**Documento de Diseño Arquitectónico – Sistema CONECTA**

**1. Introducción y Entendimiento de la Aplicación**

**1.1. Propósito y Visión del Sistema**

El sistema CONECTA ha sido concebido como un API Gateway centralizado y de nivel empresarial, diseñado para actuar como el único punto de entrada y salida para todo el tráfico de API de la organización. La visión fundamental de CONECTA trasciende la de un simple proxy inverso; se posiciona como una capa estratégica de gestión que introduce gobernanza, seguridad, control y observabilidad sobre los flujos de comunicación entre los servicios internos y los sistemas externos. En una arquitectura moderna, un API Gateway es un componente programático que se sitúa frente a las APIs para actuar como un punto de entrada único, protegiendo, asegurando y gestionando la escalabilidad y alta disponibilidad.1

CONECTA está diseñado para alinearse intrínsecamente con una arquitectura orientada a microservicios, un paradigma donde las aplicaciones se descomponen en servicios pequeños y débilmente acoplados.2 Al centralizar responsabilidades transversales (conocidas como

*cross-cutting concerns*), el sistema simplifica drásticamente la lógica de los microservicios individuales. Funcionalidades como la autenticación de usuarios, el enrutamiento de peticiones, la monitorización, la resiliencia y la auditoría se gestionan en CONECTA, liberando a los equipos de desarrollo de la necesidad de implementar estas lógicas repetidamente en cada servicio.3 Esto no solo acelera el tiempo de desarrollo, sino que también garantiza la aplicación consistente de políticas de seguridad y gobernanza en toda la organización.

La implementación de CONECTA como un API Gateway construido sobre el ecosistema reactivo de Spring (Spring Boot y Spring WebFlux) tiene como objetivo proporcionar una manera simple pero efectiva de enrutar APIs y aplicar estas funcionalidades transversales de manera eficiente y escalable.3

**1.2. Problema de Negocio y Contexto Operacional**

Actualmente, la organización enfrenta un desafío significativo en la gestión de su tráfico de red. Todo el flujo de datos, tanto entrante como saliente, debe transitar a través de un sistema IBM Datapower por mandatos de seguridad corporativa. El estado actual de la configuración implica la gestión de múltiples conexiones directas y reglas de enrutamiento individuales dentro de Datapower para cada servicio interno que necesita exponerse o consumir un servicio externo. Esta aproximación ha generado una topología de red compleja, rígida y costosa de mantener. Cada nuevo servicio o cambio en una integración existente requiere una intervención manual en Datapower, lo que introduce un cuello de botella operativo, ralentiza los ciclos de despliegue y aumenta la probabilidad de errores de configuración.

La introducción de CONECTA aborda directamente esta problemática. Arquitectónicamente, CONECTA se posicionará entre los clientes externos y el sistema IBM Datapower para el tráfico entrante, y entre los servicios internos y Datapower para el tráfico saliente. El resultado de este diseño es una simplificación radical de la configuración en Datapower. En lugar de gestionar cientos de reglas, Datapower solo necesitará mantener una única "tubería" segura y de alto rendimiento hacia y desde CONECTA.

Toda la lógica compleja de enrutamiento basado en contenido, validación de credenciales, transformación de mensajes y gestión de endpoints se traslada a CONECTA, una plataforma moderna, ágil y programáticamente configurable. Este cambio estratégico produce beneficios tangibles:

1. **Agilidad Operativa:** Reduce drásticamente la dependencia del equipo especializado en Datapower para tareas de configuración rutinarias.
2. **Empoderamiento de Equipos:** Otorga a los equipos de desarrollo la autonomía para exponer y consumir APIs de manera rápida y segura a través de la interfaz de administración de CONECTA, sin necesidad de esperar por cambios en la infraestructura central.
3. **Centralización y Consistencia:** Asegura que todas las políticas de seguridad, auditoría y enrutamiento se apliquen de manera uniforme desde un único punto de control, mejorando la postura de seguridad y la gobernanza general del ecosistema de APIs.

En esencia, CONECTA transforma el API Gateway de ser una simple necesidad de infraestructura a una herramienta estratégica que habilita la agilidad del negocio en un entorno digital en constante cambio.

**2. Análisis de Requisitos (Metodología MoSCoW)**

**2.1. Introducción a MoSCoW**

Para garantizar una planificación y ejecución del proyecto efectivas, los requisitos funcionales y no funcionales del sistema CONECTA se han priorizado utilizando la metodología MoSCoW. Este método clasifica los requisitos en cuatro categorías para establecer un entendimiento común sobre la importancia de cada funcionalidad y definir un alcance claro para las distintas fases de entrega, especialmente para el Producto Mínimo Viable (MVP).

* **Must-have (Debe tener):** Requisitos críticos para el éxito del proyecto. Sin ellos, el sistema no sería viable o no cumpliría su propósito fundamental. El MVP debe incluir todos los requisitos de esta categoría.
* **Should-have (Debería tener):** Requisitos importantes pero no vitales para el lanzamiento inicial. Su ausencia podría ser dolorosa, pero existen soluciones alternativas, aunque sean ineficientes. Se deben incluir si el tiempo y los recursos lo permiten.
* **Could-have (Podría tener):** Requisitos deseables pero menos importantes. Su impacto en el resultado final es menor en comparación con los *Must* y *Should*. A menudo se consideran mejoras que pueden añadirse en futuras iteraciones.
* **Won't-have (No tendrá):** Requisitos que se ha acordado explícitamente no implementar en el marco de tiempo actual del proyecto. Ayudan a clarificar el alcance y a evitar la "corrupción del alcance" (*scope creep*).

