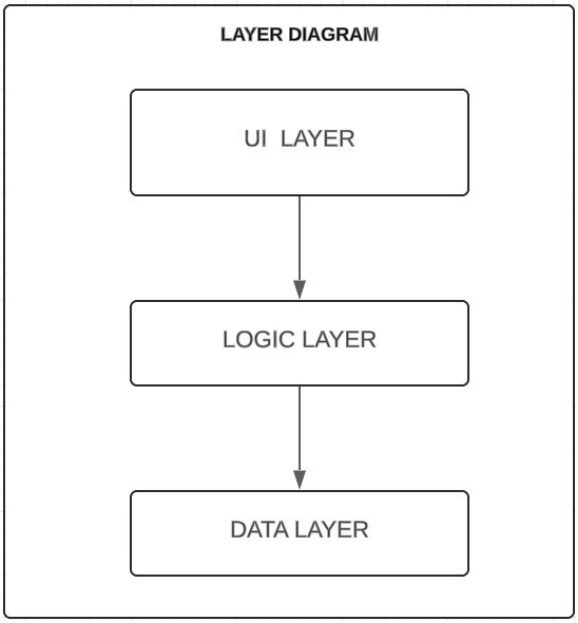
Introducción

El presente documento describe la arquitectura propuesta para un sistema de URL Shortener que permitirá generar enlaces cortos para ser utilizados en campañas de promoción en redes sociales como Twitter. La solución debe ser capaz de gestionar tráfico de hasta 1 millón de requests por minuto (RPM), operar con una disponibilidad del 99.98%, y ofrecer estadísticas de acceso casi en tiempo real. Además, las URLs generadas deben poder ser habilitadas o deshabilitadas y permitir la modificación de los componentes de su URL de destino.

El objetivo es garantizar la alta disponibilidad, escalabilidad, y eficiencia de costos, cumpliendo con los requerimientos funcionales y no funcionales exigidos para el proyecto.

Primeras aproximaciones:

El modelo esta basado en una arquitectura de capas, en la que tendremos un cliente como UI, el cual hara las peticiones a la capa logica y nuestra capa logica sera luego la encargada de la persistencia



Arquitectura Propuesta

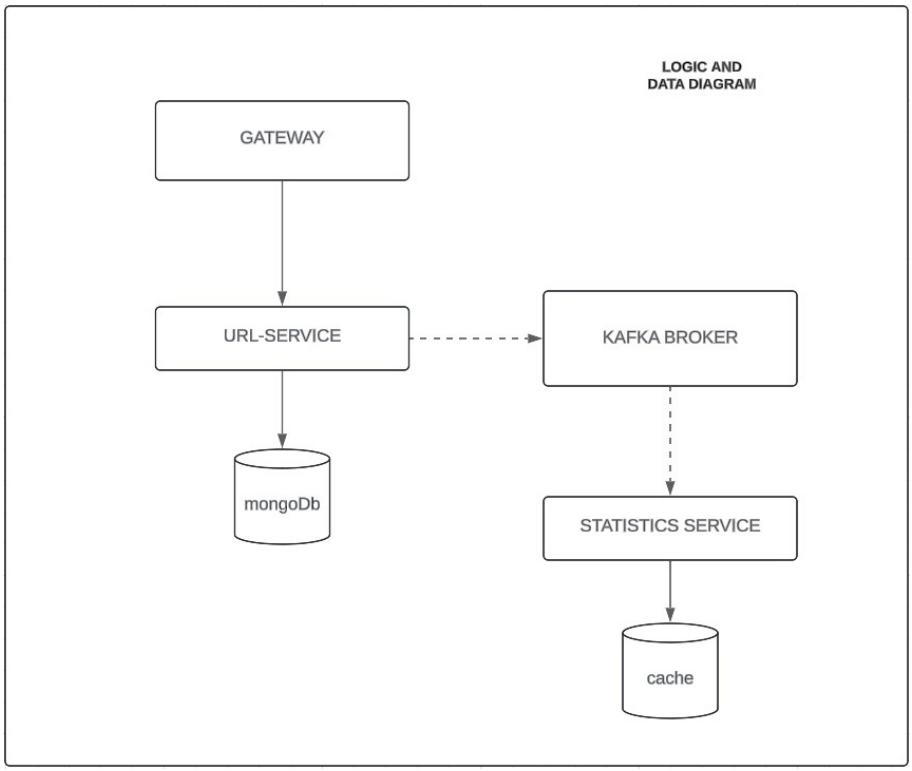
La solución está compuesta por una arquitectura de microservicios distribuida, con los siguientes componentes principales:

