E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

API Manager – KrakenD

Grado en Ingeniería Informática

Trabajo Fin de Grado

Asier Alba Osambela

Jesús Villadangos Alonso

Pamplona, 05/06/2023

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Agradecimientos

Han pasado 5 años desde que comencé el grado en ingeniería informática y empecé a adentrarme en el mundo del desarrollo software, mundo que me va a acompañar en mi día a día.

En esta época he crecido mucho en todos los aspectos tanto mentalmente como personalmente. Con este informe cierro una etapa cargada de emociones y que guardare con mucho cariño.

Por ello me parece apropiado comenzar agradeciendo a todas las personas que me han acompañado en esta aventura y a los que sé que me seguirán acompañando de aquí en adelante.

Primero agradecer a mis padres y a mi pareja, Angel Alba, Ana Osambela y María Oloriz por apoyarme en los momentos más difíciles y darme ese empujón para seguir adelante y no abandonar.

A continuación, a todos mis compañeros de trabajo por tenderme su mano con cualquier cosa que pudiera necesitar para realizar este informe. Principalmente a Iker mi mentor en la empresa en la que trabajo, a Javier Echeverria o Etxebe, mi compañero de trabajo y con el que comparto mi día a día en este mundo del desarrollo software y por último a David Gil, CTO del proyecto en el que estoy actualmente y el que me sugirió la idea.

Y por último a Jesús, mi tutor por darme la oportunidad de estructurar este informe de la manera que mejor me pareciese a mí y darme esa flexibilidad a la hora de realizarlo.

¡ Gracias a todos

Resumen

Este informe consiste en un proyecto de investigación acerca del aporte de valor de unAPI Gateway y lo que este aporta sobre el uso de distintas APIs en una arquitectura de microservicios.

El trabajo consiste en un proyecto web que será sobre el que me basaré para realizar la migración de sus endpoints a KrakenD, como si de una aplicación dentro de una arquitectura de servicios se tratase.

Esta aplicación consta de un backend diseñado siguiendo un modelo de buenas prácticas para el desarrollo del software.

Además, también he diseñado un proyecto web haciendo uso de KrakenD que esnuestro API Gateway y protagonista de este informe.

El trabajo se ha desarrollado mediante el uso del framework Laravel usando PHP como lenguaje de desarrollo [1]. Junto con ello se ha creado dos repositorios de GitHub donde tengo todo el código de los dos proyectos web. Por otro lado, he utilizado Docker para virtualizar las imágenes de las aplicaciones y así poder trabajar con ellas como si se tratase de un servidor de producción [2].

Cabe recalcar que este informe lo he redactado a modo de manual detallado para que se pueda usar como referencia para una empresa que vaya a empezar a montar un proyecto KrakenD. Además de tomarse como referencia cuando surja alguna duda sobre como montar un proyecto KrakenD.

Guion

INTRODUCCIÓN

Para introducir este informe primero me gustaría definir algunos conceptos sobre los que voy a estar hablando repetidamente:

¿Qué es un API? [3]

El termino API es una abreviación de Application Programming Interface o Interfaz de Programación de Aplicaciones. Las APIs son mecanismos que permiten que dos componentes de software se puedan comunicar entre si mediante un conjunto de definiciones y protocolos.

Hay distintos tipos de APIs, sin embargo, me he centrado exclusivamente en las API REST. Estas APIs trabajan de manera en la que el cliente envía una solicitud al servidor como datos. El servidor utiliza estos datos para iniciar unas funciones internas y devuelve los datos de salida al cliente que ha realizado la petición.

Una API web es una interfaz de procesamiento de aplicaciones entre un servidor web yun navegador web. La API REST es un tipo especial de API web.

Un API Manager (Administrador de API) y un API Gateway (Pasarela de API) son dos componentes claves en la gestión de APIs. Aunque se suelen utilizar justos, desempeñan roles diferentes en la gestión y seguridad de las APIs.

¿Qué es un API Manager o API Management?

El API Management es la práctica que se encarga de gestionar interfaces de programación de aplicaciones (APIs) a menudo utilizando software escalable para la creación, publicación, seguridad, monitoreo y análisis de APIs. El management nos permite monitorizar el ciclo de vida de la interfaz y asegurarse de que la API ofrezca el rendimiento para el cual fue diseñada.

¿Qué es un API Gateway? [4]

El proceso de integrar un API no es sencillo ya que debemos asegurarlo, monitorear como se usa y hacer modificaciones con los servicios. Para ello utilizaremos una API Gateway.

Un API Gateway es un gestor que recoge todas las llamadas a las APIs y actúa como un proxy inverso, recuperando en nombre de un cliente externo recursos de nuestras aplicaciones internas o backends. Posteriormente estos recursos se devuelven al cliente como si estos se originasen en el propio servidor en el que se ha realizado la petición a API.

Un API Gateway se encarga de desacoplar la interfaz por la que el cliente realiza la petición y la implementación del backend ya sea un entorno de aplicaciones o una arquitectura de microservicios.

En resumen, un API Manager se enfoca en la gestión completa del ciclo de vida de las APIs, mientras que un API Gateway se encarga de la gestión del tráfico y la seguridad en el punto de entrada de las APIs. Ambos son componentes esenciales para una arquitectura de APIs eficiente y segura.

Una vez que ya he definido varios de los conceptos clave de este informe, podemos seguir adelante con este informe.

KrakenD es un API Gateway creado para conseguir simplificar el proceso de creación y escalabilidad de microservicios. KrakenD actúa como una capa intermedia entre el cliente externo y nuestros microservicios.

Pero ¿por qué tiene sentido aplicar un API Gateway en un proyecto?

Hay múltiples razones o beneficios que motivan a utilizar un API Gateway en una arquitectura de microservicios, como, por ejemplo:

- Seguridad, gracias a ella podremos proteger los servicios web finales y proporciona autenticación y autorización de usuarios.

- Escalabilidad, podremos manejar el tráfico y distribuirlo por los servicios de nuestraarquitectura.

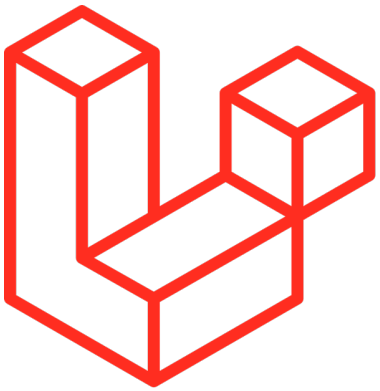
- Monitoreo, puede proporcionar una forma sencilla de monitoreo del tráfico entrante y saliente de las aplicaciones. Haciendo más sencillo el análisis.

- Simplificación de la arquitectura dejando un punto de entrada único para sus servicios web, facilitando la gestión y el mantenimiento.

En general, una API Gateway mejora la seguridad, escalabilidad y monitoreo además dela simplificación de la arquitectura.

HERRAMIENTAS

Laravel



Laravel es un framework de aplicaciones web que consta de una sintaxis expresiva y elegante. Proporciona una estructura y un punto de partida para la creación de una aplicación. Lo que permite centrarse plenamente en la creación de tu proyecto obviando los detalles ya que éstos están cubiertos por el propio framework. [5]

Laravel proporciona una ayuda extra al desarrollador como la inyección de dependencias afondo, una capa de abstracción de base de datos
expresiva, pruebas unitarias y de integración, entre otros.

Tanto si eres nuevo en los frameworks web PHP como si ya eres un experto, Laravel te ayuda a seguir creciendo ya sea ayudándote a dar los primeros pasos, así como ayudándote a elevar tu experiencia al siguiente nivel. [6]

Passport [7]

Laravel Passport proporciona una implementación completa del servidor OAuth2 parasu aplicación Laravel. Passport está construido sobre el servidor League OAuth2. [8]

Migration

Migration es una herramienta de Laravel que permite crear tablas directamente en nuestra base de datos. También nos permite modificar las tablas añadiendo o eliminandonuevas columnas de manera que nos resulte más sencillo realizar cambios en nuestras bases de datos.

Sail

Laravel Sail es una interfaz de línea de comandos ligera para interactuar con el entornode desarrollo Docker por defecto de Laravel. Sail proporciona un gran punto de partidapara la construcción de una aplicación Laravel utilizando PHP, MySQL y Redis sin necesidad de experiencia previa en Docker.

PHP

PHP es un lenguaje de programación que permite el desarrollo web o aplicaciones web dinámicas, el cual es apto para incrustar HTML. Además, que favorece la conexión entre el servidor y la interfaz del usuario. [9]

PHP Unit

PHPUnit es un entorno para realizar tests unitarios en el lenguaje de programación PHP. Este se creó con la idea de detectar cuanto antes los errores en el código para que puedan ser corregidos antes. Este conocido framework para PHP nos permite crear y ejecutar baterías de test unitarios de manera sencilla. Como todos los frameworks de test unitarios PHPUnit utiliza aserciones para verificar que el SUT es el deseado. [10]

Mockery

Mockery es un framework de dobles de tests de PHP para su uso en test unitarios con PHPUnit. Mockery nos ayuda a escribir tests usando objetos simulados para imitar el comportamiento de objetos reales de una manera controlada. [11]

Composer

Composer es un manejador de paquetes para PHP que proporciona un estándar para administrar, descargar e instalar dependencias y librerías. Composer es la solución ideal cuando trabajamos en proyectos complejos que depende de múltiples fuentes de instalación. [12]

Grum

Grum es un plugin de Composer que registra algunos git hooks en el repositorio del paquete. Cuando alguien confirme cambios, GrumPHP ejecutará algunas pruebas en el código conformado. Si las pruebas fallan no se podrán confirmar los cambios. GrumPHP no solo ayudará a mejorar el código base, sino que te ayudará a escribir código de calidad gracias a seguir las mejores prácticas que se hayan definido. [13]

Docker

Docker es un proyecto que automatiza los despliegues de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatizaciónde virtualización de aplicaciones en múltiples sistemas operativos. [14]

GitHub

GitHub es una plataforma de desarrollo colaborativo para alojar proyecto utilizando el sistema de control de versiones Git.
He decidido usar esta plataforma para los dos repositorios que conforman mi proyecto.

Git

Git es un software de control de versiones pensando en la eficiencia, la confiabilidad y compatibilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos a nivel de código. Su propósito es llevar registro de los cambios en los archivos de un proyecto incluyendo coordinar el trabajo que varias personas realizan sobre archivos compartidos en un repositorio de código como puedeser GitHub.

Pull Requests

Las pull Requests son solicitudes de código a un repositorio. Cuando el usuario ha editado el proyecto puede enviar la solicitud al repositorio a modificar. De esta forma secreará una solicitud en la pestaña de pull Requests. Una vez que una persona se ponga arevisar la pull request podrá decidir si acepta, deniega o pide ciertos cambios sobre el código nuevo. Adicionalmente, pueden revisar el código y enviar comentarios a modo de retroalimentación. Este proceso enriquece enormemente al equipo debido a que al revisar el código de los compañeros podemos encontrar posibles errores y mejoras además de que evitamos que solo una persona conozca ciertas partes del código. Aunque este proyecto lo he realizado de manera individual, he realizado igualmente las PR auto corrigiéndomelas.

GitHub Actions

GitHub Actions es una plataforma de integración y despliegue continuos (CI/CD) que permite automatizar tu mapa de compilación, pruebas y despliegue. Puedes crear flujosde trabajo y crear y probar cada solicitud de cambios en tu repositorio o desplegar solicitudes de cambios fusionadas a producción.

KARATE

Karate es la única herramienta de código abierto que combina automatización de pruebas de API, mocks, pruebas de rendimiento e incluso automatización de interfaz de usuario en un único marco unificado. La sintaxis es neutra en cuanto al lenguaje y fácil incluso para los no programadores. Las aserciones y los informes HTML están incorporados, y puede ejecutar pruebas en paralelo para mayor velocidad. [15]

CRYPTOWALLET

Introducción al proyecto base

Antes de adentrarme en la temática central de este informe el API Gateway KrakenD, me gustaría contextualizar el proyecto técnico que he realizado con el objetivo de proporcionar un enfoque práctico para comprender mejor las herramientas y tácticas que se explicarán durante el informe.

He creado un proyecto técnico que trata sobre una simulación de una cartera de crypto-monedas. En este proyecto he implementado múltiples endpoints definidos por dos contextos.

Cartera de crypto-monedas:

- Abrir una cartera nueva donde poder almacenar las crypto-monedas compradas.

- Obtener el balance de esa cartera, es decir ver el total de valor que tiene esa cartera.

- Cerrar una cartera ya existente donde eliminaremos todas las monedas que existieran en dicha cartera.

Crypto-Moneda:

- Un endpoint donde podamos obtener los datos más recientes de la crypto-moneda, siendo filtrada por su coin_id y obtener un json con los datos más importantes de esa moneda como su precio, nombre completo o puesto en el que se encuentra esa moneda por su valor.

- Comprar una moneda, en este endpoint se le deberá pasar en la request la id de la cartera donde almacenar la moneda que vas a comprar, el id de la moneda a comprar y la cantidad que quieres adquirir.

- Vender una moneda, en este endpoint se le deberá pasar en la request la id de la cartera de donde extraeremos la moneda que vamos a vender, el id de la moneda a vender y la cantidad que quieres retirar de la cartera. Además, he considerado todos los corner cases por lo que, si intentas vender una cantidad superior de la que realmente tienes en la cartera, el sistema fallará.

Técnicas de desarrollo

Infraestructura

Aprovechando que tenía que realizar un proyecto técnico para después poder aplicar mi API Gateway en un marco práctico y aun no siendo la idea principal de mi informe, decidí realizar este proyecto técnico para poner de manifiesto la importancia de elaborarun proyecto haciendo uso de buenas prácticas en el desarrollo de software por medio de

- Testing
- Validadores de código -> Precommit
- Clean code
- Integración continua

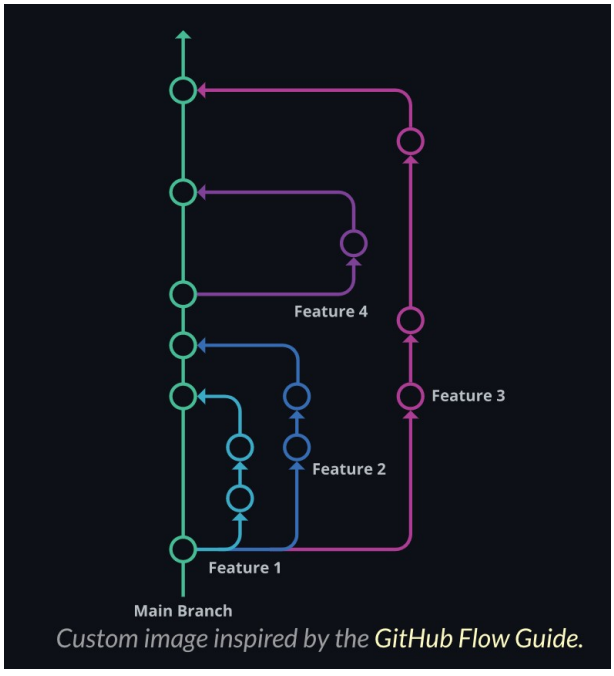
Además de todas ellas he estado trabajando con git para el control de versiones y mediante PRs o Pull Requests.

Aunque en un entorno más real las PRs se usan para que compañeros de equipo puedan revisar el código que se desea incluir en la rama principal y decidir si se acepta se deniega o se piden cambios a dicha inclusión.

Y aunque este proyecto haya sido desarrollado de manera individual he implementadola solución mediante PRs que me autocorregía.
Aunque parezca contradictorio ya que yo mismo iba creando el código cuando me las revisaba encontrada fallos que tenía y que si no lo habría realizado quizá no los hubiera encontrado.

Para el desarrollo del código no he seguido una estrategia de trabajar directamente sobrela rama master (rama de producción), sino que he seguido la GitHub Flow Branch Strategy. [16]

La estrategia de bifurcación de flujo de GitHub hace la forma de trabajar relativamente más sencilla ya que permite agilizar el trabajo.
La rama principal ha de tener el código listo para producción.
Y el resto de las ramas deben tener el nuevo trabajo de correcciones de errores o nuevas implementaciones, que serán fusionadas a la rama principal cuando el trabajo este debidamente acabado y revisado.



Hay seis principios que debes de cumplir:

1. Cualquier código de la rama principal ha de ser desplegable.

2. Crear nuevas ramas con nombre descriptivos.

3. Crear commits amenudo e ir haciendo push regularmente.

4. Para solicitar comentarios cuando creas que tu trabajo está listo para fusionarse ala rama principal abre una PULL REQUEST.

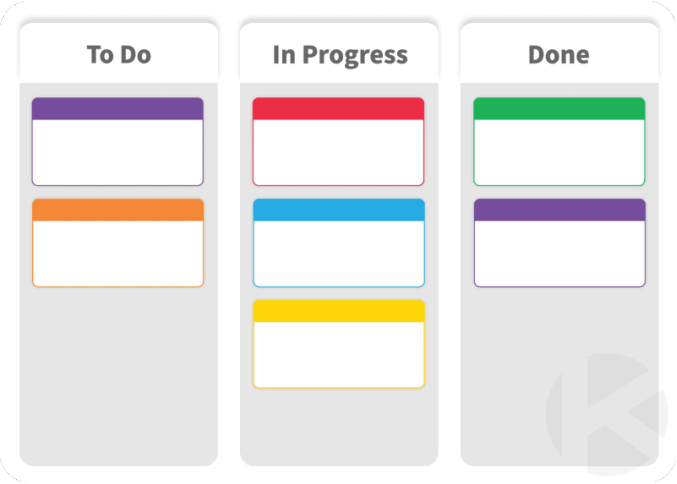
5. Una vez que tu trabajo haya sido revisado y aprobado se podrá fusionar

6. Una vez que tu trabajo haya sido fusionado debe desplegarse.

Las ventajas de esta estrategia son que gracias a la simplicidad permite la entrega y laintegración continuas.

La forma en la que he ido haciendo uso del GitHub Flow Branch Strategy ha sido creando una rama nueva por cada issue que estaba realizando y una vez que realizaba la PR y pasaba la integración continua que es un script donde se ejecutan mis tests, obtenía la luz verde para unir la funcionalidad a la rama máster.

El proyecto lo he organizado mediante la metodología ágil Kanban.



Kanban es una metodología que se implementa por medio de tableros. Es un método visual de gestión de proyectos que permite visualizar los flujos de trabajo y la carga de trabajo. En un entorno más real es mucho más aplicable ya que el flujo de trabajo es mucho mayor,aunque yo lo he utilizado sobre todo para la organización sobre tres columnas:

- To do -> Tareas que tenemos que realizar.
- In progress -> Tareas que están ya arrancadas y en proceso.
- Done -> Tareas que ya han sido finalizadas.

Un método más que añadí para saber si el código que va a ser incluido en la rama principal además de la integración continua (CI), configuré un pre-commit para antes derealizar el commit para crear la PR.

Arquitectura del software

A la hora de buscar una arquitectura para mi proyecto tenía varias para poder elegircomo MVC (Modelo Vista Controlador) o arquitectura Cliente – Servidor.
Sin embargo, he decidido quedarme con la arquitectura hexagonal ya que en los últimos años en los sistemas de software la complejidad ha aumentado por lo que el uso de "arquitecturas limpias" se han ido promoviendo más entre los desarrolladores de software.