**2.2. Tabla de Requisitos MoSCoW**

La siguiente tabla detalla la clasificación de los requisitos para el sistema CONECTA.

| ID | Requisito | Categoría (MoSCoW) | Justificación de la Clasificación |
| --- | --- | --- | --- |
| **FN-01** | Enrutamiento dinámico de peticiones entrantes basado en un segmento de la URL. | **Must-have** | Es la funcionalidad central y la razón de ser de un API Gateway. Sin la capacidad de enrutar peticiones, el sistema carece de propósito. |
| **FN-02** | Validación de token JWT para todo el tráfico entrante desde sistemas externos. | **Must-have** | Requisito de seguridad explícito y no negociable. Proteger los servicios internos es una función primordial del gateway.4 |
| **FN-03** | Adición automática de un token JWT requerido en el tráfico saliente hacia sistemas externos. | **Must-have** | Requisito de seguridad explícito para la integración con sistemas de terceros que demandan autenticación. Centraliza la gestión de credenciales salientes. |
| **FN-04** | Auditoría completa de cada transacción, incluyendo la petición original y la respuesta del servicio. | **Must-have** | Requisito explícito para trazabilidad, depuración y cumplimiento normativo. Es una capacidad fundamental para un punto de control de tráfico.7 |
| **FN-05** | Permitir la adición o modificación de rutas sin necesidad de redesplegar la aplicación. | **Must-have** | Requisito clave para la agilidad operativa. Un redespliegue por cada cambio de ruta es inaceptable en un entorno empresarial dinámico y anularía los beneficios del sistema.9 |
| **FN-06** | Gestión transaccional de las operaciones para garantizar la integridad de los datos. | **Must-have** | Requerido explícitamente para asegurar la integridad, especialmente en la escritura de auditorías. Se refinará su significado para evitar bloqueos (ver sección 6.3). |
| **NFN-01** | Alta disponibilidad y escalabilidad horizontal para soportar un alto volumen de peticiones. | **Must-have** | Como punto único de entrada/salida, el gateway es un componente de misión crítica. Su fallo impactaría a toda la organización. La arquitectura debe ser inherentemente resiliente.11 |
| **UI-01** | Interfaz de usuario (UI) para que los administradores puedan consultar los logs de auditoría. | **Should-have** | Aunque la auditoría es un *Must-have*, la consulta inicial podría realizarse con herramientas externas (ej. Kibana, acceso a BD). Una UI dedicada mejora la operatividad y la autonomía del equipo de soporte, pero no es indispensable para el lanzamiento inicial si los recursos son limitados. |
| **UI-02** | Interfaz de usuario (UI) para que los administradores gestionen la configuración de enrutamiento. | **Must-have** | Es la contraparte indispensable del requisito FN-05. Sin una UI, la "configuración en caliente" se realizaría mediante llamadas API directas o manipulación de la base de datos, lo cual es propenso a errores y no es usable por administradores no técnicos.13 |
| **FN-07** | Integración con un sistema de métricas estándar (ej. Prometheus) para monitorización. | **Could-have** | No solicitado explícitamente, pero un gateway moderno debe exponer métricas de rendimiento (latencia, tasa de errores, etc.). Spring Boot Actuator facilita enormemente esta integración 15, mejorando la monitorización proactiva. Es una mejora de alto valor para futuras iteraciones. |
| **FN-08** | Soporte para mecanismos de autenticación adicionales, como API Keys o OAuth2. | **Won't-have** | El requisito es explícitamente sobre JWT. Añadir otros mecanismos introduce una complejidad considerable y está fuera del alcance de la versión inicial definida. El foco debe permanecer en implementar el flujo JWT de manera robusta. |

**3. Arquitectura de Alto Nivel y Descomposición del Sistema**

**3.1. Visión Arquitectónica General**

Para satisfacer los requisitos de flexibilidad, escalabilidad y mantenibilidad, el sistema CONECTA se diseñará siguiendo los principios de una **Arquitectura de Microservicios**. Aunque desde una perspectiva externa CONECTA funciona como un único punto de entrada lógico, su estructura interna se descompondrá en servicios independientes y especializados. Esta descomposición promueve una clara separación de responsabilidades, permite la escalabilidad independiente de sus componentes y facilita el mantenimiento y la evolución a largo plazo.

El patrón arquitectónico central que se implementará es el de **API Gateway**.1 La tecnología seleccionada para materializar este patrón es

**Spring Cloud Gateway**. Esta elección se fundamenta en sus características clave: es un framework reactivo, no bloqueante y construido sobre Spring WebFlux y Project Reactor, lo que lo hace ideal para manejar un alto volumen de conexiones concurrentes con un uso eficiente de los recursos.3 Su profunda integración con el ecosistema de Spring, incluyendo Spring Security y Spring Boot Actuator, simplifica la implementación de funcionalidades complejas como la seguridad y la monitorización.3

**3.2. Descomposición Modular**

El sistema CONECTA se descompondrá en cuatro subsistemas o módulos lógicos principales. Dependiendo de la estrategia de despliegue final, estos módulos pueden implementarse como microservicios independientes desplegados en contenedores o como módulos dentro de una aplicación monolítica modular. La arquitectura propuesta favorece la separación en microservicios para maximizar los beneficios de escalabilidad y resiliencia.