- **API Gateway**: Controla la autenticación y la autorización de los usuarios, permitiendo el acceso a los microservicios. Tambien se podria validar roles para ciertos recursos.

- **Microservicio UrlManager**: Responsable de la generación, modificación, habilitación/deshabilitación y resolución de las URLs cortas.

- **Microservicio StatisticsService**: Se encarga de recolectar, procesar y almacenar las métricas de acceso a las URLs.

Los microservicios se comunican a través de un bus de eventos asíncrono mediante **Kafka**, y se utilizan bases de datos **MongoDB** y **Redis** para la persistencia y caching, respectivamente.



Selección de Tecnologías y Justificación

Java 17 con Spring WebFlux

Elección: Para el desarrollo de la lógica de negocio de los microservicios se ha utilizado **Java 17** junto con **Spring WebFlux**, una librería reactiva de Spring que permite crear aplicaciones no bloqueantes.

Justificación:

Reactivo y no bloqueante: WebFlux está basado en un modelo reactivo, lo cual es crucial para manejar de forma eficiente grandes cantidades de tráfico concurrente. Con picos de hasta 1M RPM, se requiere una arquitectura no bloqueante para evitar cuellos de botella y garantizar tiempos de respuesta rápidos.

Soporte y madurez: Java es un lenguaje maduro y Spring WebFlux es ampliamente utilizado en aplicaciones de alto rendimiento. Ambas tecnologías cuentan con una comunidad sólida y soporte extendido.

Escalabilidad: WebFlux permite manejar de forma eficiente aplicaciones distribuidas que necesitan escalar horizontalmente, alineándose con el requisito de soportar grandes volúmenes de tráfico.

Microservicios UrlManager y StatisticsService

Elección: La lógica de negocio se ha dividido en dos microservicios:

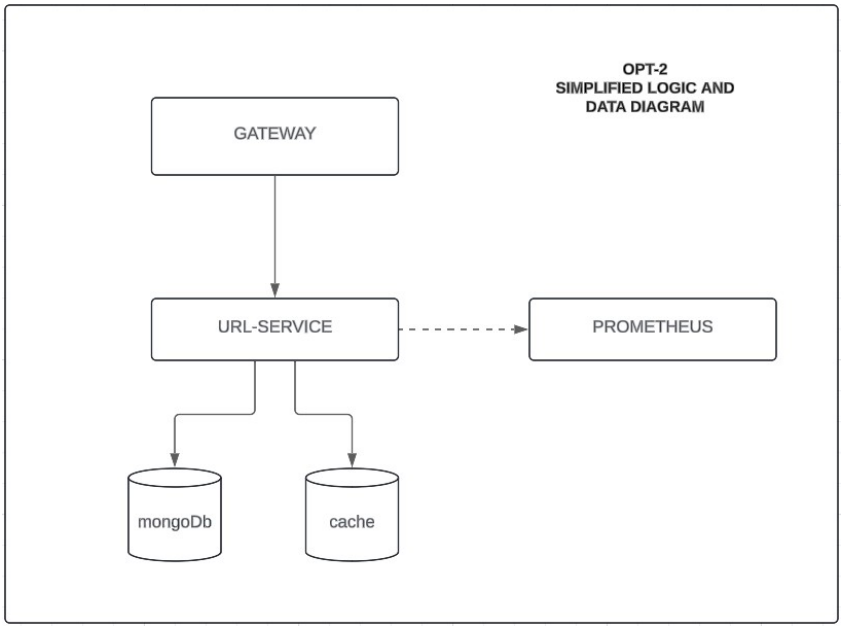
- **UrlManager**: Gestión de URLs cortas (crear, modificar, habilitar, deshabilitar).
- **StatisticsService**: Procesamiento y almacenamiento de estadísticas de acceso.