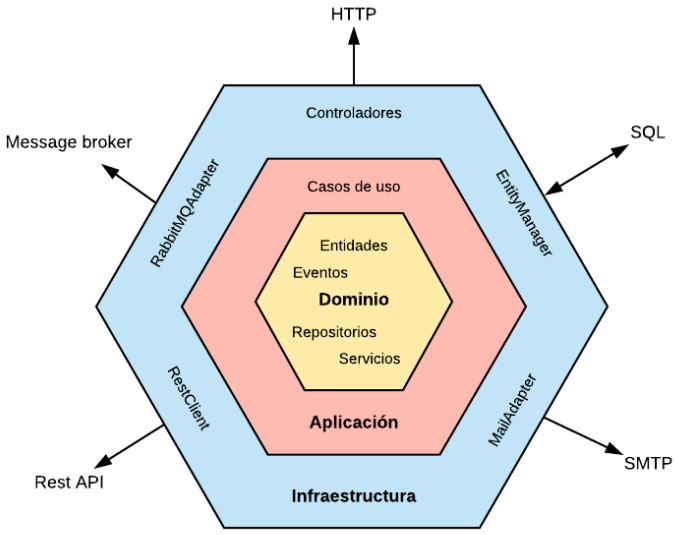
Una de las ventajas de usar la arquitectura hexagonal es la facilidad que nos aporta a la hora de separar responsabilidades mediante capas y definiendo reglas de dependencias entre ellas.
La separación de responsabilidades nos ayudara a evitar el acoplamiento de nuestro dominio con elementos externos ayudándonos a:

- Independencia del framework
- Testable
- Flexible
- Reutilizable
- Mantenible

La arquitectura hexagonal es un tipo de arquitectura para desarrollo de software que se basa en poner el foco del proyecto en nuestro dominio o modelo (nuestra parte más interna y nuestro core). [17]

La arquitectura hexagonal promueve la separación de la aplicación por contextos mediante la encapsulación de la lógica en diferentes capas de la aplicación.

Como he explicado con anterioridad lo que obtenemos haciendo uso de esta arquitecturaes un mayor nivel de aislamiento, testabilidad y control sobre nuestro código. Cada parte de nuestra arquitectura tiene unas responsabilidades y requerimientos creando así separaciones más claras entre la lógica y la funcionalidad.



Esta arquitectura se representa con forma de hexágono, pero el número de lados no es lo realmente importante, sino lo que cada uno de los lados representa.

Cada lado representa un puerto. Por ejemplo, HTTP y hacer peticiones a la aplicación. Otro puede ser un servidor de base de datos en donde persistir los datos del dominio. [18]

Nuestra arquitectura además de permitirnos desacoplar nuestra lógica de los servicios como bien he explicado antes tiene tres capas:

- Infraestructura: En ella vamos a almacenar toda la parte externa de la cual hacemos uso en la aplicación. Además, también de nuestros controladores que son los encargados de hacer las llamadas a nuestros casos de uso de la capa de aplicación, también es el encargado de manejar la respuesta de nuestros endpoints como además de nuestra request (input de datos).

- Aplicación: En esta capa es donde almacenamos nuestros casos de uso como vender moneda o abrir una nueva cartera. Desde esta capa haremos las llamadas a nuestro repositorio de la capa de Dominio. Controlaremos las respuestas de este último manejando las posibles excepciones lanzadas como que una cartera con un id especificado no existe.

- Dominio: Esta es la capa más interna, nuestro modelo y nuestro core. En ella lo que almacenamos va a ser nuestros repositorios y nuestros modelos. Sin embargo, lo que almacenaremos serán las interfaces de nuestros repositorios que más tarde mediante inyección de dependencias inyectaremos la implementación de nuestro repositorio a esta interfaz.

Desarrollo

Cada uno de los endpoint de los que conforman mi proyecto tienen un propio controlador. Cada controlador tiene una entrada de datos que usará éste en su flujo.

Como he elegido usar una arquitectura limpia y uno de los puntos importantes de esta arquitectura es la separación de responsabilidades, para tratar esos datos de entrada he creado una Form Request que se trata de una request personalizada para cada endpoint donde se define los campos que se esperan recibir en esta data. Esta request personalizada tiene distintos tipos de salidas:

- Si alguno de los campos requeridos no está definido, se lanza una excepción pidiéndole al usuario que rellene ese dato.

- Si el usuario envía un dato que debería ser de otro tipo, es decir, si se manda donde debería ir el id de una moneda su nombre, se volverá a lanzar una excepción avisándole de que ese tipo no está permitido.

- Si todos los campos son entregados de manera correcta la validación pasará permitiéndonos avanzar en la lógica de nuestro controlador.

Esta Form Request y todos los posibles escenarios de error están testados de manera unitaria. Cubriéndonos así también la entrada de data incluso antes de empezar a ejecutar nuestro código de producción.

Una vez nuestra request esté validada obtendremos un DTO o Data Transfer Object, que como su nombre indica es un objeto que transfiere datos y del cual vamos apoder extraer todos los atributos pasados en la request.

El controlador será el encargado de manejar la llamada al servicio que será nuestro caso de uso y de recoger la respuesta que este nos devolverá. Si todo ha sido ejecutado con normalidad se devolverá un json con la respuesta apropiada, en caso contrario se lanzará una excepción.

Al realizar la primera aproximación de la solución de los endpoints el controlador iba aser el encargado de manejar las excepciones lanzadas en el servicio mediante un try catch.

Sin embargo, mediante la implementación de la función render en el exceptions/Handler he conseguido automatizar el proceso. Cada vez que una excepción se lance esta función renderizará un response customizada y así no deberemos de manejar en los controladores todas las excepciones.

Uno de los fallos que tiene el uso de try/catch para el manejo de excepciones es que solo espera que se llamen a unas excepciones específicas, por lo que no sabe cómo reaccionar ante una excepción inesperada como podría ser una excepción lanzada de manera externa por parte del framework.

Todos los controladores están testados mediante tests de integración. Esta parte está testada de esta manera ya que lo que queremos comprobar en estos test no es la lógicade la aplicación de manera unitaria, si no que deseamos comprobar cómo reaccionan todas las partes de nuestra aplicación entre sí, para poder comprobar el flujo completode cada uno de los endpoints.

En nuestra capa de servicios tenemos las llamadas a los repositorios. El servicio es el encargado de manejar toda la lógica de nuestro caso de uso mediante las llamadas a los repositorios, y el encargado de lanzar las excepciones en caso de que sea necesario.

Nuestros servicios están todos testados de manera unitaria, pero usando dobles de test para poder simular un entorno donde controlemos la request que queremos recibir y los argumentos que deberíamos de tener en la llamada.

Por último, llegamos a la capa más importante y crítica para nosotros. En esta capa almacenamos nuestros modelos y los interfaces de los repositorios.

Martin Fowler definió el patrón repositorio tal que:

“El patrón repositorio consiste en separar la lógica que recupera los datos y los asignaa un modelo de entidad de la lógica de negocios que actúa sobre el modelo, esto permite que la lógica de negocios sea independiente del tipo de dato que comprende la capa de origen de datos, en pocas palabras un repositorio medio entre el dominio y lascapas de mapeo de datos, actuando como una colección de objetos de dominio en memoria.”

La razón por la que he decidido usar el patrón repositorio y como he explicado con anterioridad ya que estamos haciendo uso de una arquitectura limpia y uno de los principios que estamos intentando conseguir es desacoplarnos de las implementaciones externas como podría ser la forma en la que persistimos los datos en este caso mediante Eloquent. [19]

Eloquent es el ORM que viene incluido con Laravel, este proporciona una implementación simple para poder trabajar con la base de datos. Cada tabla de la base de datos tiene un "Modelo" correspondiente que se utiliza para interactuar con esa tabla.

Al usar el patrón repositorio si en un futuro en vez de Eloquent decidiéramos usar cualquier otro ORM como Doctrine, si tuviésemos la implementación directa de eloquent estaríamos acoplados a este y no podríamos separarnos de manera sencilla. Porlo que no estaríamos cumpliendo uno de los principios de la arquitectura limpia que estamos usando y a su vez una de la principal ventaja que nos ofrece esta.

Por lo que en nuestro dominio guardamos solo la interfaz de nuestro repositorio y le inyectamos la implementación de Eloquent desde infraestructura mediante inyección de dependencias.

En la capa de dominio además de las interfaces del repositorio tenemos los modelos usados para el proyecto.

Para este he definido 3 modelos:

1. Coin
2. Wallet
3. WalletCoin

Para cada uno de estos 3 modelos he definido su implementación con Eloquent (ej. Modelo Coin -> EloquentCoin). Estas implementaciones puro eloquent y solo me sirve para definir el objeto de base de datos y con el cual poder hacer llamadas a las bases de datos.

También he definido mi modelo puro de dominio en el que solo tengo los atributos quecomponen mi modelo y sus modelos get.

Quizá pueda no entenderse por qué he definido los dos modelos, lo explico más en profundidad…

Mi modelo eloquent u objeto de infraestructura lo usaré en la implementación de misrepositorios, ambos en capa de infraestructura.

Pero si en el repositorio lo que devuelvo es mi objeto de infraestructura estaría rompiendo la arquitectura hexagonal ya que estaría devolviendo en mi capa de dominioun objeto de infraestructura. Ya que en la arquitectura hexagonal la dirección en la que puede haber transmisión de ficheros es de fuera adentro nunca al revés.

Por lo que en mis repositorios deberé de retornar el objeto puro. Para ello hago uso de mapeadores. Estos se encargan de transformar mi objeto de infraestructura en uno dedominio puro para que así no rompamos la arquitectura hexagonal.

Y si en algún momento cambiase de ORM solo tendría que cambiar la inyección de la dependencia y crear el mapeador de mi ORM nuevo a este objeto de dominio puro.

KRAKEND
TEÓRICO

KrakenD es una Gateway extensible, declarativo y de código abierto de alto rendimiento.

Su funcionalidad principal es crear una API que actúe como un agregador de microservicios en endpoint únicos haciendo el trabajo pesado de una API Gateway automáticamente como agregar, transformar, filtrar, decodificar, acelerar, autenticar y más.

¿Por qué una API Gateway?

Cuando los consumidores de contenido de API consultan los servicios del backend, las implementaciones sufren mucha complejidad y carga con los tamaños de las respuestas de sus microservicios.

KrakenD es una API Gateway que se encuentra entre el cliente y todos los servidores. Agregando una nueva capa que elimina toda la complejidad para los clientes, brindándoles solo la información que necesita la interfaz de usuario.

KrakenD va más allá de otras herramientas que son únicamente proxis inversos y actúa como un agregador de muchas fuentes, lo que le permite consumir en una sola llamada muchos puntos finales. Le permite agrupar, validar, envolver, transformar y reducir solicitudes y respuestas. Además, admite una gran cantidad de middleware y complementos que le permiten ampliar la funcionalidad, como agregar autorización OAuth2, capas de seguridad, interrupción de circuitos, limitación de velocidad, conectividad, registro, métricas, seguimientos y mucho más.

KrakenD es un API Gateway de alto rendimiento. Un API Gateway es un componente que necesita entregarse muy rápido, ya que es una capa adicional en la infraestructura. KrakenD se creó teniendo en cuenta el rendimiento.

En un ordenador normal se calcula que KrakenD es capaz de soportar unas 18.000 requests/seg

Sin embargo, voy a probar en mi ordenador cuantás requests es capaz de soportar y en qué tiempos.

```
{
  "endpoint": "/foo",
  "backend": [
    {
      "host": ["http://127.0.0.1:8080"],
      "url_pattern": "/__debug/default"
    }
  ]
}
```

Para ello he creado un endpoint muy sencillo, en este endpoint es destacable que voy a hacer uso del endpoint __debug.

Este endpoint se puede usar para generar un backend falso usando a KrakenD como host. Es muy útil para ver la interacción entre el Gateway y los backends.

Lanzo KrakenD y voy a intentar hacer una petición a este endpoint desde consola mediante el comando *curl -i http://127.0.0.1:8080/__debug/bar* el mensaje recibido es

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Mon, 15 May 2023 11:44:08 GMT
Content-Length: 18
```

```
{"message":"pong"}
```

Si analizamos este mensaje podemos observar que recibimos un 200 y vemos que nuestro backend falso esta activo.

Si al igual que antes ahora realizamos una llamada a Krakend preguntando por el mismo endpoint lo que vamos a recibir es

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
X-Krakend: Version 2.2.0-ee
X-Krakend-Completed: true
Date: Mon, 15 May 2023 11:46:22 GMT
Content-Length: 18

{"message":"pong"}
```

Analizando el mensaje obtenido vemos que tenemos el mismo cuerpo del response, pero tenemos la cabecera X-Krakend-Completed: true lo que significa que nuestro Gateway ha obtenido satisfactoriamente la response del backend del endpoint.

A continuación, voy a utilizar *hey*. Hey es un pequeño programa que envía algo de carga a una aplicación web, en este caso a mi local. Voy a realizar dos pruebas una con 10000 requests con la cual he recibido los siguientes datos:

```
Total:        4.6758 secs
Slowest:      1.5061 secs
Fastest:      0.0012 secs
Average:      0.0916 secs
Requests/sec: 2138.6917

Total data:   180000 bytes
Size/request: 18 bytes
```

Este es el resumen obtenido de la simulación.

```
Response time histogram:
  0.001 [1]     |
  0.152 [9216]  |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.302 [383]   |■■
  0.453 [0]     |
  0.603 [0]     |
  0.754 [0]     |
  0.904 [0]     |
  1.055 [0]     |
```

El tiempo de respuesta de las peticiones, como se puede, se atiende en 0.152 segs.

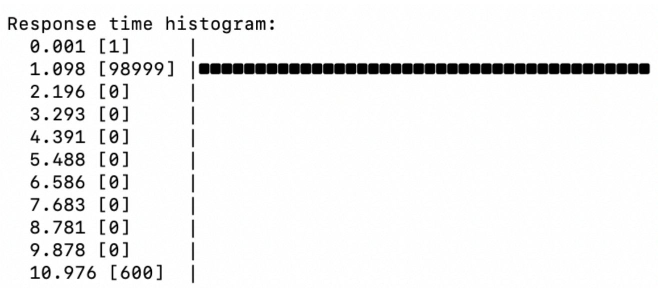
```
Latency distribution:
  10% in 0.0058 secs
  25% in 0.0157 secs
  50% in 0.0310 secs
  75% in 0.0472 secs
  90% in 0.0843 secs
  95% in 0.2082 secs
  99% in 1.4766 secs
```

La latencia obtenida es la que visualizamos en la imagen.

Para la segunda prueba usare 100000 requests. Es obvio pensar que si 10000 han tardado 4seg 100000 tardaran unos 40 segundo. Sin embargo,

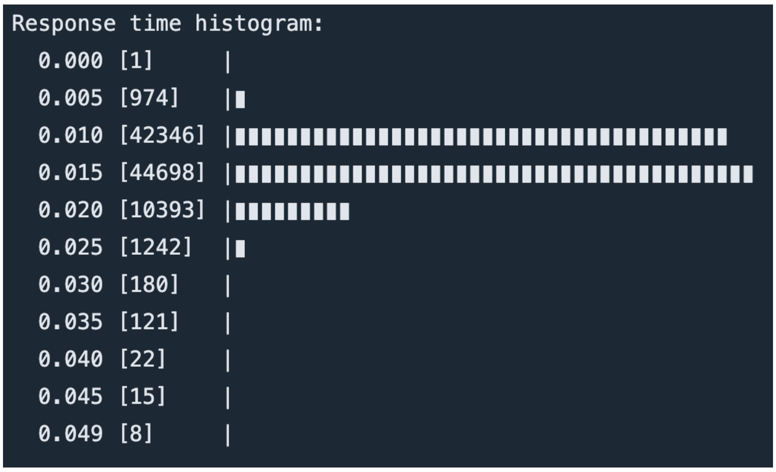
observamos que el tiempo casi se ha duplicado y que las Requests por segundo han disminuido a menos de la mitad.

```
Total:          108.6127 secs
Slowest:        10.9756 secs
Fastest:        0.0008 secs
Average:        0.1366 secs
Requests/sec:   920.7024

Total data:     1782144 bytes
Size/request:   17 bytes
```



En mi local hemos obtenido estos resultados. Si observamos los resultados de la documentación observamos

```
Total:  5.6980 secs
Slowest:  0.0494 secs
Fastest:  0.0003 secs
Average:  0.0113 secs
Requests/sec: 17549.8782
Total data: 1900000 bytes
Size/request: 19 bytes
```



Que los resultados son mucho más eficientes que los obtenidos en mi local.

A la conclusión a la que quiero llegar es a comparar la eficiencia de KrakenD frente a otras API Gateway que sean rivales, no a las velocidades obtenidas por la documentación y por mi ordenador, pero era un dato que me parecía interesante remarcarlo.

Los tres rivales a los que enfrentaré a KrakenD son:

- Vulcand
- Tyk
- Kong

<u>Vulcand</u>

Vulcand es un proxy inverso para la gestión de API HTTP y microservicios. [20]

<u>KONG</u>

Kong Gateway es una puerta de enlace API nativa de la nube liviana, rápida y flexible.

Kong Gateway se ejecuta frente a cualquier API RESTful y se puede ampliar a través de módulos y complementos. Está diseñado para ejecutarse en arquitecturas descentralizadas, incluidas las implementaciones de nubes híbridas y múltiples nubes. [21]

Tyk

Tyk es uno de los principales proveedores de administración de API en el espacio y tenemos una rica base de experiencia de proporcionar una herramienta de administración de API segura, flexible y de alto rendimiento para usuarios de todas las escalas, que buscan resolver una amplia gama de problemas digitales centrados en API desafíos de transformación. [22]

El negocio principal de Tyk es la gestión de API. Son una empresa enfocada en el producto, proporcionando una plataforma de gestión de API, que comprende Tyk Gateway, Dashboard y Portal para desarrolladores.

Para enfrentar a estas 4 empresas voy a realizar la misma práctica que para comprobar la eficiencia en local, mandar 100000 peticiones.

El resumen:

```
Total:  28.7424 secs.
Slowest:  0.2781 secs.
Fastest:  0.0009 secs.
Average:  0.0287 secs.
Requests/sec: 3479.1863
Total Data Received:  10900000 bytes.
Response Size per Request:  109 bytes.
```

KrakenD

```
Total:  50.5294 secs.
Slowest:  0.3426 secs.
Fastest:  0.0010 secs.
Average:  0.0505 secs.
Requests/sec: 1979.0451
Total Data Received:  10800000 bytes.
Response Size per Request:  108 bytes.
```

Vulcand

```
Total:  57.0194 secs.
Slowest:  1.5978 secs.
Fastest:  0.0076 secs.
Average:  0.0569 secs.
Requests/sec: 1753.7883
Total Data Received:  13600000 bytes.
Response Size per Request:  136 bytes.
```

Kong

```
Total:  57.0194 secs.
Slowest:  1.5978 secs.
Fastest:  0.0076 secs.
Average:  0.0569 secs.
Requests/sec: 1753.7883
Total Data Received:  13600000 bytes.
Response Size per Request:  136 bytes.
```

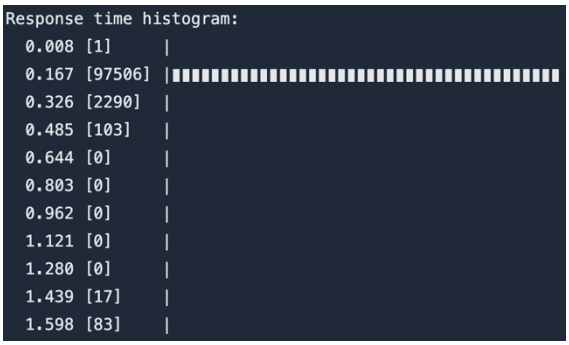
Tyk

Como se puede apreciar con gran diferencia, KrakenD es capaz de servir las 100000 request en menos de 30 segundos mientras que todas las demás tardan más de 50 segundos. Además de que es capaz de servir más de 1000 request más por segundo.
Histograma de tiempos:

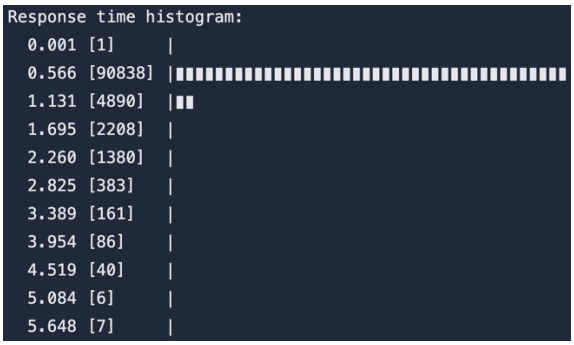
```
Response time histogram:
  0.001 [1]     |
  0.029 [71546] |∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎
  0.056 [26536] |∎∎∎∎∎∎∎∎∎∎∎∎∎∎
  0.084 [1061]  |
  0.112 [392]   |
  0.140 [248]   |
  0.167 [93]    |
  0.195 [51]    |
  0.223 [12]    |
  0.250 [27]    |
  0.278 [33]    |
```

```
Response time histogram:
  0.001 [1] |
  0.035 [21120] |∎∎∎∎∎∎∎∎∎∎∎
  0.069 [71365] |∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎
  0.103 [5946]  |∎∎∎
  0.138 [168] |
  0.172 [74]  |
  0.206 [329] |
  0.240 [496] |
  0.274 [388] |
  0.308 [88]  |
  0.343 [25]  |
```

KrakenD



Vulcand



Kong

Tyk

Como se puede observar KrakenD sirve la mayoría de sus Requests en 0.029 segundos, siguiéndole Vulcand con 0.035, posteriormente Kong con 0.167s y para finalizar Tyk con 0.566s.