1. **Gateway Core (Spring Cloud Gateway):** Es el corazón del sistema y el único componente en la ruta crítica de las peticiones de los usuarios. Basado directamente en Spring Cloud Gateway, su única responsabilidad es el procesamiento de alta velocidad del tráfico. Se encarga del enrutamiento dinámico, la ejecución de la cadena de filtros (seguridad, auditoría, etc.) y el reenvío de peticiones a los servicios de destino. Está optimizado para un rendimiento máximo y una latencia mínima.
2. **Servicio de Configuración (Config Service):** Es un microservicio estándar de Spring Boot que expone una API RESTful. Su propósito es gestionar el ciclo de vida de las configuraciones del gateway (rutas, endpoints, políticas de seguridad). Persiste esta información en una base de datos dedicada y notifica al Gateway Core cuando se producen cambios. Es el backend para la sección de administración de la Interfaz de Usuario.
3. **Servicio de Auditoría (Audit Service):** Es un microservicio de Spring Boot con una API RESTful, diseñado para recibir y almacenar las trazas de auditoría. Para no impactar la latencia del Gateway Core, la recepción de trazas se realiza de forma asíncrona. Este servicio se encarga de persistir los logs en un sistema de almacenamiento optimizado para grandes volúmenes de escritura y búsquedas complejas (por ejemplo, una base de datos de documentos o un motor de búsqueda como Elasticsearch).
4. **Interfaz de Administración (Admin UI):** Es una Single-Page Application (SPA) desarrollada en Angular. Constituye la cara visible del sistema para los administradores. Se comunica exclusivamente a través de APIs REST con el Servicio de Configuración (para gestionar rutas) y el Servicio de Auditoría (para consultar logs). No tiene comunicación directa con el Gateway Core.

Esta separación de componentes es una decisión de diseño fundamental. Desacopla las tareas de gestión (configuración, consulta de logs), que no son sensibles a la latencia, de la tarea de procesamiento de tráfico, que es de misión crítica. El Gateway Core puede así permanecer ligero, rápido y seguro, enfocado únicamente en su función principal de enrutamiento. Esta arquitectura asegura que un pico de carga en la UI de administración o una operación de escritura lenta en la base de datos de auditoría no puedan degradar el rendimiento del flujo de datos principal del gateway.

**3.3. Diagrama de Módulos del Sistema**

El siguiente diagrama de componentes, generado con PlantUML, ilustra la descomposición del sistema CONECTA y las interacciones entre sus módulos, los actores y los sistemas externos.

Fragmento de código

@startuml

!theme plain

skinparam rectangle {

shadowing false

borderRadius 10

}

skinparam component {

borderColor #333

backgroundColor #f0f0f0

arrowColor #333

}

skinparam database {

borderColor #333

backgroundColor #lightgrey

}

skinparam actor {

borderColor #333

}

package "Sistema CONECTA" {

frame "Frontend" {

component "Admin UI (Angular)" as AdminUI

}

frame "Backend" {

component "Gateway Core\n(Spring Cloud Gateway)" as GatewayCore

component "Config Service\n(Spring Boot)" as ConfigService

component "Audit Service\n(Spring Boot)" as AuditService

}

}

database "Config DB\n(PostgreSQL/MongoDB)" as ConfigDB

database "Audit DB / Log Store\n(Elasticsearch/DB)" as AuditDB

actor "Administrador" as Admin

actor "Sistema Externo" as ExternalSystem

actor "Servicio Interno" as InternalService

Admin -right-> AdminUI : Gestiona Configuración y\nConsulta Logs

AdminUI.down.> ConfigService : Lee/Escribe Configuración de Rutas

AdminUI.down.> AuditService : Consulta Logs de Auditoría

ConfigService.down.> ConfigDB : CRUD de Rutas

AuditService.down.> AuditDB : Escritura de Trazas

GatewayCore.right.> ConfigService : Carga Configuración de Rutas\n(vía RouteDefinitionLocator)

GatewayCore.down.> AuditService : Envía Trazas de Auditoría

ExternalSystem -right-> GatewayCore : Petición Entrante (HTTPS)

GatewayCore -right-> InternalService : Petición Reenviada (HTTP)

InternalService -right-> GatewayCore : Petición Saliente (HTTP)

GatewayCore -right-> ExternalSystem : Petición Reenviada (HTTPS)

@enduml

**3.4. Estrategia de Alta Disponibilidad y Escalabilidad**

La arquitectura de CONECTA está diseñada para ser altamente disponible y escalable horizontalmente, un requisito indispensable para un componente de infraestructura crítico.

* **Escalabilidad Horizontal:** El componente principal, Gateway Core, es inherentemente sin estado (*stateless*). Esto significa que cualquier instancia del gateway puede procesar cualquier petición, lo que permite una escalabilidad horizontal sencilla. Para aumentar la capacidad del sistema, simplemente se añaden más instancias (réplicas) del Gateway Core.11
* **Balanceo de Carga Externo:** Se desplegarán múltiples instancias del Gateway Core detrás de un **Balanceador de Carga Externo** (por ejemplo, Nginx, HAProxy, AWS Application Load Balancer, Azure Load Balancer). Este balanceador de carga será el punto de entrada público para todo el tráfico y se encargará de distribuir las peticiones entrantes entre las instancias activas del gateway utilizando un algoritmo como Round Robin o Least Connections.11 Esta es la estrategia recomendada para entornos de producción, ya que desacopla la lógica de balanceo de la aplicación y utiliza componentes de hardware o software especializados y de alto rendimiento.
* **Comprobaciones de Salud (Health Checks):** Para garantizar la resiliencia, el balanceador de carga se configurará para realizar comprobaciones de salud periódicas a cada instancia del Gateway Core. Spring Boot Actuator expone un endpoint estándar, /actuator/health, que proporciona el estado de la aplicación. Si una instancia deja de responder o informa de un estado DOWN, el balanceador de carga la retirará automáticamente del grupo de enrutamiento, redirigiendo el tráfico a las instancias saludables restantes. Esto previene que las peticiones se envíen a una instancia fallida, asegurando una alta disponibilidad del servicio.11
* **Escalabilidad de Servicios de Soporte:** Los servicios de Configuración y Auditoría también están diseñados para ser sin estado y pueden escalarse horizontalmente si la carga de trabajo de administración o de ingesta de logs lo requiere. Sin embargo, se anticipa que su necesidad de escalabilidad será significativamente menor que la del Gateway Core, que maneja el tráfico en tiempo real.