Justificación:

Desacoplamiento de responsabilidades: Separar las funciones en diferentes microservicios permite un mejor manejo de la escalabilidad y facilita el mantenimiento. Al tener servicios independientes, es posible escalar solo el microservicio que maneje la funcionalidad más demandada (por ejemplo, el UrlManager), evitando sobrecargar el sistema completo.

Disponibilidad y mantenibilidad: Esta división de responsabilidades también favorece la alta disponibilidad. Si uno de los microservicios experimenta problemas, el otro puede continuar funcionando, manteniendo la operatividad del sistema.

Alternativa evaluada (servicio unico): Una opción hubiera sido implementar toda la lógica en un solo servicio, pero esta opción generaría un acoplamiento excesivo, lo que dificultaría la escalabilidad y el mantenimiento del sistema, así como el manejo eficiente del tráfico elevado.



<u>MongoDB para almacenamiento de URLs</u>

Elección: Se ha elegido **MongoDB**, una base de datos NoSQL distribuida, para almacenar la información relacionada con las URLs generadas.

Justificación:

Escalabilidad horizontal: MongoDB es una base de datos diseñada para escalar horizontalmente, lo que significa que puede soportar grandes cantidades de tráfico distribuyendo la carga entre múltiples nodos.

Flexibilidad del esquema: Dado que las URLs pueden contener información variada, MongoDB permite la flexibilidad de almacenamiento con un esquema dinámico, lo que facilita el manejo de datos heterogéneos como las URLs cortas y sus componentes asociados.

Alternativa evaluada (SQL): Las bases de datos relacionales como MySQL o PostgreSQL también hubieran sido una opción. Sin embargo, estas bases de datos podrían haber presentado limitaciones en cuanto a la escalabilidad horizontal y flexibilidad, especialmente en un entorno con picos de

tráfico de 1M RPM. MongoDB, al ser NoSQL, soporta de manera más eficiente este tipo de carga distribuida.

Kafka para la comunicación asíncrona entre microservicios

Elección: Para la transmisión de eventos entre los microservicios, se ha seleccionado **Apache Kafka**.

Justificación:

Alta concurrencia y throughput: Kafka está diseñado para manejar grandes volúmenes de datos y alto throughput, lo que lo hace ideal para entornos con alta concurrencia de peticiones, como es el caso del acceso a estadísticas en tiempo real.

Desacoplamiento asíncrono: La comunicación asíncrona entre los microservicios mejora la eficiencia y reduce la latencia, ya que los servicios pueden procesar los eventos de manera independiente sin bloquear la ejecución de otros procesos.

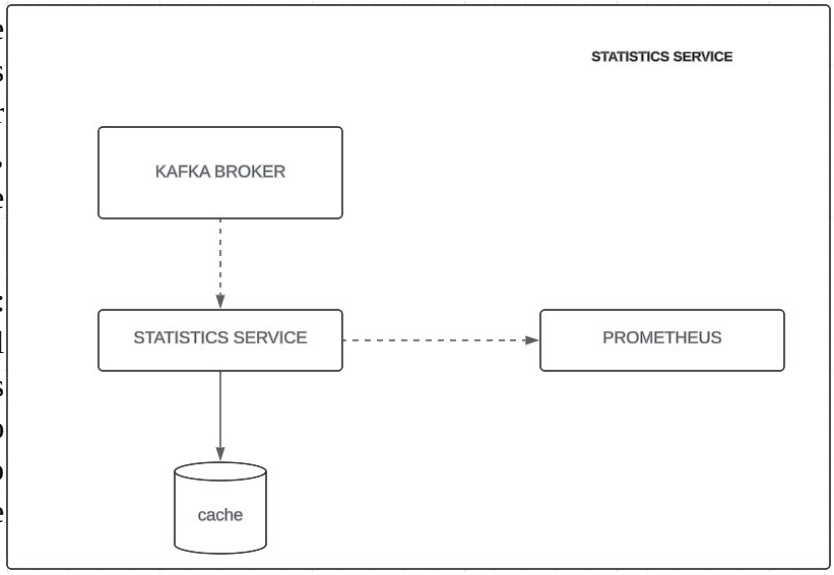
Alternativa evaluada (REST síncrono): La comunicación directa y síncrona entre los servicios hubiera generado un acoplamiento mayor, aumentando la latencia y reduciendo la escalabilidad. Kafka permite un desacoplamiento efectivo, garantizando que los servicios se puedan escalar de forma independiente.