Si en nuestro proyecto queremos utilizar un API Gateway una de las cosas que más importantes a evaluar es la eficiencia que tendrán nuestras peticiones. Gracias a estas métricas podremos tener una idea más clara de cómo de eficaces son estas 4 alternativas.

KRAKEND PRÁCTICO

Inicialización [23]

Para la puesta a punto o inicialización de KrakenD yo he usado Docker, obteniendo su imagen del Docker Hub. Sin embargo, hay diferentes formas de instalarlo.

El comando para iniciar la imagen de Docker es muy largo, de hecho, es este:

Docker run -p 8080:8080 -v $PWD:/etc/krakend devopsfaith/krakend run –config /etc/krakend/krakend.json

Corriendo ese comando tendríamos corriendo una primera imagen de KrakenD. Si analizamos el comando poco a poco estamos poniendo a escuchar el puerto 8080 a KrakenD y podemos ver cómo trabaja en esta url http://localhost:8080/__health.

Para poder arrancar el servidor de KrakenD todo lo que necesitamos es un único archivo de configuración. El archivo de configuración se denomina krakend.json por convención. Sin embargo se podrá renombrar y utilizar cualquier nombre, puede estar almacenado en cualquier directorio o incluso dividirse en varias partes.

Con este archivo cualquier cambio estará siempre bajo el sistema de control de versiones, y el código controla el estado del Gateway.

Archivo de configuración [24]

El archivo de configuración es el alma del KrakenD todo el comportamiento depende de este.

Hay un gran número de opciones que se pueden poner en el archivo. De momento me voy a centrar en la estructura básica y principal. Esta se compone de 4 opciones:

1. $schema: (Opcional) Cuando se añada esta opción se habilita la integración del IDE con autocompletado y documentación.

2. Version: (Obligatorio) Es la versión del archivo de configuración. Actualmente es la 3.

3. Endpoints: Es un array de endpoints ofrecidos por la Gateway, este campo es nuestro backend y configuración. La definición de tu API.

4. Extra_config: Configuración de los componentes del servicio.

Este archivo al ser el clave de nuestro API Gateway tenemos un comando que valida que la configuración contiene los formatos soportados, krakend check.

Flexible Configuration [25]

Hay distinto tipos de configuraciones del archivo principal, yo me voy a centrar en la configuración que he implementado. Esta es la configuración flexible.

La configuración flexible nos permite declarar la configuración utilizando múltiples archivos y utilizar un sistema de plantillas, lo cual nos abre la puerta a configuraciones multi-entorno y a la reutilización de código.

Las plantillas soportan varios tipos de formatos como json, yaml o tmpl.

Esta configuración es recomendable usarla cuando necesite:

- Dividir una configuración extensa en varios archivos
- Inyectar variables como variables de entorno en la configuración
- Bloques de código reutilizables para evitar la duplicidad del código
- Una mejor organización del código

Cabe destacar que este tipo de configuración tiene un comando distinto para arrancar KrakenD. La activación de la configuración flexible tiene varias variables de configuración que pueden ser definidas dependiendo de lo que al desarrollador le interese:

- FC_ENABLE

Activa la configuración flexible. Se puede utilizar 1 o cualquier otro valor, sin embargo 0 no desactivará esta configuración.

- FC_TEMPLATES

Es la ruta al directorio donde almacenaremos las plantillas.

- FC_SETTINGS

Es la ruta al directorio de ajustes. En este directorio tendremos los archivos JSON que se pueden utilizar para rellenar los valores en las plantillas (variables de configuración). Similar a los archivos env.

- FC_PARTIALS

Es la ruta al directorio de los parciales. Estos archivos son textos que no se evalúan y que se insertan en la posición tal cual. (ej. Circuit Breaker)

- FC_OUT

Guarda la configuración resultante tras renderizar la plantilla. Es necesario no se usa contenido json o cuando necesitas pasar la salida a otro programa para hacer un ‘check –lint’.

Una vez explicadas todas las variables de configuración paso a aplicarlo a mi caso.

```
docker run -p 8080:8080 --rm -it \
    -v "$(PWD):/etc/krakend/" \
    -e FC_ENABLE=1 \
    -e FC_SETTINGS=/etc/krakend/config/settings/local \
    -e FC_PARTIALS=/etc/krakend/config/partials \
    -e FC_TEMPLATES=/etc/krakend/config/templates \
    -e FC_OUT=compiled.json \
    krakend/krakend-ee:2.1.0 \
    run -c krakend.tmpl
```

Como podemos observar estoy arrancando KrakenD, pongo a escuchar al servidor en el puerto 8080. Activo la configuración flexible y defino los directorios para las variables.
A la plantilla resultante la llamo 'compiled.json'. El comando que ejecuto es run y con el flag -c le paso la plantilla del KrakenD, en esta plantilla estoy usando la configuración flexible y las principales opciones del KrakenD.

```
"$id": "https://www.krakend.io/schema/v3.json",
"version": 3,
"name": "tfg-crypto-krakend",
"port": {{ .env.port }},
"timeout": "{{ .env.timeout }}",
"extra_config": {{ template "extra_config.tmpl" . }},
"endpoints": [
  {{ template "coin_endpoints.tmpl" . }},
  {{ template "status_endpoints.tmpl" . }},
  {{ template "wallet_endpoints.tmpl" . }},
  {{ template "oauth_endpoints.tmpl" . }}
]
```

Herramientas

Cuando lanzamos nuestro servidor de KrakenD los mensajes de registro de la aplicación son los errores, advertencias, información de depuración y otros mensajes.

Estos registros son personalizables, ya que se puede ampliar su funcionalidad, elegir el nivel de detalle de los mensajes de salida de la aplicación. Existen muchos exportadores para enviar los registros a terceros. Con ello pasaremos de visualizar una consola

```
➜  krakend_tfg git:(main) ✗ make watch
docker run -p 8080:8080 --rm -it \
               -v "/Users/asier.alba540deg.com/Desktop/TFG/krakend_tfg:/etc/krakend/" \
               -e FC_ENABLE=1 \
               -e FC_SETTINGS=/etc/krakend/config/settings/local \
               -e FC_PARTIALS=/etc/krakend/config/partials \
               -e FC_TEMPLATES=/etc/krakend/config/templates \
               -e FC_OUT=compiled.json \
               krakend/krakend-ee:watch \
               run -c krakend.tmpl
Watching changes on files /etc/krakend/
Ignoring saves to file compiled.json
[00] Starting service
[00] Parsing configuration file: krakend.tmpl
[00] ********************************************************
[00] LICENSE ERROR
[00] ********************************************************
[00] There is a problem with your license:
[00] open ./LICENSE: no such file or directory
[00]
[00] KRAKEND IS RUNNING WITH LIMITED FUNCTIONALITY
[00]
[00] The server will start with open-source features only
[00] ********************************************************
[00] 2023/05/08 15:37:44 KRAKEND INFO: Starting KrakenD v2.2.0-ee
[00] 2023/05/08 15:37:44 KRAKEND INFO: Starting the KrakenD instance
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/status/:coin_id] Building the proxy pipe
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [BACKEND: /coin/status/{{.Coin_id}}] Building the backend pipe
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/status/:coin_id] Building the http handler
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/status/:coin_id][JWTSigner] Signer disabled
[00] 2023/05/08 15:37:44 KRAKEND INFO: [ENDPOINT: /Coin/coin/status/:coin_id][JWTValidator] Validator disabled for this endpoint
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/buy] Building the proxy pipe
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [BACKEND: /coin/buy] Building the backend pipe
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/buy][JSONSchema] Validator enabled
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/buy] Building the http handler
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/buy][JWTSigner] Signer disabled
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/buy][JWTValidator] Roles will be matched against the key: 'roles'
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/buy][JWTValidator] No scope validation required
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/buy][JWTValidator] Validator enabled for this endpoint. Operation debug is enabled
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/sell] Building the proxy pipe
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [BACKEND: /coin/sell] Building the backend pipe
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/sell][JSONSchema] Validator enabled
[00] 2023/05/08 15:37:44 KRAKEND DEBUG: [ENDPOINT: /Coin/coin/sell] Building the http handler
```

A una consola donde visualizar los registros importantes será mucho más fácil gracias a los colores por nivel.

```
➜  krakend_tfg git:(main) ✗ make watch
docker run -p 8080:8080 --rm -it \
               -v "/Users/asier.alba540deg.com/Desktop/TFG/krakend_tfg:/etc/krakend/" \
               -e FC_ENABLE=1 \
               -e FC_SETTINGS=/etc/krakend/config/settings/local \
               -e FC_PARTIALS=/etc/krakend/config/partials \
               -e FC_TEMPLATES=/etc/krakend/config/templates \
               -e FC_OUT=compiled.json \
               krakend/krakend-ee:watch \
               run -c krakend.tmpl
Watching changes on files /etc/krakend/
Ignoring saves to file compiled.json
[00] Starting service
[00] Parsing configuration file: krakend.tmpl
[00] ********************************************************
[00] LICENSE ERROR
[00] ********************************************************
[00] There is a problem with your license:
[00] open ./LICENSE: no such file or directory
[00]
[00] KRAKEND IS RUNNING WITH LIMITED FUNCTIONALITY
[00]
[00] The server will start with open-source features only
[00] ********************************************************
[00] [KRAKEND] 2023/05/08 - 15:39:13.049 ▶ DEBUG [SERVICE: telemetry/logging] Improved logging started.
[00] [KRAKEND] 2023/05/08 - 15:39:13.049 ▶ INFO Starting KrakenD v2.2.0-ee
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ INFO Starting the KrakenD instance
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ DEBUG [ENDPOINT: /Coin/coin/status/:coin_id] Building the proxy pipe
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ DEBUG [BACKEND: /coin/status/{{.Coin_id}}] Building the backend pipe
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ DEBUG [ENDPOINT: /Coin/coin/status/:coin_id] Building the http handler
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ DEBUG [ENDPOINT: /Coin/coin/status/:coin_id][JWTSigner] Signer disabled
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ INFO [ENDPOINT: /Coin/coin/status/:coin_id][JWTValidator] Validator disabled for this endpoint
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ DEBUG [ENDPOINT: /Coin/coin/buy] Building the proxy pipe
[00] [KRAKEND] 2023/05/08 - 15:39:13.050 ▶ DEBUG [BACKEND: /coin/buy] Building the backend pipe
[00] [KRAKEND] 2023/05/08 - 15:39:13.051 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JSONSchema] Validator enabled
[00] [KRAKEND] 2023/05/08 - 15:39:13.051 ▶ DEBUG [ENDPOINT: /Coin/coin/buy] Building the http handler
[00] [KRAKEND] 2023/05/08 - 15:39:13.051 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTSigner] Signer disabled
[00] [KRAKEND] 2023/05/08 - 15:39:13.075 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTValidator] Roles will be matched against the key: 'roles'
[00] [KRAKEND] 2023/05/08 - 15:39:13.075 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTValidator] No scope validation required
[00] [KRAKEND] 2023/05/08 - 15:39:13.075 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTValidator] Validator enabled for this endpoint. Operation debug is enabled
[00] [KRAKEND] 2023/05/08 - 15:39:13.075 ▶ DEBUG [ENDPOINT: /Coin/coin/sell] Building the proxy pipe
[00] [KRAKEND] 2023/05/08 - 15:39:13.075 ▶ DEBUG [BACKEND: /coin/sell] Building the backend pipe
[00] [KRAKEND] 2023/05/08 - 15:39:13.075 ▶ DEBUG [ENDPOINT: /Coin/coin/sell][JSONSchema] Validator enabled
```

Como podemos observar en las capturas anteriores al utilizar la configuración flexible los comandos tienen una longitud bastante considerable, por lo que he decidido crear un fichero Makefile. En él se almacenarán todos los comandos incluso poder hacer una concatenación de comandos para por ejemplo obtener toda la documentación con un solo comando.

En el Makefile he añadido varios comandos, no solo únicamente comandos de KrakenD sino también comandos para correr los tests de Karate. Estos comandos son:

- Run
- Test -> Ejecutar todos los test de karate
- coinTest -> Ejecutar los test de la clase coin
- loginTest -> Ejecutar los test de obtención de token
- walletTest -> Ejecutar los test de la clase Wallet
- Lint -> Comprobar si el archivo de configuración es correcto

Si estamos usando para lanzar nuestro KrakenD el comando make run cada vez que realicemos un cambio en la configuración se quedará obsoleto ya que no recoge los nuevos datos. Para ello tenemos una herramienta que reinicia KrakenD de manera automática.

Se trata de la herramienta watch y lo único que dista del comando run es la imagen de krakend. En vez de `krakend/krakend-ee:2.1.0` pasaremos a `krakend/krakend-ee:watch`. [26]

Es decir, cuando realicemos un cambio en el archivo de configuración en vez de tener que parar el servidor de KrakenD a mano el propio watch reiniciara este servidor con los cambios aplicados.

Sin embargo, qué pasa cuando comentemos un fallo en la confirmación como una coma sobrante o una faltante, unas comillas faltantes o una llave mal cerrada.

Si tenemos una consola abierta con el servidor de KrakenD ejecutándose veremos que no se ha ejecutado como debería y podremos ver el error velozmente. Por ejemplo, en la imagen siguiente podemos observar una pequeña traza de como el servidor se está ejecutado correctamente, se muestran algunos de nuestros endpoints.

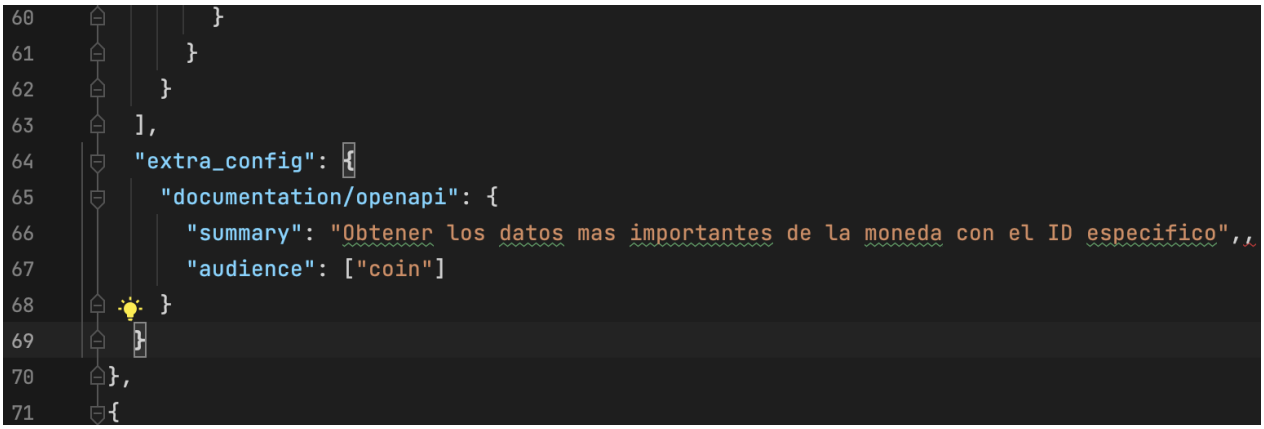
```
docker run -p 8080:8080 --rm -it \
               -v "/Users/asier.alba540deg.com/Desktop/TFG/krakend_tfg:/etc/krakend/" \
               -e FC_ENABLE=1 \
               -e FC_SETTINGS=/etc/krakend/config/settings/local \
               -e FC_PARTIALS=/etc/krakend/config/partials \
               -e FC_TEMPLATES=/etc/krakend/config/templates \
               -e FC_OUT=compiled.json \
               krakend/krakend-ee:watch \
               run -c krakend.tmpl
Watching changes on files /etc/krakend/
Ignoring saves to file compiled.json
[00] Starting service
[00] Parsing configuration file: krakend.tmpl
[00] ****************************************************
[00] LICENSE ERROR
[00] ****************************************************
[00] There is a problem with your license:
[00] open ./LICENSE: no such file or directory
[00]
[00] KRAKEND IS RUNNING WITH LIMITED FUNCTIONALITY
[00]
[00] The server will start with open-source features only
[00] ****************************************************
[00] [KRAKEND] 2023/05/08 - 13:19:46.950 ▶ DEBUG [SERVICE: telemetry/logging] Improved logging started.
[00] [KRAKEND] 2023/05/08 - 13:19:46.950 ▶ INFO Starting KrakenD v2.2.0-ee
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ INFO Starting the KrakenD instance
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [ENDPOINT: /Coin/coin/status/:coin_id] Building the proxy pipe
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [BACKEND: /coin/status/{{.Coin_id}}] Building the backend pipe
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [ENDPOINT: /Coin/coin/status/:coin_id] Building the http handler
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [ENDPOINT: /Coin/coin/status/:coin_id][JWTSigner] Signer disabled
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ INFO [ENDPOINT: /Coin/coin/status/:coin_id][JWTValidator] Validator disabled for this endpoint
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [ENDPOINT: /Coin/coin/buy] Building the proxy pipe
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [BACKEND: /coin/buy] Building the backend pipe
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JSONSchema] Validator enabled
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [ENDPOINT: /Coin/coin/buy] Building the http handler
[00] [KRAKEND] 2023/05/08 - 13:19:46.951 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTSigner] Signer disabled
[00] [KRAKEND] 2023/05/08 - 13:19:46.988 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTValidator] Roles will be matched against the key: 'roles'
[00] [KRAKEND] 2023/05/08 - 13:19:46.988 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTValidator] No scope validation required
[00] [KRAKEND] 2023/05/08 - 13:19:46.988 ▶ DEBUG [ENDPOINT: /Coin/coin/buy][JWTValidator] Validator enabled for this endpoint. Operation debug is enabled
[00] [KRAKEND] 2023/05/08 - 13:19:46.988 ▶ DEBUG [ENDPOINT: /Coin/coin/sell] Building the proxy pipe
[00] [KRAKEND] 2023/05/08 - 13:19:46.988 ▶ DEBUG [BACKEND: /coin/sell] Building the backend pipe
[00] [KRAKEND] 2023/05/08 - 13:19:46.988 ▶ DEBUG [ENDPOINT: /Coin/coin/sell][JSONSchema] Validator enabled
```

Añado intencionalmente un fallo al archivo de configuración, es decir, una coma sobrante y obtenemos este error:

```
[00] [KRAKEND] 2023/05/08 - 13:19:46.999 ▶ INFO [SERVICE: Gin] Listening on port: 8080
[00] [KRAKEND] 2023/05/08 - 13:19:46.999 ▶ DEBUG [PLUGIN: Server] No plugins registered for the module
[00] Killing service
[00] ^C2023/05/08 13:22:14 Signal intercepted: interrupt
[00] [KRAKEND] 2023/05/08 - 13:22:14.918 ▶ INFO [SERVICE: Gin] Router execution ended
[00] Starting service
[00] ERROR parsing the configuration file: 'compiled.json': invalid character ',' looking for beginning of object key string, offset: 1422, row: 65, col: 88
```

Lo que nos muestra este error es que en el compiled.json tenemos un fallo, `invalid carácter ‘,’` en la línea 65. Si nos fijamos al lanzar el comando del watch definimos que la salida de ese comando se refleje en un fichero que es el compiled.json.