Esta estrategia combinada asegura que CONECTA pueda manejar un alto volumen de tráfico, soportar fallos de instancias individuales sin interrumpir el servicio y crecer en capacidad de forma lineal añadiendo más recursos de computación.

**4. Diseño Detallado de Módulos y Flujos de Información**

Esta sección profundiza en el diseño interno de cada módulo, sus responsabilidades específicas y los flujos de datos que los interconectan.

**4.1. Módulo Gateway Core (Spring Cloud Gateway)**

Este módulo es el motor de procesamiento de peticiones. Su diseño se centra en la eficiencia, la extensibilidad a través de filtros y la configuración dinámica.

**4.1.1. Enrutamiento Dinámico desde Base de Datos**

Para cumplir con el requisito de configurar rutas sin redesplegar la aplicación [FN-05], el Gateway Core no almacenará las definiciones de ruta en un archivo estático application.yml. En su lugar, las cargará dinámicamente desde una fuente externa, en este caso, la base de datos de configuración.

* **Mecanismo de Implementación:** Se creará una implementación personalizada de la interfaz RouteDefinitionLocator de Spring Cloud Gateway.9 Esta interfaz tiene un único método,

getRouteDefinitions(), que debe devolver un Flux<RouteDefinition>, un flujo reactivo de definiciones de ruta. Nuestra clase, que podría llamarse DatabaseRouteDefinitionLocator, será un @Component de Spring.

* **Interacción con la Base de Datos:** Dentro de DatabaseRouteDefinitionLocator, se inyectará un repositorio de datos reactivo para comunicarse con la Config DB. Dado que Spring Cloud Gateway es una pila totalmente reactiva, es imperativo evitar operaciones de base de datos bloqueantes. Por lo tanto, se utilizará **R2DBC (Reactive Relational Database Connectivity)** si la Config DB es una base de datos SQL (como PostgreSQL) 19, o un driver reactivo como Spring Data Reactive MongoDB si es una base de datos NoSQL. Esto asegura que la consulta de rutas no bloquee el hilo de eventos de Netty.
* **Flujo de Carga Inicial:** Al arrancar la aplicación, Spring Cloud Gateway descubrirá nuestro DatabaseRouteDefinitionLocator y llamará a su método getRouteDefinitions(). Este método consultará la Config DB para obtener todas las rutas activas, las transformará en objetos RouteDefinition y las devolverá como un Flux. El gateway procesará este flujo y cargará las rutas en memoria para un acceso de alta velocidad.
* **Refresco en Caliente (Hot Reloading):** La actualización dinámica de rutas es un proceso crítico. La estrategia propuesta es la siguiente:
  1. El Admin UI realiza un cambio en una ruta, que se guarda en la Config DB a través del Config Service.
  2. Tras confirmar la escritura en la base de datos, el Config Service publica un evento, por ejemplo, ConfigChangedEvent, en un bus de mensajes (como RabbitMQ o Kafka) o utiliza Spring Cloud Bus.
  3. Todas las instancias del Gateway Core estarán suscritas a este bus de mensajes.
  4. Al recibir el evento, cada instancia del gateway activará un refresco de su caché de rutas. Esto se puede lograr programáticamente publicando un RefreshRoutesEvent en el contexto de la aplicación Spring, o invocando el endpoint de Actuator /actuator/gateway/refresh.20
  5. Esta acción de refresco provoca que Spring Cloud Gateway vuelva a invocar el método getRouteDefinitions() de nuestro DatabaseRouteDefinitionLocator, que recargará el conjunto completo de rutas desde la base de datos, aplicando los cambios de forma atómica y sin tiempo de inactividad.23

**4.1.2. Filtro de Seguridad JWT**

Para manejar los requisitos de seguridad [FN-02, FN-03], se implementará un GatewayFilterFactory personalizado. Crear una factoría en lugar de un GlobalFilter permite aplicar la lógica de seguridad selectivamente solo a las rutas que lo requieran, proporcionando mayor flexibilidad.

* **Implementación:** Se creará una clase JwtAuthGatewayFilterFactory que extienda AbstractGatewayFilterFactory. Esta factoría permitirá configurar el filtro en el application.yml o, en nuestro caso, en la definición de ruta almacenada en la base de datos. La lógica del filtro se inspira en las mejores prácticas de seguridad para gateways.4
* **Flujo de Tráfico Entrante (Validación):**
  1. El filtro intercepta la petición entrante.
  2. Busca el encabezado Authorization. Si no existe o no comienza con Bearer , la petición se rechaza inmediatamente con un estado HTTP 401 Unauthorized.
  3. Extrae el token JWT del encabezado.
  4. Utiliza las capacidades de Spring Security (spring-boot-starter-oauth2-resource-server) para validar el token. Se configura con el jwk-set-uri del servidor de autorización, desde donde obtiene las claves públicas para verificar la firma del token.4 También valida la expiración (exp) y otras reclamaciones estándar.
  5. Si la validación del token falla por cualquier motivo (firma inválida, token expirado), la petición se rechaza con un estado HTTP 401 Unauthorized.
  6. Si la validación es exitosa, el filtro puede, opcionalmente, mutar la petición para añadir encabezados adicionales que serán consumidos por los servicios internos, como X-User-Id o X-User-Roles, extraídos de las reclamaciones del token. Esto evita que cada microservicio tenga que decodificar el JWT.25
  7. La petición, ahora validada y potencialmente enriquecida, continúa hacia el siguiente filtro en la cadena.
* **Flujo de Tráfico Saliente (Inyección):**
  1. Para las rutas que representan llamadas desde un servicio interno hacia un sistema externo, se aplicará una configuración diferente de este mismo filtro.
  2. El filtro intercepta la petición saliente.
  3. Obtiene el token JWT necesario para el sistema externo. Este token puede estar preconfigurado en los metadatos de la ruta en la Config DB o ser obtenido dinámicamente de un servicio de gestión de credenciales.
  4. Añade el encabezado Authorization: Bearer <token> a la petición saliente.
  5. La petición continúa su curso hacia el sistema externo.