Redis para caching de estadísticas

Elección: Se ha optado por **Redis** como capa de caching para almacenar temporalmente las estadísticas de acceso antes de su persistencia y publicación.

Justificación:

Bajo tiempo de respuesta: Redis es conocido por ser extremadamente rápido gracias a que almacena datos en memoria, lo que permite acceder a las estadísticas casi en tiempo real, cumpliendo con el requerimiento de latencia mínima.



Eficiencia en la lectura de datos: Redis mejora significativamente el tiempo de acceso a datos frecuentemente consultados, como las estadísticas de URLs, reduciendo la carga sobre el microservicio de estadísticas y MongoDB.

Alternativa evaluada (almacenamiento directo en MongoDB): Almacenar las estadísticas directamente en MongoDB hubiera incrementado los tiempos de lectura/escritura, generando una latencia mayor. Redis actúa como una capa intermedia eficiente, disminuyendo la latencia y mejorando la performance general del sistema.

Prometheus para monitorización de métricas

Elección: Para la recopilación y visualización de métricas en tiempo real, se ha utilizado **Prometheus**.

Justificación:

Monitoreo eficiente: Prometheus es una de las soluciones más populares para monitorear aplicaciones distribuidas y microservicios. Es capaz de manejar grandes volúmenes de datos y ofrece flexibilidad para recopilar y visualizar estadísticas en tiempo real.

Integración sencilla con Redis y Kafka: Prometheus se integra fácilmente con Redis y Kafka para capturar métricas, lo que facilita la instrumentación del sistema sin añadir complejidad adicional.