Si nos dirigimos a este archivo a la línea 65 encontraremos el error:



Otra herramienta para ver si tenemos fallos en nuestros archivos de configuración es nuestro comando check, este comando valida los archivos de configuración de KrakenD.

Con esta herramienta podemos realizar tres tareas:

1. Validación de sintaxis.
2. Linting. Además de comprobar si el archivo está bien configurado este comando comprueba considerando reglas estándar de KrakenD con el objeto de detectar tipos erróneos, atributos desconocidos o componentes mal colocados.
3. Testing. Prueba una ejecución del servicio para detectar problemas no solo de linting sino de tiempo de ejecución.

Por lo que con el comando check puede garantizar que la configuración es válida con las tres validaciones. [27]

Los flags que este comando acepta son:

- -c o –config -> Para especificar la ruta del archivo de configuración en cualquiera de los formatos soportados o la plantilla de inicio para la configuración flexible
- -t o –test-gin-routes -> Para la probar la configuración intentando iniciar el servicio durante un segundo, este flag puede ayudar mucho para evitar rutas conflictivas y otros problemas no relacionados con el linting.
- -l o –lint -> Comprobar que la configuración esta lintado correctamente.
- -d o –debug -> Para habilitar la depuración. Tenemos 3 niveles de información:
    - -d -> Nos devuelve la información más importante.
    - -dd -> Añade más información en la configuración global y muestra la configuración del extra_config.
    - -ddd -> Nos devuelve todo lo que KrakenD puede parsear de la configuración

ENDPOINT CONFIGURATION

Introducción

KrakenD tiene una forma particular de definir los endpoints. Un endpoint es una URL de una API o en otras palabras un backend que se encarga de contestar a una petición realizada por un usuario.

El endpoint recibe peticiones con el objetivo de que nuestra aplicación retorne una respuesta de la información que está definida en dicho punto.

Como mi proyecto usa una API REST la definición de endpoint varía en que nuestro endpoint es una interfaz que sirve para realizar la conexión de varios sistemas. Se trata de un HTTP que sirve para obtener información y enviar datos o información, si procede, en formato XML o JSON.

Los endpoints de KrakenD es la parte de la configuración más crítica ya que es la parte que consumen los usuarios finales. Al añadir estos endpoints se crea el contrato de API que consumirán los usuarios.

Los endpoints se deben colocar en la raíz del archivo de configuración.

```
"endpoints": [
  {{ template "coin_endpoints.tmpl" . }},
  {{ template "status_endpoints.tmpl" . }},
  {{ template "wallet_endpoints.tmpl" . }},
  {{ template "oauth_endpoints.tmpl" . }}
]
```

Definición básica [28]

Para crear un contrato para un endpoint, se debe de añadir este en la colección de endpoints. Esta definición debe contener el nombre y la sección backend (a donde se conectará nuestro endpoint). Si no se declara más información se toman valores por defecto. Sin embargo, para mi proyecto yo he utilizado algunos atributos más como:

Endpoint

Se trata de la URL exacta del recurso que queremos exponer de nuestra API. Ej: /Coin/coin/status/{coin_id} o /Coin/coin/buy.

Method

En este campo definiremos el método soportado por este endpoint. Ej: GET, POST, PUT, PATCH, DELETE

Output_encoding

Este atributo lo definiré en profundidad más adelante, pero de momento especificar que el Gateway puede trabajar con distintos tipos de contenido, permitiendo como consumir el contenido.

Timeout

Este atributo define toda la duración de la tubería o el tiempo que se mantiene activa esa tubería hasta que los backends respondan y se procesen todos los componentes que intervienen en el endpoint (la petición, la obtención de datos, la manipulación, etc.).

Este campo se especifica en segundos(s) o en milisegundos(ms). Por defecto 2s.

Extra_config

Entradas adicionales de configuración para los componentes que se ejecutan dentro de este endpoint. En mi caso para realizar la documentación de cada uno de los endpoints y algunas herramientas más. Como el template del coin_validator.tmpl que más adelante explicare para que lo uso o nuestro *validation/json-schema.*

Para aclarar más como es la definición del endpoint, adjunto una imagen donde vamos la definición del endpoint para la ruta /Coin/coin/buy.

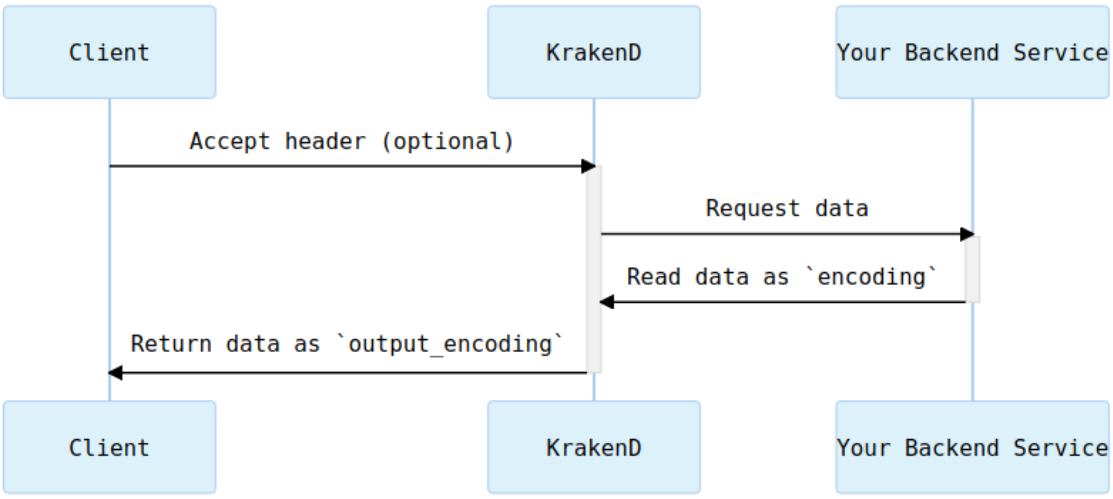
```
"endpoint": "/Coin/coin/buy",
"method": "POST",
"output_encoding": "json",
"timeout": "{{ .env.long_timeout }}",
"extra_config": {
  {{ template "coin_validator.tmpl" . }},
  "validation/json-schema": {
    "type": "object",
    "properties": {
      "wallet_id": {
        "type": "integer"
      },
      "coin_id": {
        "type": "integer"
      },
      "amount": {
        "type": "number"
      }
    },
    "required": [ "wallet_id", "coin_id", "amount" ]
  },
  "documentation/openapi": {
    "summary": "Comprar una moneda definiendo su ID la cantidad a comprar y la ID de la wallet donde guardar",
    "audience": ["coin"]
  }
},
```

Output_encoding [29]

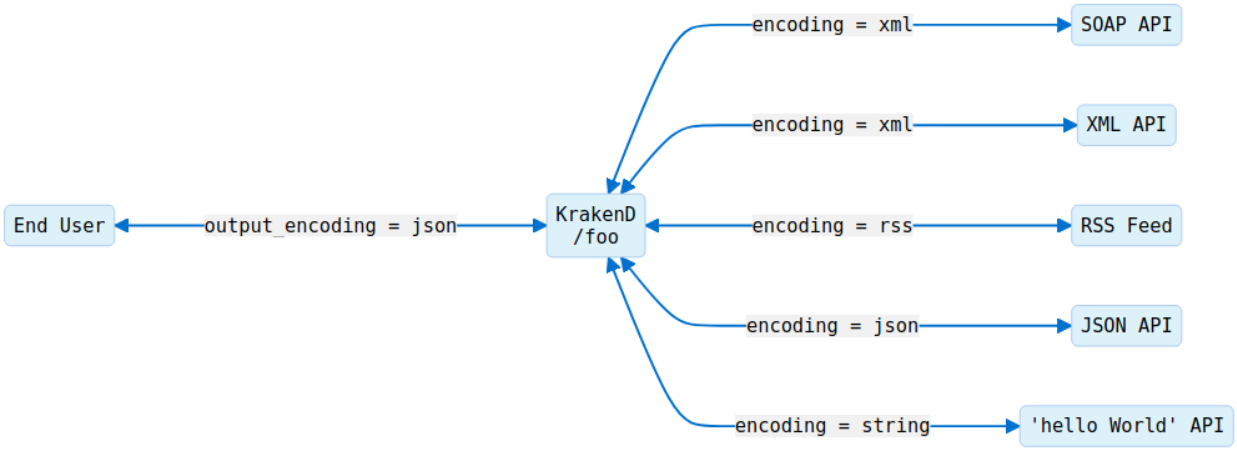
KrakenD no funciona como un proxy inverso a no ser que este atributo tenga la codificación `no-op`. Un proxy inverso es un tipo de servidor proxy que recupera recursos en nombre de un cliente externo desde uno o más servidores externos. Estos recursos se devuelven al cliente como si se originaran en el propio servidor web.

Cuando los clientes consumen los endpoints a través de KrakenD, se transforman automáticamente en la codificación que se desee para tener la oportunidad de manipular y agregar los datos fácilmente.

KrakenD puede enviar la respuesta de vuelta al cliente en distintos formatos a los que proporcionan los propios servicios. Definimos output_encoding a la codificación que se le proporciona al usuario final. El contenido que el backend proporciona los datos a KrakenD se definen como encoding.



En la siguiente imagen podemos observar un esquema muy sencillo donde se ve el flujo completo de una petición a nuestro backend.



Nuestro usuario final hace una petición a nuestro endpoint /foo definido en nuestro KrakenD, este usuario realiza la petición en formato JSON. Sin embargo, este endpoint tiene diferentes backends cada uno le devuelve un formato distinto como por ejemplo en formato XML. Por lo

que nuestro KrakenD que recibe la respuesta del backend en formato Xml transformará esa respuesta a formato JSON y le enviará al usuario final en el formato inicial.

Los valores que puede tener el atributo output-encoding son:

No-op

No operation, significa que KrakenD salta cualquier tipo de encoding o decoding y devuelve el contenido tal y como lo devuelve el backend.

Json

Es el tipo establecido por defecto. Significa que el endpoint siempre devolverá la response en el formato JSON.

Negotiate

Admite que el cliente decida el parseo de la response recibida mediante el header Accept. Puede elegir entre application/json, application/xml y text/plain

Validation/json-schema [30]

La mayoría de endpoints de KrakenD van a recibir una request o un objeto JSON, en su cuerpo pueden aplicar validaciones automáticas utilizando vocabulario JSON Schema antes de que el contenido pase a los backends.

Esta herramienta permite definir reglas sobre el cuerpo, la definición del tipo o incluso validar los valores de los campos.

Si la validación falla, KrakenD nos devuelve un 400 Bad Request, si tiene éxito se devuelve la response del backend.

Valiéndonos del endpoint /Coin/coin/buy vemos que los 3 atributos son requeridos en la request. En caso de no añadir ninguno de éstos recibiremos:

- (root): wallet_id is required
- (root): coin_id is required
- (root): amount is required

Si añadimos todos los campos con un tipo de variable distinto al definido en el json-schema recibiremos:

- amount: Invalid type. Expected: number, given string
- coin_id: Invalid type. Expected: integer, given number
- wallet_id: Invalid type. Expected: integer, given number

Los tres parámetros indispensables que tenemos que definir en el json-schema son:

1. Type
2. Required
3. Properties

Dentro de este último parámetro podríamos crear un array o incluso otro objeto, imaginemos que tenemos los 3 atributos del endpoint englobados en un objeto llamado parámetro. Nuestros json-schema debería ser:

```
"validation/json-schema": {
  "type": "object",
  "properties": {
    "parametros": {
      "type": "object",
      "properties": {
        "wallet_id": {
          "type": "integer"
        },
        "coin_id": {
          "type": "integer"
        },
        "amount": {
          "type": "number"
        }
      },
      "required": [
        "wallet_id",
        "coin_id",
        "amount"
      ]
    }
  },
  "required": [
    "parametros"
  ]
}
```

Además, al haber englobado a las 3 propiedades en una la response si mandamos una request vacía variará ya que recibiremos:

- (root): parametros is required

En el momento en que definamos los tres atributos dentro de parámetros recibiremos la response del backend que deseamos.

Otro atributo que puede ser útil es el *pattern,* para definir fechas ya que podremos restringir la entrada de la fecha a un patrón. Como ejemplo solo quiero dejar pasar una fecha con expresión regular '\d{4}-[01]\d-[0-3]\d' que esta expresión cumplirá 2023-05-29, todas las fechas que no cumplan este formato serán rechazadas.

Concurrent request [31]

Gracias a KrakenD podemos realizar peticiones concurrentes a nuestros servidores. Las peticiones concurrentes son una buena técnica para mejorar los tiempos de respuesta y disminuir las tasas de error solicitando en paralelo la misma información varias veces.

Si en vez de realizar la misma petición a un backend se realiza a varios backends a la vez cuando el primer backend nos devuelva la información se cancelarán las peticiones restantes.

Dependerá mucho de la configuración, pero está comprobado que mejora los tiempos de respuesta hasta en un 75% o más en comparación con el uso directo sin pasarela.

Cuando se utilizan peticiones concurrentes, los servicios backend deben ser capaces de manejar una carga adicional, ya que esta técnica añade más presión a los backends. Para ello podemos hacer uso de las concurrent_calls.

Las concurrent_calls son las llamadas concurrentes que se realizan, pero debes de tener en cuenta que si solo cuentas con un único servidor para manejar la carga. KrakenD abrirá las tres conexiones contra este servidor. Sin embargo, tuviésemos dos servidores y 3 llamas el servidor que recibirá las llamadas dependerá de la decisión que tome el balanceador de carga interno.

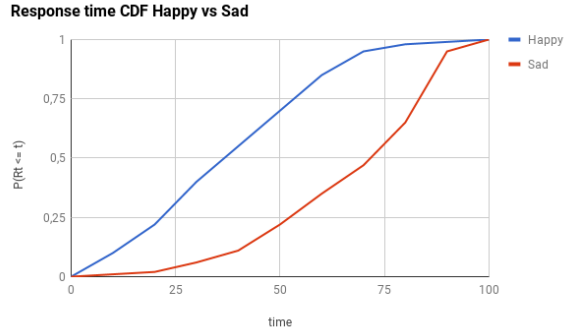
¿Cómo funciona concurrent_callls?

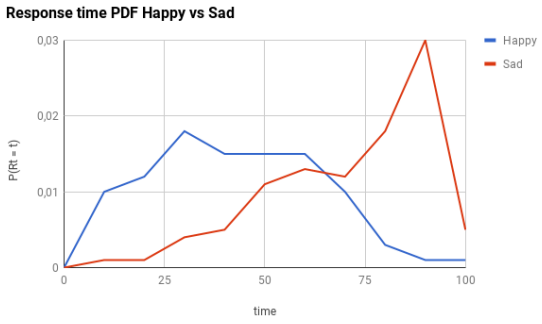
KrakenD envía las N llamadas concurrentes a los backends para la misma petición a un endpoint. Cuando se recibe la primera respuesta satisfactoria, KrakenD cancela las solicitudes restantes e ignora cualquier fallo anterior. Solo en el caso en el que todas las llamadas fallen el endpoint también recibirá el fallo,

La contra más aparente del uso de las llamadas concurrentes es el incremento de la carga en los servicios del backend. Sin embargo, a tus usuarios les gustara más, menos errores y respuestas más rápidas.

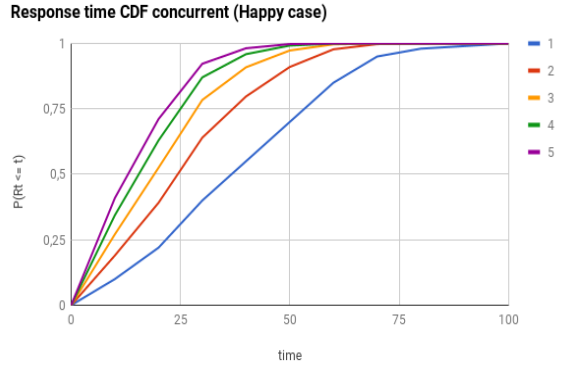
Impacto de las concurrent_calls

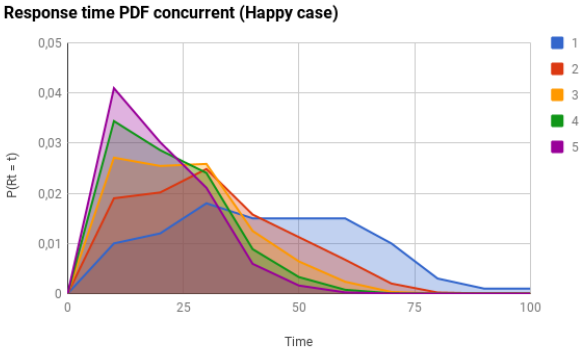
Para demostrar el uso de estas llamadas concurrentes vamos a imaginar dos escenarios diferentes: el happy y el sad. Voy a mostrar dos graficas un CDF o función de distribución acumulativa y una PDF o función de densidad de probabilidad. En caso de que nuestros valores de tiempo de respuesta sean en 100 ms las gráficas quedarían así…





Los siguientes gráficos muestran el efecto del uso de las distintas llamadas concurrentes





Sequential Proxy [32]

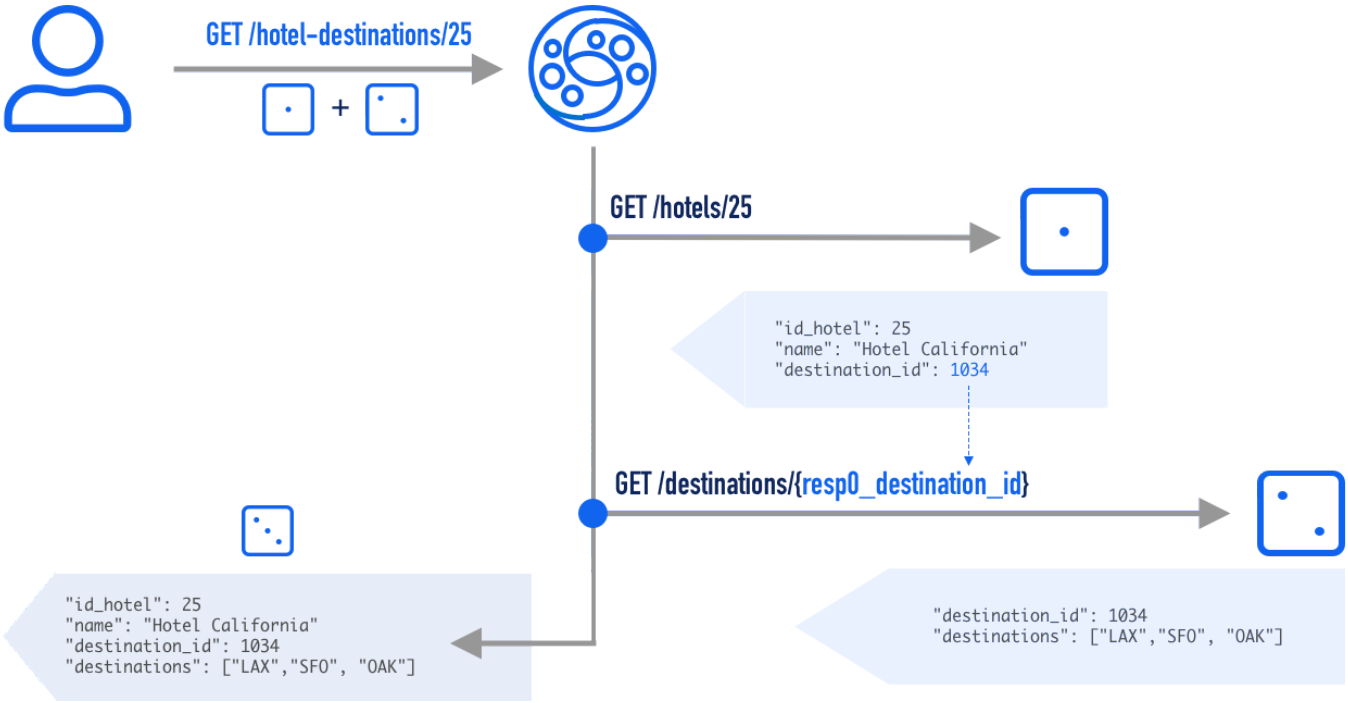
La mejor experiencia que los consumidores pueden tener con KrakenD es dejando que el sistema obtenga todos los datos de los diferentes backends concurrentes al mismo tiempo. Sin embargo, hay ocasiones que es necesario retrasar una llamada al backend hasta que se pueda inyectar como entrada el resultado de una llamada anterior.