**4.1.3. Filtro Global de Auditoría**

Dado que la auditoría es un requisito para todas las transacciones [FN-04], se implementará como un GlobalFilter. Este filtro se ejecutará para cada petición que pase por el gateway.

* **Desafío Técnico y Solución:** El principal desafío en un entorno reactivo como WebFlux es que el cuerpo de la petición (Flux<DataBuffer>) es un flujo de datos que solo puede ser consumido una vez. Si el filtro de auditoría lee el cuerpo para registrarlo, este ya no estará disponible para ser enviado al servicio de destino, rompiendo la aplicación.27 La solución canónica es utilizar decoradores para "envolver" la petición y la respuesta, permitiendo la intercepción del cuerpo sin consumirlo de forma destructiva.30
* **Implementación y Flujo:**
  1. Se creará una clase AuditGlobalFilter que implemente GlobalFilter y Ordered, asignándole una alta precedencia para que se ejecute al principio de la cadena para la petición y al final para la respuesta.
  2. **Fase "Pre" (Petición):** El filtro envuelve el objeto ServerHttpRequest en un ServerHttpRequestDecorator personalizado. Este decorador sobreescribe el método getBody(). Cuando se invoca getBody(), el decorador se suscribe al flujo de datos original, lee los bytes en un buffer local (ej. ByteArrayOutputStream), y luego devuelve un nuevo Flux<DataBuffer> creado a partir de los bytes cacheados. Esto permite registrar el cuerpo de la petición mientras se preserva para el resto de la cadena de filtros.32 Se registran también los encabezados, el método, la URI y la IP del cliente.
  3. Se invoca chain.filter(exchange) para pasar la petición (con el decorador) al resto de la cadena.
  4. **Fase "Post" (Respuesta):** Utilizando operadores reactivos como then(Mono.defer(...)), el filtro intercepta la respuesta después de que ha sido procesada por el servicio de destino.
  5. De manera similar a la petición, se envuelve el ServerHttpResponse en un ServerHttpResponseDecorator. Este decorador sobreescribe el método writeWith(). Al ser invocado, intercepta el Publisher<? extends DataBuffer> del cuerpo de la respuesta, lee su contenido para el log, y luego lo pasa al flujo de escritura original para ser enviado al cliente.28 Se registran el código de estado y los encabezados de la respuesta.
  6. **Ensamblaje y Envío:** Una vez que se han capturado todos los datos (petición y respuesta completas), se ensambla un objeto de dominio AuditTrace.
  7. Este objeto AuditTrace se envía de forma asíncrona al Audit Service. Esto se puede hacer mediante una llamada HTTP no bloqueante con WebClient a un endpoint del Audit Service o publicando el objeto en una cola de mensajes (Message Queue). Este desacoplamiento asíncrono es vital para que la latencia de la escritura del log no afecte el tiempo de respuesta percibido por el cliente.34

**4.2. Servicio de Auditoría**

Este microservicio es el repositorio centralizado de todas las trazas de auditoría.

* **API REST:**
  + POST /api/audit/traces: Endpoint que recibe el objeto AuditTrace desde el Gateway Core. Para maximizar el rendimiento del gateway, este endpoint debe ser asíncrono (@Async en Spring) y responder inmediatamente con un 202 Accepted, indicando que la traza ha sido recibida para su procesamiento, pero sin esperar a que se complete la escritura en la base de datos.
  + GET /api/audit/traces: Endpoint utilizado por la Admin UI para consultar los logs. Debe ser robusto y soportar paginación, ordenación y filtrado por múltiples criterios: rango de fechas, ID de la ruta, código de estado de la respuesta, IP del cliente, o búsqueda de texto libre en los cuerpos de petición/respuesta.
* **Persistencia:** La elección del sistema de persistencia es crucial. Mientras que una base de datos relacional es posible, un sistema como **Elasticsearch** es altamente recomendado. Está optimizado para la ingesta de grandes volúmenes de datos de series temporales y proporciona capacidades de búsqueda de texto completo extremadamente potentes y rápidas, lo que es ideal para los requisitos de consulta de la Admin UI.34

**4.3. Servicio de Configuración**

Este microservicio actúa como la autoridad de configuración para el gateway.

* **API REST:** Expondrá una API CRUD (Crear, Leer, Actualizar, Borrar) completa para la gestión de las definiciones de ruta.
  + POST /api/config/routes: Crea una nueva definición de ruta.
  + GET /api/config/routes: Devuelve una lista paginada de todas las rutas configuradas.
  + GET /api/config/routes/{id}: Obtiene los detalles de una ruta específica por su ID.
  + PUT /api/config/routes/{id}: Actualiza una ruta existente.
  + DELETE /api/config/routes/{id}: Elimina una ruta.
* **Lógica de Negocio:** La responsabilidad más importante de este servicio, más allá del CRUD, es la orquestación del refresco de configuración. Después de cada operación de escritura (POST, PUT, DELETE) que modifique el estado de las rutas, el servicio debe publicar el evento ConfigChangedEvent al bus de mensajes para notificar a todas las instancias del Gateway Core que deben recargar su configuración.

**4.4. Interfaz de Administración (Angular)**

La Admin UI es la herramienta que permite a los administradores humanos interactuar con CONECTA.