Escalabilidad y Orquestación con Kubernetes

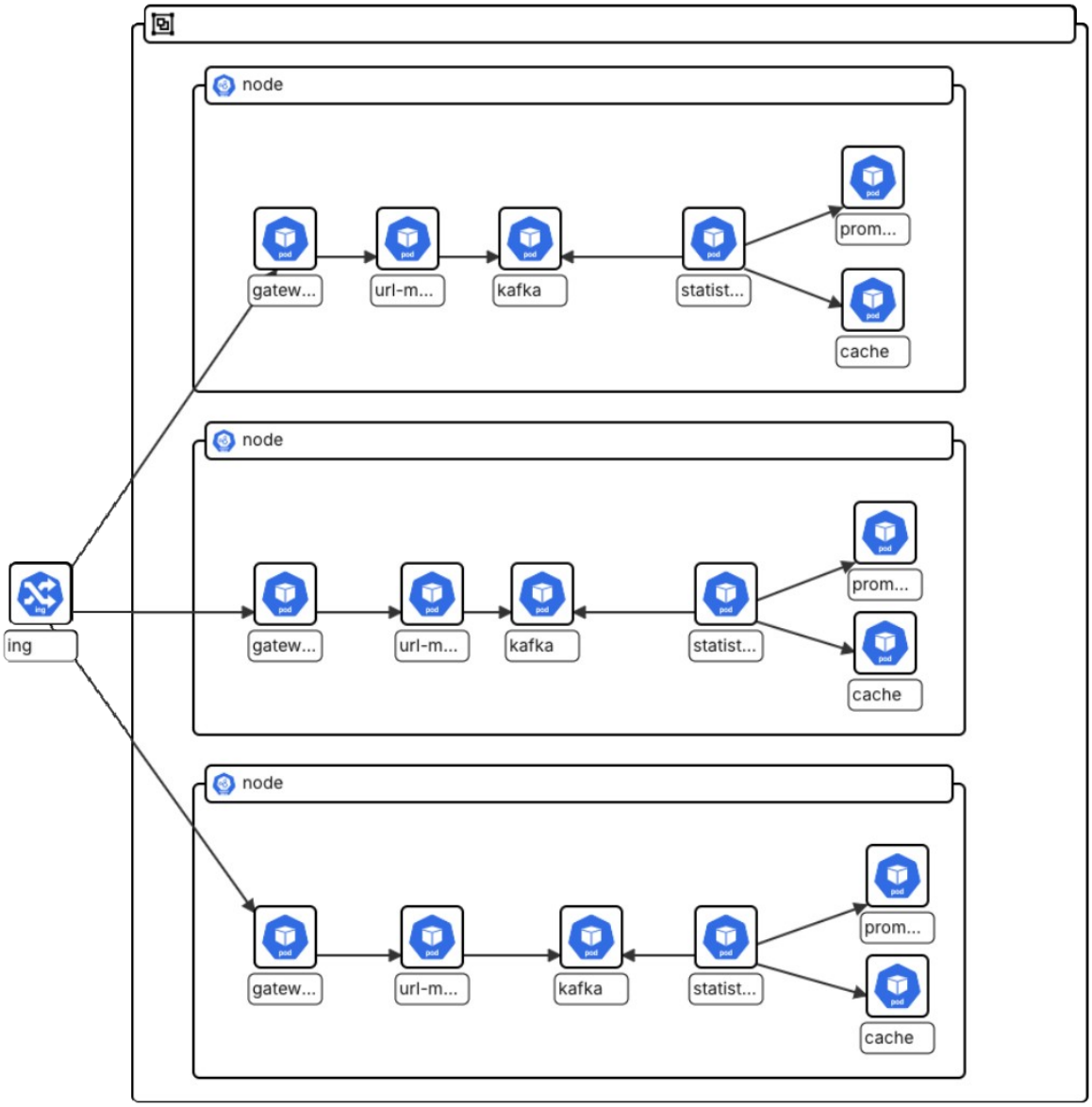
Para garantizar que la arquitectura de microservicios sea **escalable de forma eficiente** y pueda manejar el tráfico masivo con elasticidad, la propuesta incluye el uso de **Kubernetes** como plataforma de orquestación de contenedores. Kubernetes proporciona una manera flexible y eficiente de escalar automáticamente los microservicios según la demanda, manteniendo la estabilidad y la disponibilidad del sistema.

Se implementará un **LoadBalancer** que distribuya el tráfico entrante hacia el **API Gateway**. Este componente será responsable de gestionar el acceso a los microservicios y balancear la carga de manera eficiente. El uso de un LoadBalancer asegura que el tráfico se distribuya de manera equitativa entre las réplicas del **API Gateway**, garantizando una respuesta rápida y un manejo óptimo del tráfico, especialmente durante picos de tráfico.

Cada microservicio, tanto el **UrlManager** como el **StatisticsService**, se desplegará en pods independientes. Estos pods tendrán réplicas configuradas para escalar automáticamente según la carga del sistema. Separar los microservicios en pods independientes asegura que un fallo en un microservicio no afecte al resto del sistema.