Para esto KrakenD nos ofrece una herramienta que es el proxy secuencial, este te permite encadenar peticiones de backends. Para activar esta herramienta solo necesitamos definir la siguiente configuración:

```
"extra_config": {
        "proxy": {
                "sequential": true
        }
}
```

Cuando el proxy secuencial esta activado, el url_pattern de cada backend puede utilizar una nueva variable que hace referencia a la respuesta de una llamada anterior a la API. La variable es semejante a : *{resp0_XXXX}*

Donde 0 es el índice del backend especifico al que queremos acceder, siendo 0 el primer backend y XXXX es nombre del atributo que se quiere inyectar desde la respuesta de la llamada anterior.



El usuario llama a la pasarela con una URL /hotel-destinations/{id}, que necesita obtener la información del hotel y todos sus destinos asociados. Si suponemos que el ID solicitado es el numero 25. La pasarela llamará al backend /hotels/25 que devuelve los datos del hotel solicitado, incluido un campo destination_id que es el identificador de relación. La salida para GET /hotels/25 es :

{
	“hotel_id”: 25,
	“name”: “Hotel Destination”,
	“destination_id”: 1034
}

KrakenD espera la respuesta del backend e inyecta el valor de destination_id en la URL de la siguiente llamada al backend. En este caso, la siguiente llamada es GET /destinations/1034, y la respuesta es:

{
	“destination_id”: 1034,
	“destination”: [
		“LAX”,
		“SFO”,
		“OAK”
	]
}

Ahora KrakenD tiene ambas respuestas de los backends y puede fusionar los datos, devolviendo el siguiente objeto agregado al usuario:

{
  "hotel_id": 25,
  "name": "Hotel California",

```
  "destination_id": 1034,
  "destinations": [
    "LAX",
    "SFO",
    "OAK"
  ]
}
```

Una vez que ya hemos visto el ejemplo de la documentación voy a explicar un caso de uso con sequential proxy aplicado a mi proyecto y con un punto más de complejidad.

He creado un endpoint que lo que va a realizar es dos llamadas a diferentes endpoints. En primer lugar, llamaremos al endpoint cuya ruta es /oauth/token, este endpoint lo explicaré en profundidad más adelante en la sección de authentication.

Para realizar este endpoint debemos de llamar con un request especifica. Para definir esta request en la propia definición del endpoint hago uso de *modifier/martian* que es un componente que nos ayuda a transformar request y responses. Como necesito pasarle un cuerpo lo defino así:

```
"modifier/martian": {
  "header.Modifier": {
    "scope": ["request"],
    "body": "{{- `{"grant_type": "password","client_id": "4","client_secret":
"XbGSNyZiRTpGWQYiB5hcseoMYnga0zMFejT8J8T5","username":
"alba.126427@e.unavarra.es","password": "asiertfg"}` | b64enc -}}"
  }
}
```

Con esta request y la llamada al endpoint conseguiré el access_token que necesito para poder pasarle al segundo endpoint para autenticar mi petición a la url /coin/status/90.

Por lo que como necesitaré pasarle un header de Authentication volveré a hacer uso del modifier/martian en este caso así:

```
"modifier/martian": {
  "header.Modifier": {
    "scope": ["request"],
    "name": "Authorization",
    "value": "Bearer {resp0_access_token}"
  }
}
```

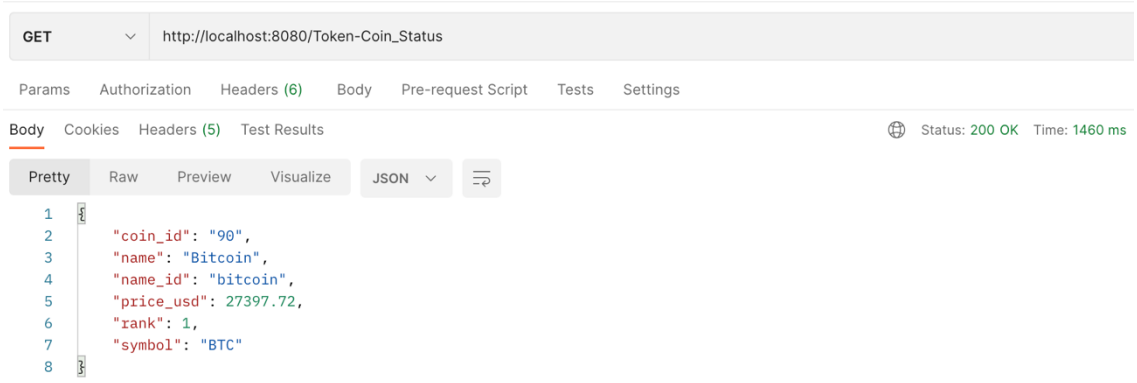
Gracias al sequential proxy puedo automatizar las llamadas a los endpoint en vez de estar pasándole el Access_token al endpoint en un header de authentication.

El código del ejemplo es:

```
{
  "endpoint": "/Token-Coin_Status",
  "method": "GET",
  "output_encoding": "json",
  "timeout": "{{ .env.long_timeout }}",
  "backend": [
    {
      "url_pattern": "/oauth/token",
      "encoding": "json",
```

```
      "method": "POST",
      "host": [
        "{{ .env.crypto_url }}"
      ],
      "extra_config": {
        "modifier/martian": {
          "header.Modifier": {
            "scope": ["request"],
            "body": "{{- `{"grant_type": "password","client_id":
"4","client_secret": "XbGSNyZiRTpGWQYiB5hcseoMYnga0zMFejT8J8T5","username":
"alba.126427@e.unavarra.es","password": "asiertfg"}` | b64enc -}}"
          }
        }
      }
    },
    {
      "url_pattern": "/coin/status/90",
      "encoding": "json",
      "method": "GET",
      "host": [
        "{{ .env.crypto_url }}"
      ],
      "extra_config": {
        "modifier/martian": {
          "header.Modifier": {
            "scope": ["request"],
            "name": "Authorization",
            "value": "Bearer {resp0_access_token}"
          }
        },
        "backend/http": {
          "return_error_details": "backend"
        }
      }
    }
  ]
}
```

Y la response de la llamada a este endpoint es



BACKEND CONFIGURATION

The backend Object [33]

El concepto de backend se refiere a los servidores de origen que proporcionan los datos necesarios para rellenar tus endpoints. Un backend puede ser algo como tu API basada en HTTP.

Un backend puede ser cualquier servidor dentro o fuera de la red si es accesible por KrakenD. Por ejemplo, puedes crear endpoints que obtengan sus datos de servidores internos y enriquecerlos añadiendo datos de terceros de una API externa como GitHub.

Un objeto backend es un array de todos los servicios a los que se conecta un endpoint. Define la lista de nombres de host a los que se puede conectar y la URL para enviar y/o recibir datos. Si un backend tiene más de un host el array dependerá del balanceador de la carga y de cómo este configurado.

Cuando se accede a un endpoint de KrakenD, el motor solicita todos los backends definidos en paralelo (a menos que se use un proxy secuencial) y el contenido se combina y se agrega (a menos que se use el proxy con la codificación no-op). El contenido devuelto se analiza según la codificación y la configuración extra.

Dentro del backend necesitaremos crear un objeto para cada uno de los servicios utilizados por el endpoint declarante. La combinación de host y url_pattern es la url completa que KrakenD utilizara para obtener sus servicios.

La configuración más simple que los backends requerirán será:

- Host: Array con todos los hosts disponibles para realizar las peticiones.
- Url_pattern: La ruta dentro del servicio, sin protocolo ni host ni método.

En mi configuración además de estas dos mencionadas he usado dos atributos de configuración más:

- Encoding: Define la codificación necesaria para establecer como analizar la respuesta recibida del backend. Por defecto es el valor de la codificación de su punto final o json si no se define en otro lugar. Por defecto es json, pero los posibles valores son:
    - Safejson
    - Xml
    - Rss
    - String
    - No-op

- Method: Se trata del método enviado a este backend en mayúsculas. No es necesario que el método definido en el endpoint coincida con el del backend. Si se omite el valor se utiliza el método del endpoint. Por defecto es GET, pero los posibles valores son:
    - POST
    - PUT
    - PATCH
    - DELETE

Backend headers [34]

La política por defecto de KrakenD con respecto a los errores y códigos de estado es ocultar al cliente cualquier detalle del backend, esto incluye cabeceras y errores, excepto cuando se utiliza la codificación no-op.

La filosofía detrás de esto es que los clientes tienen que ser desacoplados de sus servicios subyacentes, como una API Gateway debe hacer.

Aun así, podemos anular esta política por defecto y devolver detalles de error del backend con diferentes estrategias.

1. Estrategia por defecto

Degradación agradecida de la respuesta (cuando se utiliza la codificación no-op): Esta estrategia soporta múltiples backends. Los códigos de estado HTTP devueltos al cliente son 200 o 500 independientemente de los códigos de estado del backend.

El cuerpo se construye, fusiona y manipula a partir de los backends en funcionamiento.

2. Cabeceras y los errores manejados enteramente por el backend.

Muestra los códigos de estado HTTP, las cabeceras y el cuerpo de las response tal y como se devuelve del backend. Máximo uno. No se pueden manipular los datos

3. Devuelve el código de estado HTTP de un único backend.

Establece un cuerpo vacío cuando hay errores, pero reserva los códigos de estado HTTP del backend. Para esta estrategia se puede usar la herramienta *return_error_code.*

Cuando solo tenemos un backend y se utiliza una codificación diferente a no-op, podemos optar por la estrategia que devuelve el código de estado HTTP original al cliente.

La pasarela bloquea los códigos de estado HTTP del backend por defecto por varias razones en las que se incluyen seguridad y consistencia. Aun así, si queremos utilizar el código de estado HTTP del backend podemos activar la herramienta *return_error_code.*

El cuerpo del error estará vacío a menos que también lo complementemos con la herramienta return_error_msg. La opción segura de KrakenD es que todos los errores generados en la pasarela no sean devueltos al cliente en el cuerpo de la respuesta.

Pero esta opción puede cambiar estableciendo la herramienta return_error_msg a true, cuando se produce un error que se devolverá al cliente con el motivo del fallo. El error se escribe en el cuerpo tal cual.

4. Devuelve el código de estado HTTP y el cuerpo de error de un único backend.

Esta estrategia devuelve el cuerpo del error y el código de estado de un único backend. Para ello se utilizan las herramientas *return_error_code y return_error_msg.*

5. Devuelve los errores de backend en una nueva clave.

Soporta múltiples backends. Como la opción de la degradación agradecida, pero añade una nueva clave de error en el cuerpo cuando el backend falla. Para poder mostrar los errores al cliente podemos mostrarlos en la respuesta para ello debemos activar la opción *return_error_details* en la configuración del backend.

```
{
    "url_pattern": "/return-my-errors",
    "extra_config": {
        "backend/http": {
            "return_error_details": "backend_alias"
        }
    }
}
```

El error lo recibiremos en un formato similar al de la siguiente imagen:

```
{
    "error_backend_alias": {
        "http_status_code": 404,
        "http_body": "404 page not found\\n"
    }
}
```

6. Muestre la interpretación del error por parte de la pasarela, pero no el cuerpo real del error.

El cliente ve la interpretación de la pasarela como código de estado inválido o límite de contexto excedido, pero no se ve el error real entregado por el backend. El código de estado es siempre 200 o 500.

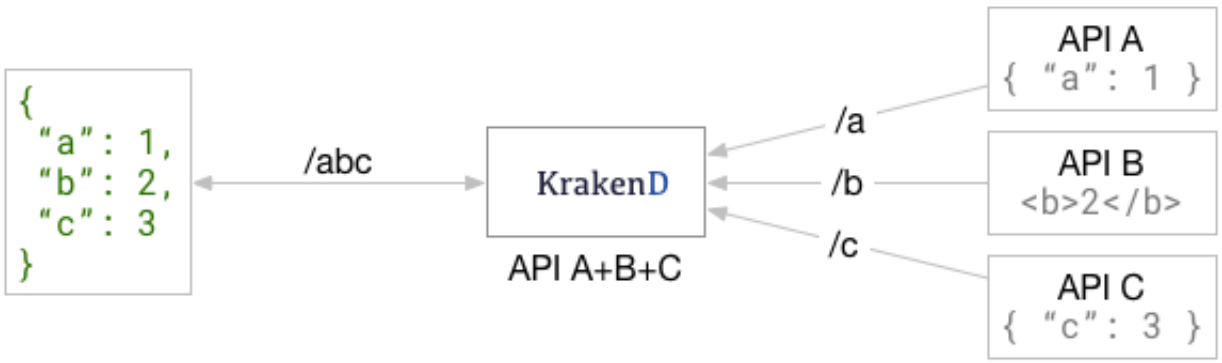
Response Manipulation [35]

KrakenD te permite realizar manipulaciones de las responses del backend. Las manipulaciones que son posibles son:

Agregación y fusión

Cuando hay más de un backend conectado a un endpoint que no utiliza la codificación no-op, el Gateway agregará y combinará de manera automática las respuestas de todos los backends en la respuesta final.

Por ejemplo, si tenemos 3 servicios API diferentes que exponen los recursos /a, /b y /c. Deseamos revelarlos todos juntos en el endpoint /abc de KrakenD conseguiríamos una respuesta así:



La operación de fusión da prioridad a la experiencia de usuario y a la capacidad de respuesta. Esta operación hace lo posible por obtener todas las partes necesarias de los backends involucrados y devolver el objeto compuesto en la mayor brevedad posible.

KrakenD marca el resultado de la operación de fusión con el header X-KrakenD-Completed, siendo verdadero si todos los backends funcionaron correctamente o falso si algunos fallaron. Cuando ninguno tuvo éxito, la pasarela devuelve un código 500.

Al tener varios backends de los cuales estamos esperando una respuesta KrakenD no se quedará atascado hasta que todos decidan responder. En un Gateway fallar rápido es mejor que tener éxito lentamente y KrakenD nos asegura que esto pasará.

El valor de timeout puede ser introducido en la definición de cada uno de los endpoints o globalmente colocando el timeout en la raíz del fichero de configuración.

Si KrakenD espera a que respondan todos los backends y se alcanza el tiempo de espera, la respuesta estará incompleta ya que no se habrán podido obtener todos los datos antes del timeout. Sin embargo, todas las partes que se pudieran recibir antes del timeout se devolverán en la respuesta.

Si faltan partes de la respuesta la cabecera de caché no existirá ya que no queremos que se almacenen en caches respuestas incompletas.

En todo momento la cabecera X-Krakend-Completed contiene un booleano que indica si todos los backends han devuelto su contenido por lo que el booleano será **true** o si se trata de una respuesta incompleta el booleano será **false**.

Filtrado

Al crear un endpoint de KrakenD puedes decidir si devolver todos los campos del backend o especificar cuales están permitidos a través de una lista de permitidos o denegados.

Puedes hacer uso de esta herramienta por muchas razones sin embargo la que se recomienda por el propio KrakenD es para ahorrar ancho de banda al usuario, proporcionar al cliente lo que necesita y disminuir tiempos de carga y renderizado.

Puedes elegir dos estrategias de filtrado:

- Denegar lista (Deny)
- Lista de permitidos (Allow)

Deny

KrakenD eliminará de la respuesta del backend todos los campos coincidentes, distinguiendo entre mayúsculas y minúsculas, definidos en una lista y devolviendo así los que no coincidan.

Para excluir un campo de la respuesta, añada bajo la configuración del backend un array deny con todos los campos que no desea mostrar.

Los campos de denegación también pueden estar anidados. Se utilizará un punto como separador de niveles. Por ejemplo, el campo a1 en la siguiente respuesta JSON { "a": { "a1": 1 } } puede añadirse a la lista de denegación como a.a1.

Allow

El filtro de lista de permitidos de KrakenD incluirá en la respuesta del endpoint solo aquellos campos que coincidan exactamente con su elección.

Los campos permitidos de su elección también pueden ser anidados. Utilice un punto como separador de nivel. Por ejemplo, el campo a1 en la siguiente respuesta JSON { "a": { "a1": 1 } } puede definirse como a.a1.

¿Allow o Deny?

A la hora de filtrar, hay que elegir entre la lista de denegados o de permitidos ya que las dos operaciones no pueden coexistir ya que sus comportamientos son contradictorios.

Aunque desde el punto de vista del rendimiento, la lista de denegados es ligeramente más rápida que la de permitidos.

Agregación

KrakenD puede agrupar las respuestas de los backends dentro de diferentes objetos. Cuando se establece un atributo de grupo para un backend, en lugar de colocar los atributos de respuesta en la raíz de la respuesta KrakenD crea una nueva clave y encapsula la respuesta en su interior.

Al agrupar diferentes respuestas backend no debemos compartir el mismo nombre para varios grupos, ya que el backend más lento sobrescribirá la respuesta con el mismo grupo.

Mapeo

El mapeo, o también conocido como renombrado, te permite cambiar el nombre de los campos de las respuestas generadas, para que tu respuesta compuesta sea lo más parecida posible a tu caso de uso sin cambiar una línea en ningún backend.

En la sección de mapeo, mapea el nombre del campo original con el nombre deseado.

Un ejemplo claro podría ser n lugar de mostrar el campo email queremos nombrarlo personal_email.

```
{
  "endpoint": "/users/{user}",
  "method": "GET",
  "backend": [
    {
      "url_pattern": "/users/{user}",
      "host": [
        "https://jsonplaceholder.typicode.com"
      ],
      "mapping": {
        "email": "personal_email"
      }
    }
  ]
}
```

Target

Es muy frecuente que en algunos endpoint los datos se nos devuelvan dentro de un campo genérico como data o content. Si no deseamos que los datos estén dentro de ese nivel de encapsulamiento haremos uso de esta herramienta.