* **Estructura:** Será una aplicación de página única (SPA) construida con Angular 18, organizada en módulos funcionales. La comunicación con los servicios de backend (Config Service y Audit Service) se realizará exclusivamente a través de HttpClient de Angular, consumiendo sus APIs REST.35
* **Módulo de Auditoría:**
  + **Componente** AuditLogViewer**:** Presentará los logs de auditoría en una tabla de datos interactiva (ej. usando ag-Grid o Angular Material Table). Incluirá controles de formulario para filtrar por rango de fechas, buscar por texto, y seleccionar por ID de ruta o código de estado. Implementará paginación del lado del servidor para manejar grandes volúmenes de datos de manera eficiente.
* **Módulo de Configuración de Rutas:**
  + **Componente** RouteListView**:** Mostrará una tabla con todas las rutas configuradas, con botones para Crear, Editar y Eliminar.
  + **Componente** RouteEditForm**:** Este será el componente más complejo. Se implementará como un **formulario dinámico** utilizando **Angular Reactive Forms**.36 El formulario permitirá a los administradores definir una

RouteDefinition completa:

* + - Campos básicos: ID de la ruta, URI de destino, orden de precedencia.
    - **Gestión dinámica de Predicados:** El usuario podrá añadir predicados de una lista predefinida (Path, Method, Header, Host, Cookie, etc.).3 Por cada predicado añadido, se renderizarán dinámicamente los campos de formulario necesarios para sus argumentos (ej. para

Path, un campo de texto para el patrón; para Header, dos campos para el nombre del encabezado y una expresión regular para el valor).

* + - **Gestión dinámica de Filtros:** De manera similar, el usuario podrá añadir filtros (StripPrefix, RewritePath, AddRequestHeader, JwtAuth, etc.) y configurar sus argumentos específicos.
    - Esta naturaleza dinámica es esencial para dar soporte a la flexibilidad de Spring Cloud Gateway sin tener que modificar el código del frontend.

**4.5. Flujos de Datos Principales (Diagramas de Flujo PlantUML)**

Los siguientes diagramas de secuencia ilustran los flujos de interacción más críticos del sistema.

**Flujo 1: Procesamiento de una Petición Entrante Válida**

Fragmento de código

@startuml

title Flujo de Procesamiento de Petición Entrante Válida

actor "Sistema Externo" as Client

participant "Balanceador de Carga" as LB

participant "Gateway Core" as GW

participant "Filtro Global\nde Auditoría" as AuditFilter

participant "Filtro de\nAutenticación JWT" as JwtFilter

participant "Motor de\nEnrutamiento" as Router

participant "Servicio Interno" as Backend

participant "Servicio de Auditoría" as AuditSvc

autonumber

Client -> LB : Petición HTTPS (con JWT en Header)

LB -> GW : Reenvía petición a una instancia

GW -> AuditFilter : Intercepta Petición

note right of AuditFilter : Envuelve la petición en un\n`ServerHttpRequestDecorator`\npara cachear el cuerpo.

AuditFilter -> GW : Pasa la petición decorada

GW -> JwtFilter : Intercepta Petición

note right of JwtFilter : Valida firma y expiración del JWT.\nSi es inválido, responde 401 y termina el flujo.

JwtFilter -> GW : Validación OK, pasa la petición

GW -> Router : Busca ruta coincidente en caché

Router -> Backend : Reenvía petición al servicio de destino

Backend --> Router : Respuesta del servicio

Router --> GW : Devuelve respuesta

GW -> AuditFilter : Intercepta Respuesta

note right of AuditFilter : Envuelve la respuesta en un\n`ServerHttpResponseDecorator`\npara leer el cuerpo.

AuditFilter -> AuditFilter : Ensambla objeto `AuditTrace` completo

AuditFilter -[#blue,dashed]-> AuditSvc : Envía Traza (Llamada Asíncrona)

AuditFilter --> GW : Pasa la respuesta decorada

GW --> LB : Devuelve respuesta al balanceador

LB --> Client : Devuelve respuesta final

@enduml

**Flujo 2: Actualización de una Ruta desde la Interfaz de Administración**

Fragmento de código

@startuml

title Flujo de Actualización de Ruta

actor "Administrador" as Admin

participant "Admin UI\n(Angular)" as UI

participant "Config Service\n(Spring Boot)" as ConfigSvc

database "Config DB" as DB

participant "Message Broker\n(RabbitMQ/Kafka)" as Broker

participant "Gateway Core\n(Instancia N)" as GW

autonumber

Admin -> UI : Modifica los detalles de una ruta y hace clic en "Guardar"

UI -> ConfigSvc : `PUT /api/config/routes/{id}` con datos de la ruta en JSON

ConfigSvc -> DB : Actualiza el registro de la ruta en la base de datos

DB --> ConfigSvc : Confirmación de escritura

ConfigSvc -> Broker : Publica evento `ConfigChangedEvent`

Broker --> ConfigSvc : Confirmación de publicación

ConfigSvc --> UI : Responde `200 OK`

UI -> Admin : Muestra notificación de éxito

group Notificación Asíncrona a Gateways

Broker -> GW : Entrega `ConfigChangedEvent` a todas las instancias suscritas

GW -> GW : Dispara `RefreshRoutesEvent` interno

GW -> ConfigSvc : `DatabaseRouteDefinitionLocator` vuelve a consultar las rutas

note right of GW : Las nuevas reglas de enrutamiento\nse aplican en memoria sin reinicio.

end

@enduml

**5. Diseño del Dominio y Modelo de Datos**

El diseño del dominio define las entidades clave del sistema y sus relaciones. Este modelo servirá como base para la implementación de las clases de persistencia, los objetos de transferencia de datos (DTOs) en las APIs REST y los modelos en el frontend.