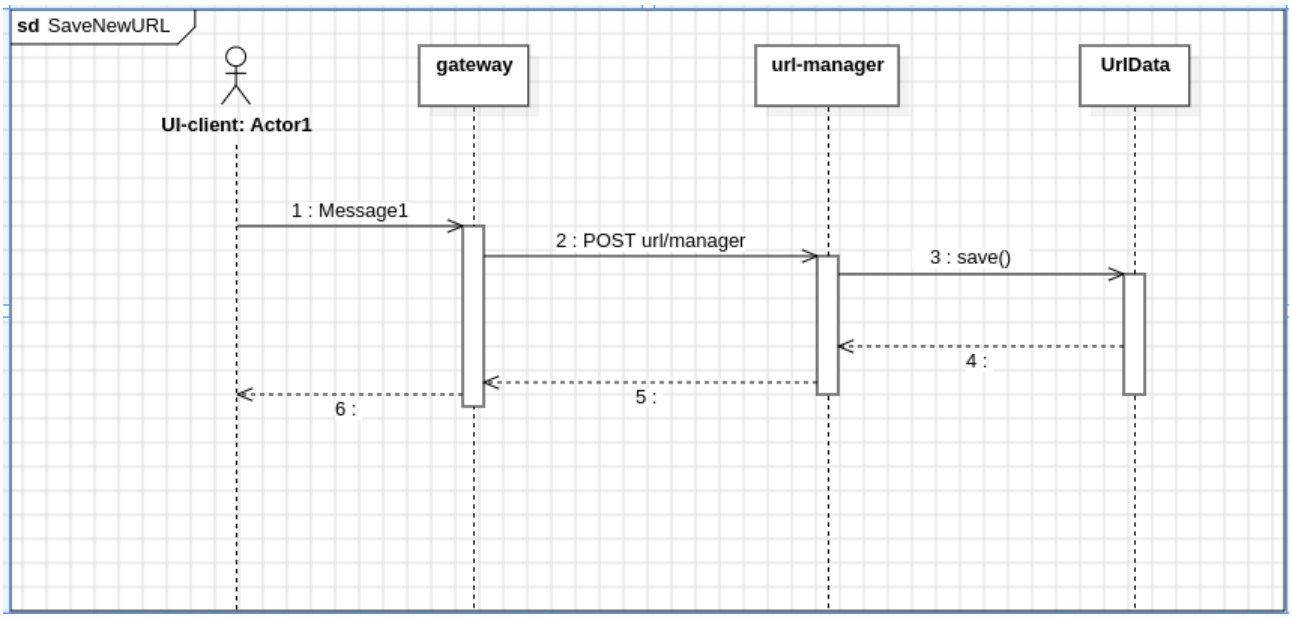
En un principio, seria ideal tener un Kafka cluster externo, al igual que el cache, pero como eso podria incrementar los costos y ademas es dificil teniendo en cuenta para la aplicación de esta demo, se opta por hacer el deploy dentro de un pod de K8s. Los servicios de Kafka y Redis estarán desplegados entonces en pods dentro de Kubernetes, con réplicas que aseguren la disponibilidad y el manejo eficiente de la comunicación asíncrona y el caching.

La base de datos **MongoDB** se desplegará externamente a Kubernetes. En el caso de Prometheus, podria estar securitizado mediante una **VPN** en un entorno externo para garantizar que solo los componentes autorizados puedan acceder a sus métricas. Al igual que antes, Mantener Prometheus fuera del clúster de Kubernetes y securitizado mediante una VPN tiene beneficios clave como mayor seguridad y carga reducida en el cluster, pero como complijiza la solucion para esta demo, se pondra el prometheus ejecutandose internamente en Kubernetes,



Conclusión

La arquitectura diseñada está alineada con los requisitos de **alta disponibilidad**, **escalabilidad**, y **bajo costo**. Las decisiones tecnológicas fueron tomadas tras una cuidadosa evaluación de las alternativas disponibles, priorizando la capacidad de manejar picos de tráfico y la eficiencia en la resolución de URLs y la recolección de estadísticas. La implementación basada en **Java 17 con WebFlux**, **microservicios desacoplados**, **MongoDB**, **Kafka**, **Redis**, y **Prometheus** asegura un rendimiento óptimo, flexibilidad para adaptarse a nuevos requisitos, y un fácil mantenimiento del sistema.


sd SaveNewURL
UI-client: Actor1
gateway
url-manager
UrlData
1 : Message1
2 : POST url/manager
3 : save()
4 :
5 :
6 :