Utilizando target podremos capturar el contenido dentro de estos contenedores genéricos y así poder evitarnos un nivel más de encapsulamiento en tus respuestas.

Circuit breaker [36]

El Circuit Breaker es una máquina de estado directa en medio de la petición y la respuesta que monitoriza son los fallos del backend. Cuando alcanzan un umbral configurado, el disyuntor impedirá él envió de más tráfico a un backend que este fallando, aliviando su presión en condiciones difíciles.

Cuando KrakenD exige más rendimiento del que su API puede ofrecer correctamente, el mecanismo disyuntor detectara los fallos y evitara estresas sus servidores al no enviar solicitudes que probablemente fallaran. También es útil para hacer frente a los problemas de red y otros problemas de comunicación mediante la prevención de demasiadas solicitudes de morir debido a los tiempos de espera, etc.

Es importante destacar que el número de errores máximos son errores consecutivos, y no el total de errores en el periodo. Este enfoque funciona mejor cuando tu tráfico es variable, ya que se basa en un patrón probabilístico y no se ve afectado por el volumen que puedas tener.

El interruptor automático está disponible en el espacio de nombres qos/circuit-breaker dentro de la clave extra_config. Un ejemplo de cómo he montado mi circuit breaker para mis endpoints es

```
"qos/circuit-breaker": {
  "interval": 60,
  "timeout": 10,
  "max_errors": 10,
  "name": "CB-Wallet",
  "log_status_change": true
}
```

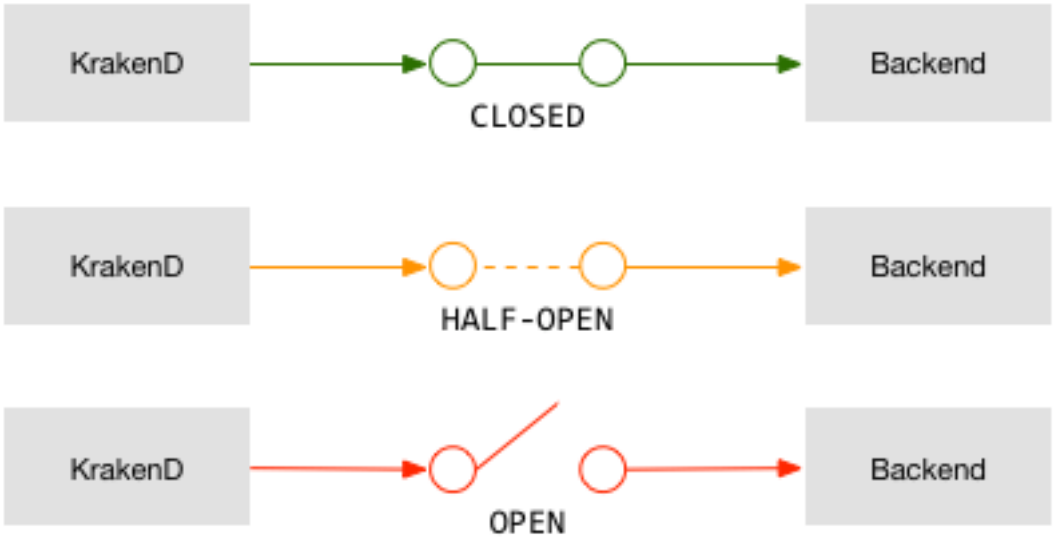
```
"qos/circuit-breaker": {
  "interval": 60,
  "timeout": 10,
  "max_errors": 10,
  "name": "CB-Coin",
  "log_status_change": true
}
```

Los atributos que podemos observar en la definición del circuit breaker es:

- Interval: Ventana de tiempo en la que cuentan los errores, en segundos.
- Timeout: Durante cuántos segundos esperará el disyuntor antes de volver a probar si el backend está operativo.
- Max_errors: El número consecutivo de errores dentro de la ventana de intervalo para considerar que el backend no es saludable. Un error es cualquier respuesta sin un código de estado de éxito (20x) o ninguna respuesta.
- Name: Un nombre amigable para seguir la actividad de este interruptor en los registros.
- Log_status_change: Si registrar o no los cambios de estado de este interruptor.

Como funciona el Circuit Breaker

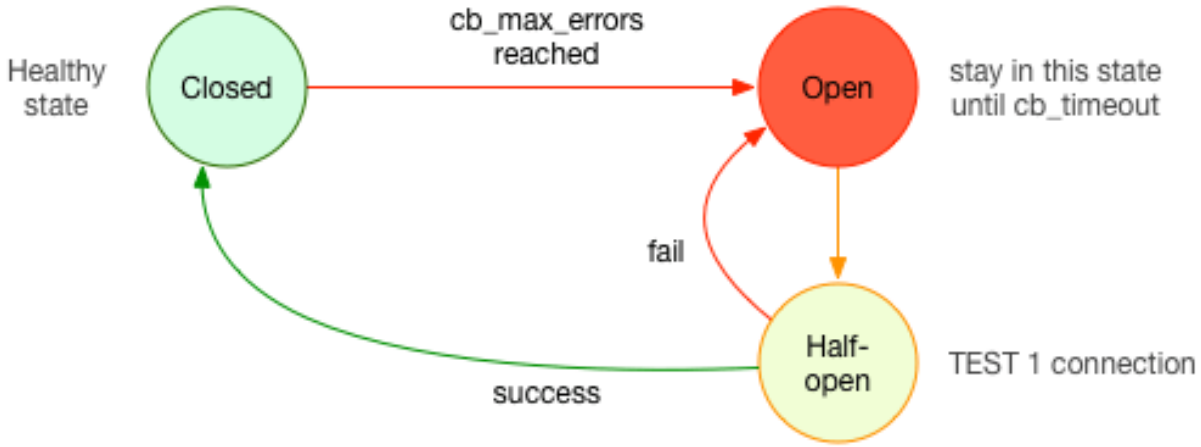
Para explicar el funcionamiento del Circuit Breaker podemos imaginar el estado del disyuntor como un componente eléctrico, en el que un circuito abierto significa que no hay flujo de electricidad entre los extremos y uno cerrado que el flujo es normal.



El disyuntor comienza con el estado CERRADO, lo que significa que la electricidad puede fluir hacia los extremos ya que se consideran operativos.

Cuando el componente observa que el estado de las conexiones con el backend con una tolerancia a fallos (max_errors) durante un intervalo del tiempo (interval) detiene toda la interacción con el backend durante el tiempo de espera cambiado su estado a ABIERTO.

Tras esperar el intervalo de tiempo el estado cambia a HALF_OPEN y permite solo pasar una conexión para probar el backend. En este estado si la conexión falla el estado vuelve a abierto y el interruptor volverá a esperar N segundo para probarlo de nuevo. Sin embargo, si esta conexión tiene éxito volverá a estar en el estado CERRADO y el sistema se considerará operativo.



Transiciones del disyuntor

- CERRADO: Es el estado inicial, el sistema está operativo y enviando conexiones al backend.

- ABIERTO: Cuando se supera el número consecutivo de errores soportados desde el backend (max_errors), el sistema cambia a Abierto durante N segundos que es el tiempo de espera.

- SEMI_ABIERTO: Una vez superado el tiempo de espera, se cambia a este estado y permite el paso de una conexión al backend. Si esta conexión es satisfactoria el estado se cambiará a CEERADO y se considera que el backend vuelve a estar operativo. Pero si falla volverá al estado ABIERTO y esperará otro tiempo de espera.

Modify request [37]

El componente krakend-martian permite transformar peticiones y respuesta mediante una definición e el fichero de configuración.

Puede utilizar esta herramienta cuando queramos interceptar la petición del usuario final y hacer modificaciones antes de pasar el contenido a los backends. También, al revés, transformar la respuesta de los backends antes de pasarla al usuario.

Martian nos brinda infinitas posibilidades de controlar lo que entra y sale de la pasarela. Como, por ejemplo:

- Establecer una nueva cookie.
- Añadir, eliminar o cambiar cabeceras específicas.
- Añadir cadenas de consulta antes de realizar la solitud de backend.

Hay cuatro interacciones diferentes con Martian:

1. Modificadores: Cambiar el estado de una solicitud o respuesta.
2. Filtros: Añadir una condición para ejecutar un modificador
3. Grupos: Agrupar varias operaciones para ejecutarlas en el orden especificado en el grupo.
4. Verificadores: Seguimiento del tráfico de red.

Algunos ejemplos que he utilizado en mi código son:

Añadir en el header una cabecera más:

```
"modifier/martian": {
  "header.Modifier": {
    "scope": ["request", "response"],
    "name": "Accept",
    "value": "application/json"
  }
},
```

Añadir en el header una request pasada completamente por Martian:

```
"modifier/martian": {
  "header.Modifier": {
    "scope": ["request"],
    "body": "{{- `{"grant_type": "password",
    "client_id": "4",
    "client_secret": "XbGSNyZiRTpGWQYiB5hcseoMYnga0zMFejT8J8T5",
    "username": "alba.126427@e.unavarra.es",
    "password": "asiertfg"}`
     | b64enc -}}"
  }
}
```

AUTHORIZATION

Introducción

La autorización es una parte del sistema operativo que protege los recursos del sistema permitiendo que sólo sean usados por aquellos consumidores a los que se les ha concedido autorización para ello

La especificación Web Token es un estándar de la industria para representar una conexión segura entre dos partes. El JWT es un objeto codificado en base64 que contiene clave-valor de atributos firmados por una autoridad de confianza.

Cuando un token JWT securiza uno o un grupo de endpoints, las solicitudes al API deben de proporcionar un token para la verificación. Esta verificación de token tiene lugar en cada solicitud incluyendo una verificación de la firma, garantía de que su emisor y audiencia son correctos para entrar al endpoint.

Sólo en el caso de que el token sea válido y pase todas las comprobaciones, el usuario está autorizado a acceder al endpoint y continuar con la petición.

KrakenD implementa el modelo de firma JWT como el de validación JWT para proteger los endpoints de usuarios no deseados que no tienen derecho a utilizar la información, reforzando la seguridad.

El modo de funcionamiento de el Token Web es que dentro de este transporta la información que sus usuarios finales pasan al sistema para ser reconocidos como usuarios legítimos.

Un token JWT es una cadena codificada en base64 con la estructura header, payload y signature.

En el header tenemos el encabezado donde indicamos el algoritmo, el tipo de token y el Kid (para mi proyecto muy importante).

La sección del payload es donde aparecen los datos de usuarios y sus privilegios además de toda la información que queramos añadir o los datos que creamos convenientes.

En la signatura como su nombre indica se trata de una forma que nos permite verificar si el token es válido. Esta es la parte más importante de este token ya que si estamos tratando de hacer una comunicación segura entre dos partes y podemos coger cualquier token y ver en su contenido que clase de seguridad estamos implantando en nuestros endpoints.

Por ello la seguridad es una de las partes más importantes de mi informe y voy a explicar cómo he resuelto yo este problema para mi proyecto base. Ya que gracias a KrakenD y a una pequeña implementación en nuestro proyecto podemos asegurar nuestra API que no tienen ningún tipo de seguridad para que terceras personas no puedan obtener ningún tipo de dato.

Desarrollo

Para solucionar el problema que se me planteaba sobre la seguridad con KrakenD he decidido usar Laravel Passport.

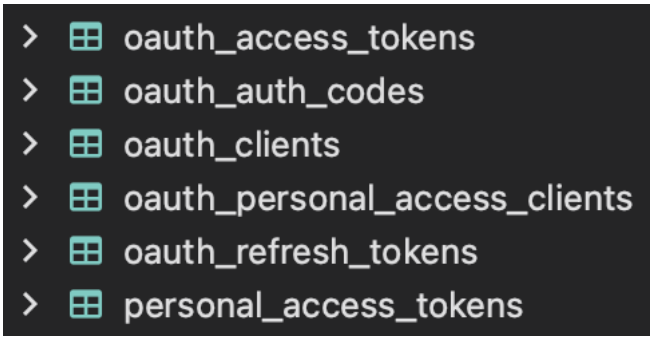
Laravel Passport es una herramienta más de nuestro framework que nos provee de una completa implementación del servidor OAuth2.

OAuth2.0 es el framework que nos ayuda a crear una aplicación de terceros para obtener acceso limitado a un servicio HTTP, ya sea en nombre del propietario de un recurso o permitiendo que la aplicación de permiso a un tercero a hacer uso de estos servicios.

Los pasos que he seguido para el desarrollo han sido…

Mediante vía Composer he descargado todas las dependencias correspondientes a la herramienta Passport que necesitaba mediante el comando -> *`Composer require laravel/passport`*

El proveedor de servicios de Passport tiene su propia migración de base de datos, lo que quiere decir que después de descargar todos los paquetes tendremos que descargar la base de datos necesaria para el OAuth2 mediante *`php artisan migrate`*. Una vez que se complete tendremos la migración completa de las tablas que necesitamos para almacenar todo lo relacionado con OAuth.



Estas son todas las tablas que necesitamos para el OAuth2.

Las tablas que más vamos a usar y son relevantes son:

-Oauth_access_tokens: En ella se almacenará los accesos de los clientes a las a los clientes de Oauth2, es decir, que id de usuario tiene un Access token a un oauth client.

- Oauth_clients: En ella se almacenarán todos los clientes o aplicación que están registrados en nuestro servidor OAuth2.

Los Access tokens son credenciales utilizadas para dar permisos a un usuario de acceso a recursos protegidos. Es una cadena que representa una autorización emitida al cliente. Este proporciona una capa de abstracción ya que puede sustituir diferentes formas de autenticación como por ejemplo nombre de usuario y contraseña con un único token emitido por el servidor de recursos.

Una vez que hemos migrado la base de datos necesitamos correr *php artisan passport:instal`* para crear las llaves de encriptación que necesitaremos para generar nuestros tokens de acceso seguro.

Gracias a Laravel tenemos un modelo de un usuario creado por defecto, el cual vamos a usar para la autenticación. Para ello deberemos de cambiar una de las librerías que tiene definida por defecto, de *Laravel\Sanctum\Passport* a *Laravel\Passport\Passport*.

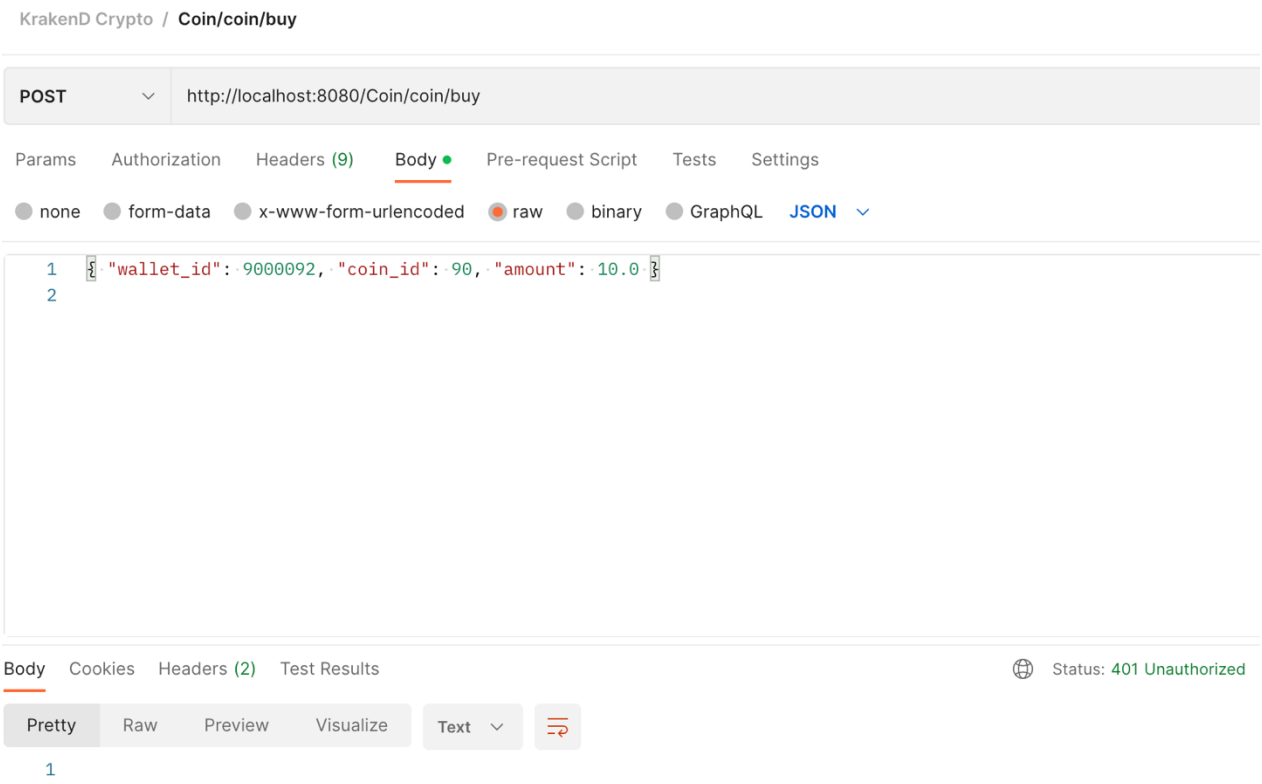
Finalmente, en nuestro archivo de configuración *auth.php* definiremos el driver que usaremos. Definiéndolo tal que *'api' => ['driver' => 'passport', 'provider' => 'users'].*

Para crear un nuevo oauth client usaremos *php artisan passport:client –password* con este comando conseguiremos crear una aplicación para la cual solicitaremos acceso.

```
➜  cryptocurrencies_wallet_tfg git:(master) ✗ php artisan passport:client --passwor

 What should we name the password grant client? [Laravel Password Grant Client]:
 > CryptoWalletTFG

 Which user provider should this client use to retrieve users? [users]:
  [0] users
 > 0

Password grant client created successfully.
Client ID: 4
Client secret: XbGSNyZiRTpGWQYiB5hcseoMYnga0zMFejT8J8T5
```

A este client le voy a llamar CryptoWalletTFG y le pongo que el proveedor de usuarios sea nuestro users.

Si ahora probamos en postman una llamada a un endpoint que es privado como por ejemplo el /Coin/coin/buy



Como era de esperar obtenemos un status code 401 Unauthorized y un cuerpo de response vacío.

Pero para antes de esto he debido definir en nuestro KrakenD una configuración, que he definido bajo el archivo coin_validator.tmpl [38]

```
"auth/validator": {
  "alg": "RS256",
  "jwk_local_path": "{{ .env.jwk_local_path }}",
  "cache": false,
  "operation_debug": {{ .env.is_development_env }},
  "disable_jwk_security": {{ .env.disable_jwk_security }}
}
```

Como podemos apreciar en la imagen he definido varios atributos, pero los importantes son 2:

1. Alg: el algoritmo que voy a usar para encriptar mi token y que usará para desencriptar el token para conseguir el acceso.

2. Jwk_local_path: Ruta al directorio donde tengo la llave pública de los tokens JWT.

Una vez explicado cómo hacemos la conexión entre KrakenD y el servidor OAuth2, debemos crear el token JWK. Para ello haremos uso del endpoint definido con ruta /Auth/token/v1.

Si hacemos la llamada con una request vacía obtendremos un 404 cuyo response será:

```
"error": "The authorization grant type is not supported by the authorization server."
```

En la request de la llamada al endpoint deberemos definir el campo *grant_type: password*, será password ya que he creado el OauthClient con el flag de password específicamente.