**5.1. Objetos de Dominio Clave**

* ApiRoute: Es la representación central de una regla de enrutamiento en CONECTA. Corresponde directamente al objeto RouteDefinition de Spring Cloud Gateway. Contiene toda la información necesaria para que el gateway decida si una petición coincide con esta ruta y cómo procesarla.
  + id: Un identificador único y legible para la ruta (ej. servicio-clientes-v1).
  + uri: El URI del servicio de destino al que se reenviará la petición (ej. lb://servicio-clientes o https://api.externa.com).
  + order: Un entero que define la prioridad de la ruta. Las rutas con un valor de order más bajo se evalúan primero. Es crucial para resolver ambigüedades cuando una petición podría coincidir con múltiples rutas.
  + metadata: Un mapa de pares clave-valor para almacenar información adicional sobre la ruta, que puede ser utilizada por filtros personalizados.
  + predicates: Una lista de objetos PredicateDefinition que deben cumplirse para que la ruta se active.
  + filters: Una lista de objetos FilterDefinition que se aplican a la petición si la ruta coincide.
* PredicateDefinition: Representa una condición de coincidencia.
  + name: El nombre de la factoría de predicados de Spring Cloud Gateway (ej. Path, Method, Header).
  + args: Un mapa de argumentos para el predicado (ej. para Path, la clave es pattern y el valor es /clientes/\*\*).
* FilterDefinition: Representa una acción a realizar sobre la petición o la respuesta.
  + name: El nombre de la factoría de filtros (ej. StripPrefix, RewritePath, JwtAuth).
  + args: Un mapa de argumentos para el filtro (ej. para StripPrefix, la clave es parts y el valor es 1).
* AuditTrace: Modela una entrada de auditoría completa para una única transacción.
  + traceId: Un identificador único universal (UUID) para la traza.
  + routeId: El ID de la ApiRoute que procesó la petición.
  + requestMethod, requestUri, clientIp: Información básica de la petición.
  + responseStatusCode: El código de estado HTTP de la respuesta final.
  + processingTimeMs: El tiempo total en milisegundos que tardó el gateway en procesar la petición.
  + requestHeaders, requestBody: Los encabezados y el cuerpo de la petición, almacenados como texto (potencialmente truncados o sanitizados).
  + responseHeaders, responseBody: Los encabezados y el cuerpo de la respuesta.
  + timestamp: La marca de tiempo (con zona horaria) de cuándo se procesó la petición.

**5.2. Diagramas de Clases de Dominio (PlantUML)**

El siguiente diagrama de clases visualiza la estructura y las relaciones entre los objetos de dominio clave.

Fragmento de código

@startuml

title Diagrama de Clases del Dominio CONECTA

skinparam class {

ArrowColor #333

BorderColor #333

BackgroundColor #FFFFF0

}

class ApiRoute {

-String id

-String uri

-int order

-Map<String, Object> metadata

+getPredicates(): List<PredicateDefinition>

+getFilters(): List<FilterDefinition>

}

class PredicateDefinition {

-String name

-Map<String, String> args

+getName(): String

+getArgs(): Map<String, String>

}

class FilterDefinition {

-String name

-Map<String, String> args

+getName(): String

+getArgs(): Map<String, String>

}

class AuditTrace {

-UUID traceId

-String routeId

-String requestMethod

-String requestUri

-String clientIp

-int responseStatusCode

-long processingTimeMs

-String requestHeaders

-String requestBody

-String responseHeaders

-String responseBody

-ZonedDateTime timestamp

}

ApiRoute "1" \*-- "0..\*" PredicateDefinition : contains

ApiRoute "1" \*-- "0..\*" FilterDefinition : contains

note right of ApiRoute

Representa una `RouteDefinition` de Spring Cloud Gateway.

Se persiste en la base de datos de configuración.

end note

note right of AuditTrace

Representa una transacción completa a través del gateway.

Se persiste en el almacén de auditoría (ej. Elasticsearch).

end note

@enduml

**5.3. Esquema de la Base de Datos**

El diseño de la persistencia debe reflejar la estructura de los objetos de dominio, eligiendo las tecnologías adecuadas para cada caso de uso.

* **Base de Datos de Configuración (**Config DB**)**
  + **Tecnología Recomendada:** PostgreSQL con R2DBC. PostgreSQL ofrece soporte nativo para tipos de datos JSONB, lo cual es ideal para almacenar las estructuras flexibles y anidadas de los predicados y filtros sin necesidad de complejas normalizaciones de tablas.
  + **Tabla:** gateway\_routes
    - route\_id (VARCHAR(255), PRIMARY KEY): El ID único de la ruta.
    - uri (VARCHAR(1024), NOT NULL): El URI de destino.
    - route\_order (INT, NOT NULL, DEFAULT 0): La prioridad de la ruta.
    - predicates (JSONB, NOT NULL): Un array JSON que contiene los objetos PredicateDefinition.
    - filters (JSONB, NOT NULL): Un array JSON que contiene los objetos FilterDefinition.
    - metadata (JSONB): Un objeto JSON para metadatos adicionales.
    - is\_active (BOOLEAN, NOT NULL, DEFAULT true): Un flag para habilitar o deshabilitar la ruta.
    - created\_at (TIMESTAMPTZ, NOT NULL, DEFAULT NOW()): Fecha de creación.
    - updated\_at (TIMESTAMPTZ, NOT NULL, DEFAULT NOW()): Fecha de última modificación.
* **Almacén de Auditoría (**Audit DB / Log Store**)**
  + **Tecnología Recomendada:** Elasticsearch. Su capacidad para indexar y buscar en documentos JSON complejos a alta velocidad lo convierte en la opción superior para almacenar y consultar logs de auditoría.
  + **Índice:** audit-traces-\* (patrón de índice basado en tiempo, ej. audit-traces-2024-05)
    - **Documento** AuditTrace **(Mapeo de campos):**
      * traceId: keyword
      * routeId: keyword
      * requestMethod: keyword
      * requestUri: text (para búsqueda de texto completo)
      * clientIp: ip
      * responseStatusCode: integer
      * processingTimeMs: long
      * requestHeaders: object (mapeo dinámico)
      * requestBody: text
      * responseHeaders: object (mapeo dinámico)
      * responseBody: text
      * @timestamp: date (campo estándar de Elasticsearch para series temporales)

Este diseño de persistencia dual aprovecha la tecnología más adecuada para cada tipo de dato, garantizando tanto la flexibilidad en la configuración como el alto rendimiento en la auditoría.