Añadiremos adicionalmente la auditoria o id de nuestro cliente y su secreto. Además de que deberemos de tener un usuario definido, en mi caso he utilizado mi correo de la universidad y una password inventada sencilla.

```
{
  "grant_type": "password",
  "client_id": "4",
  "client_secret": "XbGSNyZiRTpGWQYiB5hcseoMYnga0zMFejT8J8T5",
  "username": "alba.126427@e.unavarra.es",
  "password": "asiertfg"
}
```

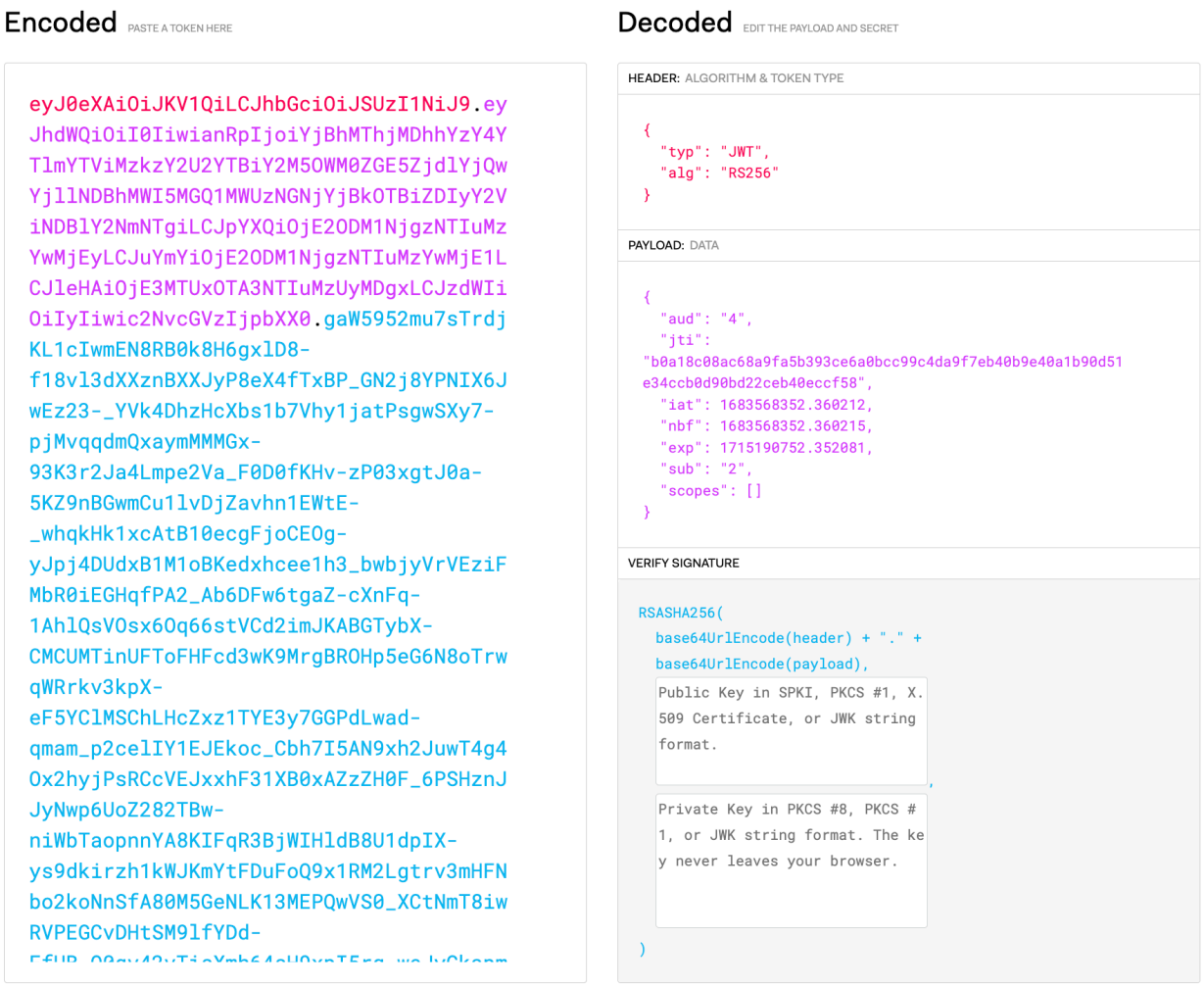
Al realizar la petición con esta request obtendremos una reponse donde tendremos varios campos definidos, nos quedamos con el Access_token.

Body Cookies Headers (12) Test Results — Status: 200 OK Time: 313 ms Size: 2.09 KB Save as Example

Pretty Raw Preview Visualize JSON


```
{
  "token_type": "Bearer",
  "expires_in": 31622400,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.
    eyJhdWQiOiI0IiwianRpIjoiYjBhMThjMDhhYzY4YTlmYTViMzkzY2U2YTBiY2M5OWM0ZGE5ZjdlYjQwYjllNDBhMWI5MGQ1MWUzNGNjYjBkOTBiZDIyY2ViNDBlY2NmNTgiLCJpYXQiOjE
    2ODM1NjgzNTIuMzYwMjEyLCJuYmYiOjE2ODM1NjgzNTIuMzYwMjE1LCJleHAiOjE3MTUxOTA3NTIuMzUyMDgxLCJzdWIiOiIyIiwic2NvcGVzIjpbXX0.
    gaW5952mu7sTrdjKL1cImwEN8RB0k8H6gxlD8-f18vl3dXXznBXXJyP8eX4fTxBP_GN2j8YPNIX6JwEz23-_YVk4DhzHcXbs1b7Vhy1jatPsgwSXy7-pjMvqqdmQxaymMMMGx-93K3r2Ja4
    Lmpe2Va_F0D0fKHv-zP03xgtJ0a-5KZ9nBGwmCu1lvDjZavhn1EWtE-_whqkHk1xcAtB10ecgFjoCEOg-yJpj4DUdxB1M1oBKedxhcee1h3_bwbjyVrVEziFMbR0iEGHqfPA2_Ab6DFw6tg
    aZ-cXnFq-1AhlQsVOsx6Oq66stVCd2imJKABGTybX-CMCUMTinUFToFHFcd3wK9MrgBROHp5eG6N8oTrwqWRrkv3kpX-eF5YClMSChLHcZxz1TYE3y7GGPdLwad-qmam_p2celIY1EJEkoc
    _Cbh7I5AN9xh2JuwT4g4Ox2hyjPsRCcVEJxxhF31XB0xAZzZH0F_6PSHznJJyNwp6UoZ282TBw-niWbTaopnnYA8KIFqR3BjWIHldB8U1dpIX-ys9dkirzh1kWJKmYtFDuFoQ9x1RM2Lgtr
    v3mHFNbo2koNnSfA80M5GeNLK13MEPQwVS0_XCtNmT8iwRVPEGCvDHtSM9lfYDd-FfUB_O0gv42vTisXmh64sH9xpI5rq_weJyGkapm_oSklbSU",
  "refresh_token":
    "def5020086085d582568b16f77e7a10242bda6e27ff3885196bfcc16589a5ab55a5b10bd7f05689da4d37ee4595ab343035f19a346dc160a2ce2926dc3a01743cfbbb5e4778545
    8b97edcccbdf123bc8078ffd3988c3371835e5897a82e3b0e9d32b9e6cb663d237b46a0fe3c683aef7671573ad3eec976bf5ef0e41358de63a0e7ee19ac87c200bb9bff93ac999f
    50f33f58abef0cf364cd180a8d6559429bc4bc550194027fc3af210c81cd2f51bd271e70deda6c12df4c3202e7fae16277fe66ab42fcdf80876892064894d62846cc5e5f7792b75
    8587c6e7fc1a1f952290c2018629ab6319f4f7d1c3755ae377b4ca002049dcb31647b4d2208ccf7058e89f0c15855bccf33d2efb10efe4c79de739aae5f945a30f2ba3713e5b7e8
    e5bd9706c4b17662ec457cd7693e63593210440b267b4d6ba226efe2bfc0ae4cd7c264c406e06972707c4ce55405de1ca1737c32e457da5e9368e2ef1f9d9885ff8cfa6"
}
```

Si copiamos el Access_token y nos vamos a jwt.io, que es una web en la que pasamos nuestro token JWT y nos lo desglosa en los 3 campos podemos observar los atributos que están definidos dentro de nuestra cadena en base64

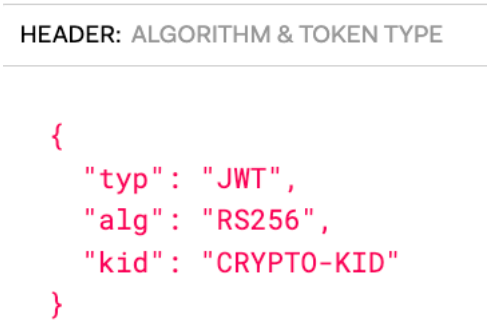


En nuestro header no aparece ningún KID, este es un atributo opcional que nos proporciona un identificador de key.

Si realizamos la llamada al endpoint con este Access_token seguiremos obteniendo un 401 ya que la firma del token JWT no concuerda con la llave que tengo creada. Por la falta de ese header KID.

Para ello investigando en las librerías internas del propio framework tenemos un fichero llamado AccessTokenTrait donde existe una función en la cual se define como se crea ese token JWT. Como deseamos que se genere un token con KID refactorizaremos esta función añadiéndole un header mediante *->withHeader('kid', 'CRYPTO-KID'))*.

Si realizamos este cambio y ejecutamos de nuevo el endpoint que genera el token y copiamos nuestro Access_token y lo añadimos al jwt.io.



Obtendremos que en el header ya está incluido nuestro kid.

Solo nos queda probar a realizar la petición con ese token y obtendremos ya acceso y habremos conseguido realizar la autorización de un usuario al OauthClient.

Para mi esta es la mejor solución que hay para conseguir una autorización de KrakenD, sin embargo, hay más formas de conseguirlo como por ejemplo mediante Oauth 2.0 Client Credentials. [39]

A través de la concesión de credenciales de cliente, KrakenD puede hacer un flujo de dos patas. Esto significa que la pasarela solicita al servidor de autorización un token de acceso antes de llegar a los endpoints protegidos del backend. El token se pasa en el header Authorization.

Las credenciales del cliente autorizan a KrakenD, como cliente, a acceder a los recursos protegidos.

Lo malo de esta forma de securiza nuestro servidor es que la conexión satisfactoria de las credenciales de un cliente para un backend significa que KrakenD puede obtener contenido protegido. Aun así, el endpoint que se ofrece al usuario final será publico mientras que no se proteja con JWT u otro mecanismo de autenticación de usuario final.

KARATE TESTS

Introducción

Siguiendo por el camino de las buenas prácticas, código limpio y tests. Vamos a testar nuestra API Gateway para ello utilizaremos Karate.

Karate es la única herramienta de código abierto que combina automatización de tests de API, pruebas de rendimiento e incluso automatización de la interfaz de usuario en un solo marco unificado. La sintaxis es independiente del lenguaje y fácil incluso para quienes no son programadores. Las aserciones y los informes HTML están integrados, y puede ejecutar pruebas en paralelo para aumentar la velocidad.

No tienes que compilar código, simplemente escribir los test en una sintaxis simple y legible.

Instalación

Hay distintas formar de empezar a usar Karate sin embargo yo opté por descargarme su .jar y usarlo como un ejecutable.

Una vez descargado el .jar tenemos un archivo de configuración de Karate al que definí como karate-config.js. En él defino una variable urlBase que utilizaré repetidamente en vez de estar invocando continuamente a *http://localhost:8080*. También defino el connectTimeout y readTimeout para los tests.

Desarrollo

La estructura de carpetas que voy a seguir para definir los tests de mi API Gateway se compone de dos carpetas padre, *specs/karate*.

Dentro de nuestras carpetas padres vamos a definir dos más:

Fixtures: Donde guardaremos los templates de las responses que sean de un tamaño extenso y para que nuestros tests sean más legibles usare templates.

Tests: Donde guardaremos todos nuestros tests separados por nuestro dominio del proyecto base que se separaba en dos modelos Coin y Wallet.

Como se puede observar en cada clase de tests solo testamos una regla de negocio.

También se puede observar que varios de los ficheros de la carpeta Fixtures son repetidos, la razón por la que son repetidos es porque en un proyecto real estas reglas de negocio están en continuo cambio por lo que si solo tuviéramos un archivo para cada una regla de negocio y esta cambia deberían cambiar todas.
Los tests son muy básicos y sencillos.

```
@github
Scenario: A wallet with the specified ID was not found
Given url urlBase
And path '/Coin/coin/buy'
And request { wallet_id: 9000092, coin_id: 90, amount: 10.0 }
When method post
Then status 200
And match response == read('../../../fixtures/Response/Coin/Coin_buy/A_wallet_with_the_specific_id_was_not_found.json')
```

Como vemos tenemos un campo para el título que es el Scenario.

Posteriormente el esquema del test es:

Given url urlBase -> Donde se define el host en nuestro caso http://localhost:8080/

And path '/Coin/coin/buy' -> Donde añadimos el path que vamos a testar además del url ósea http://localhost:8080/Coin/coin/buy

And request { ... } -> Donde definimos la request necesaria o la request incompleta para testar cierto corner-case.

When method post -> Donde definimos el HTTP Method.

Then status 200 -> Comprobamos el HTTP Status Code devuelto después de realizar la petición a la url que hemos definido

And match response == read('../../example.json') -> Comprobamos la response devuelta después de realizar la petición a la url definida. Podemos observar que usamos una función read que nos ayuda a leer un archivo en el directorio que especificamos dentro del parámetro.

Para correr los tests he creado unas acciones en el Makefile que ejecuta todos mis test gracias al comando *java -jar bin/karate.jar specs/karate –output=/tmp –configdir=specs/karate* con el que corro todos los tests.

Una vez que se termina la ejecución de esta acción nos aparecerá una tabla resumen

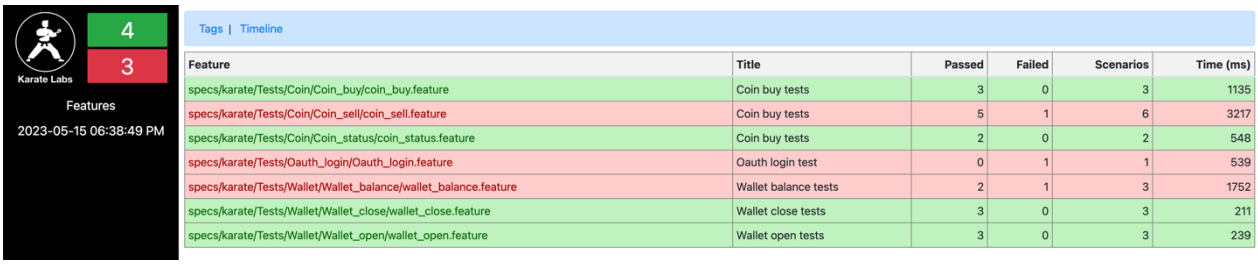
```
Karate version: 1.3.1
=======================================================
elapsed:   9,39 | threads:    1 | thread time: 7,64
features:     7 | skipped:    0 | efficiency: 0,81
scenarios:   21 | passed:    18 | failed: 3
=======================================================
```

Como podemos ver observamos que se han ejecutado 21 scenarios. De los cuales 18 son los que se han pasado satisfactoriamente y 3 tests han fallado.

Después de esta tabla obtenemos los tests que han fallado y por qué.

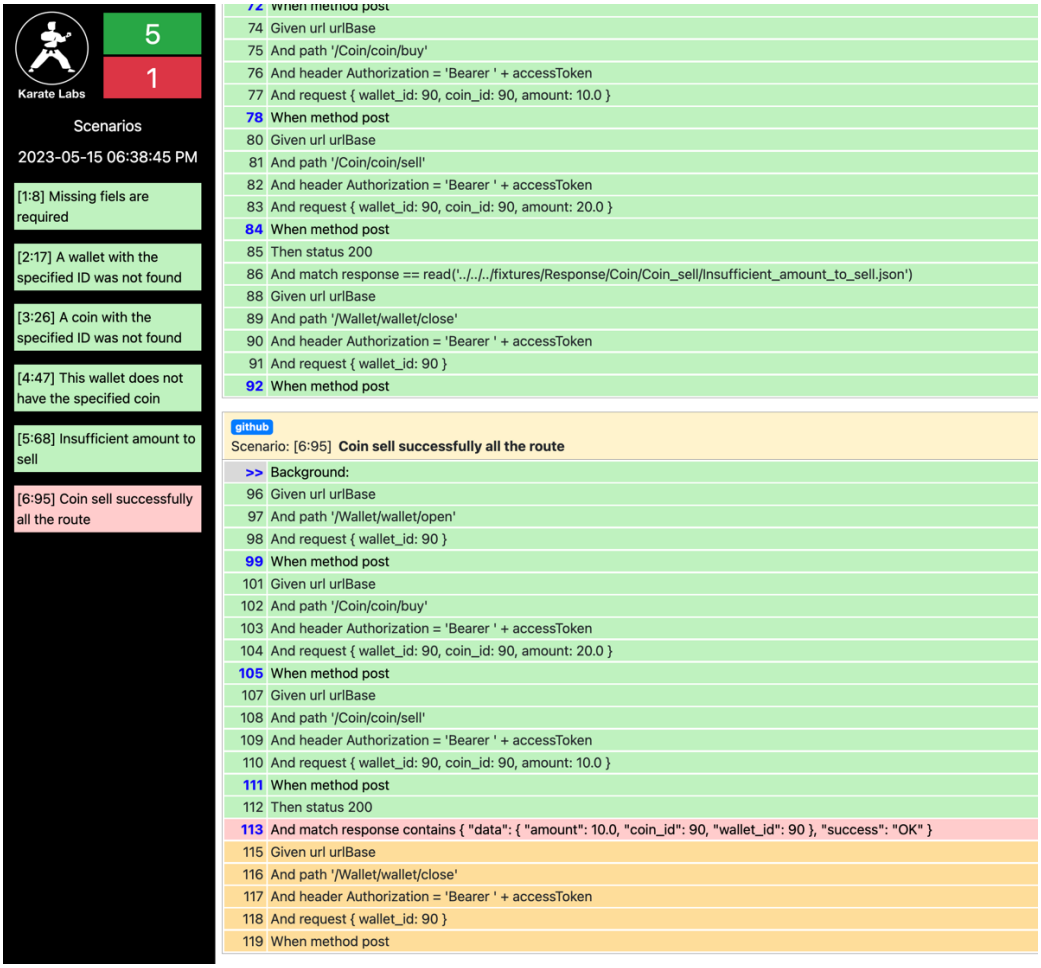
```
>>> failed features:
match failed: CONTAINS
  $ | actual does not contain expected | all key-values did not match, expected has un-matched keys - [data] (MAP:MAP)
  {"data":{"amount":20,"coin_id":90,"wallet_id":90},"success":"OK"}
  {"data":{"amount":10.0,"coin_id":90,"wallet_id":90},"success":"OK"}

    $.data | not equal | match failed for name: 'amount' (MAP:MAP)
    {"amount":20,"coin_id":90,"wallet_id":90}
    {"amount":10.0,"coin_id":90,"wallet_id":90}

      $.data.amount | not equal (NUMBER:NUMBER)
      20
      10.0

specs/karate/Tests/Coin/Coin_sell/coin_sell.feature:113
status code was: 404, expected: 200, response time in milliseconds: 250, url: http://localhost:8080/Auth/token/v1, response:
{"error":"Client authentication failed"}
specs/karate/Tests/Oauth_login/Oauth_login.feature:8
match failed: EQUALS
  $ | not equal | actual has 1 more key(s) than expected - {"data":{"balanceUsd":549419.6,"walletId":90},"success":"OK"} (MAP:MAP)
  {"data":{"balanceUsd":549419.6,"walletId":90},"success":"OK"}
  {"error_backend":{"http_status_code":404,"http_body":"{\"error\":\"There is not coins associated to the wallet with ID 90\"}"}}


specs/karate/Tests/Wallet/Wallet_balance/wallet_balance.feature:27
<<<

HTML report: (paste into browser to view) | Karate version: 1.3.1
file:///tmp/karate-reports/karate-summary.html
======================================================================
```

Como se ve al final de la imagen aparece un enlace a una web que se trata de un HTML Report que lo que nos hace es crearnos una tabla donde aparecen todos los tests, ejecutados satisfactoriamente y no.