**6. Consideraciones Tecnológicas y Decisiones de Diseño**

Esta sección finaliza el documento arquitectónico detallando las decisiones clave relacionadas con la pila tecnológica y abordando interpretaciones matizadas de los requisitos.

**6.1. Contrato de API REST**

La comunicación entre los diferentes componentes del sistema CONECTA se estandarizará mediante APIs RESTful que utilizan JSON como formato de intercambio de datos. Específicamente:

* **Frontend-Backend:** La Admin UI (Angular) se comunicará exclusivamente con el Config Service y el Audit Service a través de sus APIs REST. Este desacoplamiento es fundamental en las arquitecturas de microservicios y SPAs, ya que permite que el frontend y los servicios de backend evolucionen de forma independiente.35
* **Inter-Servicios:** La comunicación entre el Gateway Core y los servicios de Configuración y Auditoría también se realizará a través de APIs REST (o, alternativamente, a través de un bus de mensajes para eventos asíncronos).

Este enfoque basado en APIs bien definidas asegura la interoperabilidad y la claridad en las responsabilidades de cada componente.

**6.2. Justificación del Stack Tecnológico**

La selección de la pila tecnológica se ha realizado para cumplir con los requisitos funcionales y no funcionales, especialmente los de alto rendimiento y escalabilidad.

* **Spring Cloud Gateway:** Se elige sobre otras alternativas (como el antiguo Zuul 1) por su arquitectura reactiva y no bloqueante, construida sobre Project Reactor y Netty.3 Esto es esencial para un componente de E/S intensiva como un API Gateway, ya que le permite manejar miles de conexiones concurrentes con un número limitado de hilos, resultando en un menor consumo de memoria y una mayor capacidad de respuesta bajo carga. Su integración nativa con Spring Boot Actuator para métricas y gestión 15 y Spring Security para la protección de endpoints es una ventaja decisiva.
* **Angular:** Como framework para la Admin UI, Angular (específicamente la versión 18) proporciona una estructura robusta y escalable para construir aplicaciones complejas de una sola página (SPA). Su potente sistema de **Reactive Forms** es particularmente adecuado para el requisito de crear formularios dinámicos para la gestión de rutas, predicados y filtros, permitiendo la construcción de una interfaz de usuario flexible y potente que puede adaptarse a la adición de nuevos tipos de predicados o filtros en el backend sin requerir cambios en el frontend.37
* **R2DBC (Reactive Relational Database Connectivity):** Para mantener la pila tecnológica del Gateway Core completamente reactiva de extremo a extremo, la comunicación con la base de datos de configuración debe ser no bloqueante. Si se opta por una base de datos relacional como PostgreSQL, el uso de R2DBC en lugar del tradicional JDBC es mandatorio.19 JDBC es una API bloqueante, y su uso en el

Gateway Core crearía un cuello de botella que anularía los beneficios de la arquitectura reactiva de Spring Cloud Gateway.

**6.3. Interpretación de "Transaccionalidad"**

El requisito [FN-06] solicita que "todas las operaciones de enrutamiento y reenvío de mensajes se gestionen de manera transaccional para garantizar la integridad de los datos". En un sistema distribuido y de alto rendimiento como un API Gateway, la interpretación de "transaccional" debe ser matizada y adaptada al contexto para no introducir anti-patrones.

Una interpretación literal que implique una transacción distribuida (ej. usando un coordinador de transacciones de dos fases o 2PC) que abarque la llamada al servicio de backend y la escritura en la base de datos de auditoría es inviable. Tal enfoque introduciría un acoplamiento síncrono y bloqueante, aumentando drásticamente la latencia y creando un punto único de fallo masivo. Si la base de datos de auditoría estuviera lenta o no disponible, todo el tráfico del gateway se detendría.

Por lo tanto, la "integridad" en este contexto se redefine como **fiabilidad y atomicidad de la auditoría**, no como una transacción ACID tradicional. El objetivo es garantizar que cada petición que atraviesa el gateway sea auditada de forma fiable y que la traza de auditoría (que incluye tanto la petición como su correspondiente respuesta) se registre como una unidad atómica, sin que se pierdan datos.

Para lograr esto, se implementará un patrón de **"Auditoría Atómica Desacoplada"**:

1. El GlobalFilter de auditoría captura la petición y la respuesta completas en memoria, como se describió en la sección 4.1.3.
2. Una vez que el objeto AuditTrace está completamente ensamblado, se envía a un mecanismo de entrega asíncrono y fiable. La opción más robusta es una **cola de mensajes persistente** (como RabbitMQ o Apache Kafka). El Gateway Core publica el mensaje AuditTrace en la cola.
3. La respuesta al cliente original solo se libera *después* de que el broker de mensajes haya confirmado la recepción del mensaje de auditoría (un ack). Esto garantiza que la traza no se pierda incluso si la instancia del gateway falla inmediatamente después.
4. El Audit Service actúa como un consumidor de esta cola, procesando las trazas a su propio ritmo y escribiéndolas en el almacén de datos final (Elasticsearch).

Este diseño logra la integridad requerida:

* **Atomicidad:** La petición y la respuesta se registran juntas en un único mensaje de auditoría.
* **Fiabilidad (Durabilidad):** El uso de una cola de mensajes persistente garantiza que las trazas no se pierdan en tránsito.
* **Desacoplamiento y Rendimiento:** La latencia del proceso de escritura en la base de datos de auditoría está completamente desacoplada del ciclo de vida de la petición del cliente, preservando el alto rendimiento y la baja latencia del Gateway Core.

Esta reinterpretación de la transaccionalidad es una decisión arquitectónica clave que alinea los requisitos funcionales con las realidades no funcionales de un sistema reactivo de misión crítica.