Feature	Title	Passed	Failed	Scenarios	Time (ms)
specs/karate/Tests/Coin/Coin_buy/coin_buy.feature	Coin buy tests	3	0	3	1135
specs/karate/Tests/Coin/Coin_sell/coin_sell.feature	Coin buy tests	5	1	6	3217
specs/karate/Tests/Coin/Coin_status/coin_status.feature	Coin buy tests	2	0	2	548
specs/karate/Tests/Oauth_login/Oauth_login.feature	Oauth login test	0	1	1	539
specs/karate/Tests/Wallet/Wallet_balance/wallet_balance.feature	Wallet balance tests	2	1	3	1752
specs/karate/Tests/Wallet/Wallet_close/wallet_close.feature	Wallet close tests	3	0	3	211
specs/karate/Tests/Wallet/Wallet_open/wallet_open.feature	Wallet open tests	3	0	3	239

Aunque lo mejor de este panel interactivo es que puedes acceder dentro de cualquier clase de test por lo que vamos a acceder a la primera clase que falla.



Si accedemos a la parte que tenemos en rojo, es decir la parte que falla, obtendremos lo que se recibe en esa parte de la response y con lo que lo comparamos. De manera que si conseguimos arreglar el test obtendremos todo el test en verde y todas las clases en verde.

Feature	Title	Passed	Failed	Scenarios	Time (ms)
specs/karate/Tests/Coin/Coin_buy/coin_buy.feature	Coin buy tests	3	0	3	1099
specs/karate/Tests/Coin/Coin_sell/coin_sell.feature	Coin buy tests	6	0	6	3758
specs/karate/Tests/Coin/Coin_status/coin_status.feature	Coin buy tests	2	0	2	548
specs/karate/Tests/Oauth_login/Oauth_login.feature	Oauth login test	1	0	1	966
specs/karate/Tests/Wallet/Wallet_balance/wallet_balance.feature	Wallet balance tests	3	0	3	1247
specs/karate/Tests/Wallet/Wallet_close/wallet_close.feature	Wallet close tests	3	0	3	217
specs/karate/Tests/Wallet/Wallet_open/wallet_open.feature	Wallet open tests	3	0	3	217

Todos mis tests lo que he hecho es que sean consistentes y persistentes, esto significa que por mucho que mis tests se lancen en ningún momento la base de datos se verá alterada por ellos ya

que todo lo que crea una acción la siguiente lo anula. En el siguiente ejemplo se ve más claro, para el test de 'Coin Sell Successfully al the route'

```
Background:
* def accessToken = read('../../accesToken.txt')
* header Authorization = 'Bearer ' + accessToken
```

La primera parte de nuestros test se encarga de leer el fichero donde tengo almacenado el accessToken para satisfacer la autorización creada y pasarla en un header.

```
Given url urlBase
And path '/Wallet/wallet/open'
And request { wallet_id: 90 }
When method post
```

Lo siguiente que necesitamos para poder vender monedas es tener una cartera donde vamos a poder realizar todas las operaciones de compra/venta de monedas. Y en la cual podremos ver su balance

En este punto lo que podemos observar es que compramos 20 monedas de la moneda de ID 90 y la almacenamos en la cartera con ID 90 que es la que acabamos de crear. Para vender una moneda primero deberemos tenerla.

```
Given url urlBase
And path '/Coin/coin/buy'
And header Authorization = 'Bearer ' + accessToken
And request { wallet_id: 90, coin_id: 90, amount: 20 }
When method post
```

```
Given url urlBase
And path '/Coin/coin/sell'
And header Authorization = 'Bearer ' + accessToken
And request { wallet_id: 90, coin_id: 90, amount: 10 }
When method post
Then status 200
And match response contains { "data": { "amount": 10, "coin_id": 90, "wallet_id": 90 }, "success": "OK" }
```

Posteriormente ya llegamos a la parte importante de este Scenario. Venderíamos 10 monedas de ID 90 que están almacenadas en la cartera con ID 90. Comprobamos que el status code es un 200 y que la response es la esperada. Aunque en esta parte pensemos que en base de datos se nos va a quedar un record ya que compramos 20 y vendemos 10. En la siguiente parte se resuelve todo.

```
Given url urlBase
And path '/Wallet/wallet/close'
And header Authorization = 'Bearer ' + accessToken
And request { wallet_id: 90 }
When method post
```

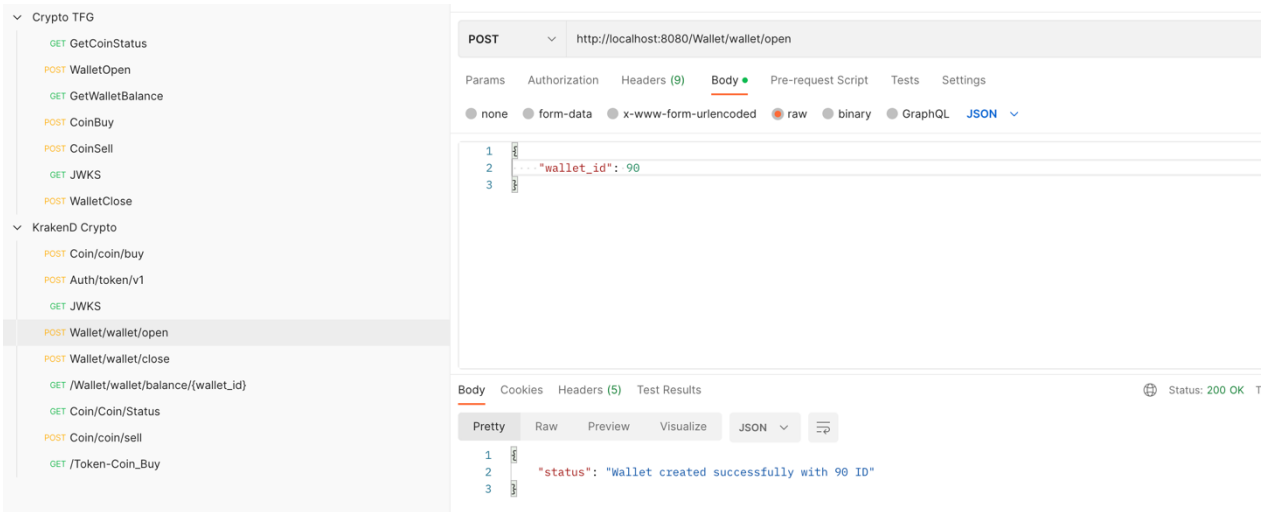
En esta parte lo que haremos será cerrar la cartera con ID 90. Haciendo uso de este endpoint lo que haremos será borrar el record de la base de datos Wallet. Pero a su vez también todos los records que haya de la cartera con ID 90 también se eliminarán. Ya que por definir una regla de negocio clara no tendría sentido no tener una cartera pero que si se guarden sus transacciones.

Extra

Una herramienta que me ha sido útil para ir depurando y entendiendo porque algunos tests me fallaban sin sentido es POSTMAN. Gracias a esta herramienta he creado dos colecciones de peticiones para ver los endpoints de mi API.

En la primera colección hacia las llamadas directamente a mi backend sin pasar por KrakenD.

En la segunda colección hacia las llamadas a mi KrakenD por lo que tenía que definir el path según lo hubiese definido en mis endpoints, además en este también debía de añadir un header de autenticación pasándole el Authorization: Bearer <Access_token>



TRAFFIC MANAGEMENT

KrakenD ofrece varias formas de proteger el uso de su infraestructura que pueden actuar en niveles muy diferentes.

El tipo más significativo de función de gestión de tráfico es el límite de tasa, que le permite estrangular el tráfico de los usuarios finales o el tráfico de KrakenD contra sus servicios backend.

Los límites de tasa cubren principalmente los siguientes propósitos:

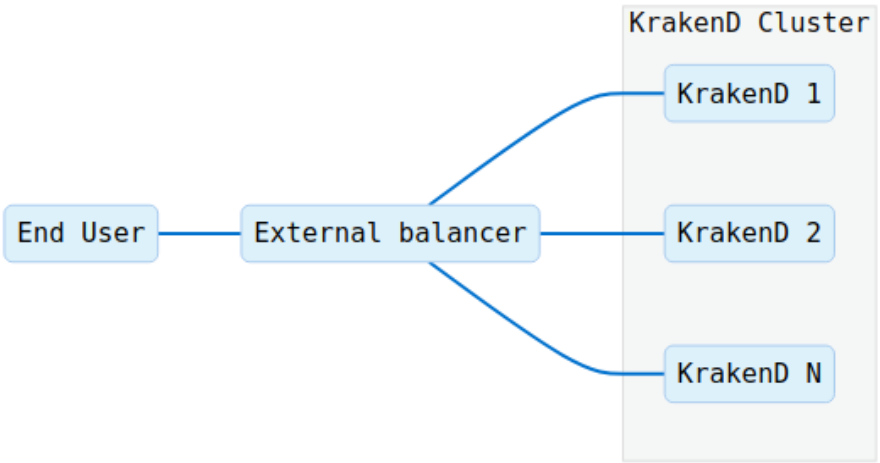
- Evitar estresar o inundar sus servicios backend con peticiones masivas (límite de tasa proxy).
- Establecer una cuota de uso para su API expuesta (límite de tasa de enrutador)
- Crear una estrategia sencilla de QoS para su API

Balanceador de carga [40]

La ubicación de un API Gateway está entre los consumidores API y sus servicios. Cuando hablamos de equilibrio de carga, podemos referirnos a ambos lados del Gateway: tráfico de entrada (usuario a pasarela) o tráfico de salida (pasarela a servicios).

Equilibrar el tráfico de entrada

Aunque en mi proyecto no he tenido la oportunidad de tener varios servidores se recomienda tener varios contenedores en producción para tener una alta disponibilidad. Además, se recomienda colocar un balanceador externo que sirva como único punto de contacto para que los clientes distribuyan el tráfico entrante a todos los nodos de KrakenD.



KrakenD no necesita ninguna configuración para recibir tráfico de entrada.

Balance del tráfico de salida

KrakenD se conecta a sus servicios utilizando la estrategia elegida para su balanceo.

El balanceador de carga puede funcionar para servicios internos y externos simultáneamente, y es irrelevante para KrakenD la ubicación física o la red siempre que se pueda alcanzar el destino.

Cuando escribimos la configuración implícitamente estamos estableciendo la estrategia de balanceo de la carga en base a cada backend dependiendo de la lista de hosts. Esto significa que puedes tener un endpoint que se conecte a un conjunto de backends balanceados mientras tienes otro endpoint que se relacione con un servicio balanceado externamente.

En esencia hay dos entradas relevantes en backend:

- Host: Array con todos los hosts posibles
- Sd: El sistema Service Discovery para resolver sus servicios backend.

Cuando tengas un único host no hará falta más que definir el host ya que Sd por defecto es static.

Timeouts [41]

KrakenD es una API Gateway que habla con servicios. Controlar estos tiempos de espera para diferentes aspectos es crítico y gracias a Krakend podemos afinar estos ajustes.

Los tiempos de espera puede aplicarse a:

- La duración de toda la tubería
- Los tiempos de espera relacionados con la petición HTTP

Tiempo de espera por defecto

El timeout en el nivel raíz se utiliza para aplicar un tiempo de espera por defecto para aquellos endpoints que no especifiquen uno.

La duración que escribas representa toda la duración de la tubería, por lo que cuenta el tiempo que todos tus backends tardan en responder y el procesamiento de todos los componentes involucrados en el endpoint.

Superado el plazo de contexto

KrakenD esperará el tiempo de espera especificado intentando obtener y procesar el contenido, pero si eso ocurre el usuario lo que recibirá será un código de estado HTTP 500 cuando no haya contenido que devolver.

Si se conecta más de un backend y al menos tiene una respuesta valida, el Gateway devolverá un código de estado 200 con una respuesta parcial.

IP Filtering

El plugin de filtrado IP permite restringir el tráfico a su pasarela API basándose en la dirección IP.

Funciona de dos maneras diferentes (permitir o denegar) donde usted define la lista de IPs (bloques CIDR) que están autorizadas a usar la API, o que tiene denegado su uso.

```
"ip-filter": {
    "CIDR": [
        "192.168.0.0/24",
        "172.17.2.56/32",
        "10.0.0.0/16",
        "127.0.0.1"
    ],
    "allow": true
}
```

El filtrado se aplica globalmente a todos los puntos finales o solo a puntos finales específicos dependiendo de la ubicación de la configuración.

TELEMETRY DATA

La observabilidad y la conexión en red son claves para tener éxito en un escenario de arquitectura de microservicios distribuidos y para ello necesitamos una herramienta de monitorización. Estas herramientas deben proporcionar al menos opciones para detectar las causas de los problemas, monitorización y detalle de las transacciones.

KrakenD puede exponer métricas muy detalladas para proporcionar un cuadro de mando de monitorización. Una de las soluciones de monitorización más ricas a nivel de métricas es el exportador nativo de Influx. Los dos componentes te permiten enviar métricas detalladas a InfluxDB y dibujarlas más tarde en nuestro dashboard Grafana preconfigurado.

Influx v2 [42]

Para utilizar Influx he utilizado Docker. Para ello he utilizado un archivo docker-compose.yml:

```yaml
version: "3"
services:
  influxdb:
    image: influxdb:2.4
    container_name: influx
    environment:
      - "DOCKER_INFLUXDB_INIT_MODE=setup"
      - "DOCKER_INFLUXDB_INIT_USERNAME=krakend-dev"
      - "DOCKER_INFLUXDB_INIT_PASSWORD=pas5w0rd"
      - "DOCKER_INFLUXDB_INIT_ORG=my-org"
      - "DOCKER_INFLUXDB_INIT_BUCKET=krakend"
      - "DOCKER_INFLUXDB_INIT_RETENTION=1w"
      - "DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=my-super-secret-auth-token"
    ports:
      - "192.168.0.25:8086:8086"
    volumes:
      - "./config/telemetry/influx:/docker-entrypoint-initdb.d"
```

En los campos db, username, y password de la configuración del componente refleje los mismos valores que en DOCKER_INFLUXDB_INIT_BUCKET, DOCKER_INFLUXDB_INIT_USERNAME, y DOCKER_INFLUXDB_INIT_PASSWORD correspondientemente.

Tras correr el container podemos acceder a InfluxDB en la dirección http://192.168.0.25:8086.

Para hacer las conexiones he accedido al interior de mi contenedor docker con:

Docker exec -it Influx /bin/bash

Una vez dentro de mi contenedor creare una configuración para mi Influx

influx config create --config-name krakend-config \

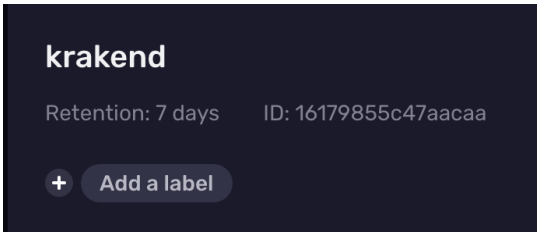
```
  --host-url http://192.168.0.25:8086 \
  --org my-org \
  --token my-super-secret-auth-token \
  --active
```

Una vez creada la configuración tengo que autorizar a mi Influx para mandar data y no recibir

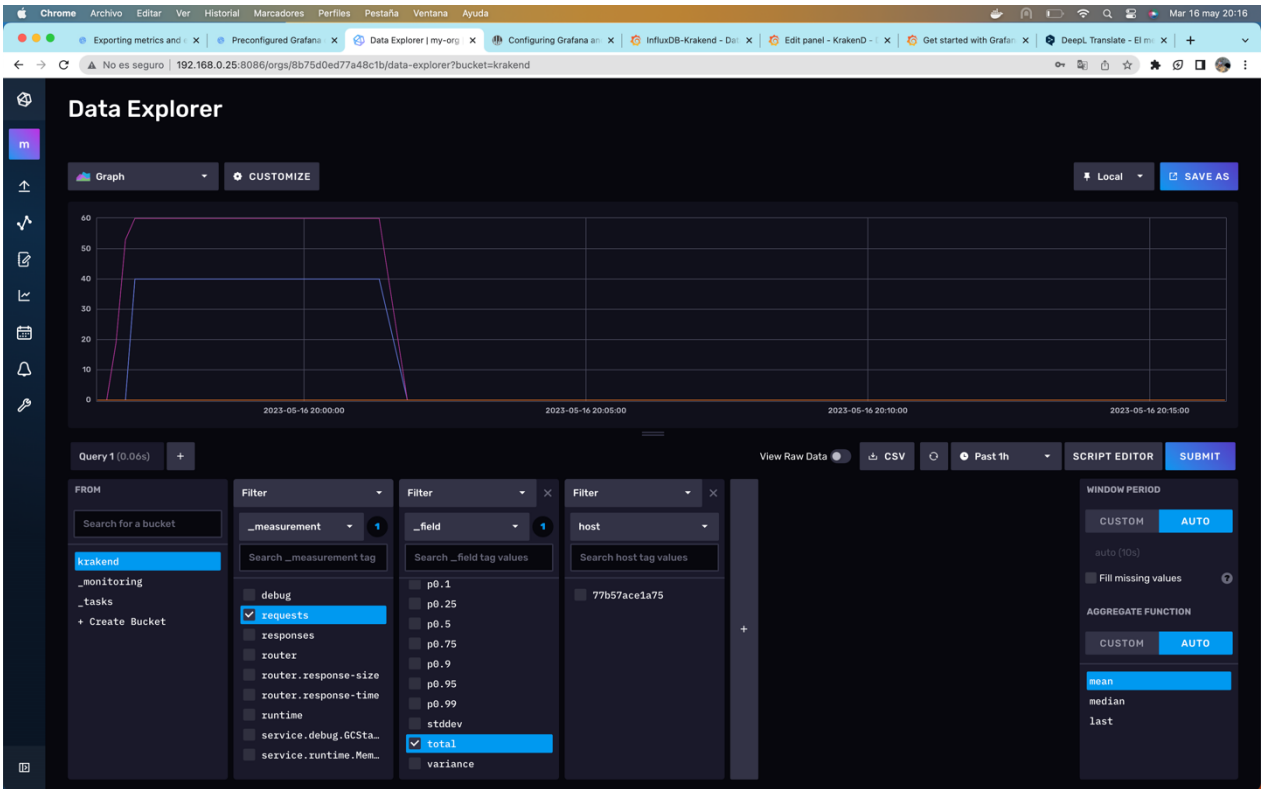
ERROR [SERVICE: InfluxDB] Couldn't write to server:
{"code":"unauthorized","message":"unauthorized access"}

Para ello:

```
influx v1 auth create \
  --read-bucket 07c0c133618bb08b \
  --write-bucket 07c0c133618bb08b \
  --username krakend-dev
```



Una vez que ya tenemos la conexión establecida gracias a la herramienta que he usado con anterioridad *hey* generare request y las visualizaré en InfluxDB.



El dashboard preconfigurado de Grafana para KrakenD ofrece información valiosa para entender el rendimiento de tus servicios y detectar anomalías en el servicio.

El dashboard es extenso y te ofrece métricas como:

Peticiones de usuarios a KrakenD
Peticiones de KrakenD a sus backends
Tiempos de respuesta

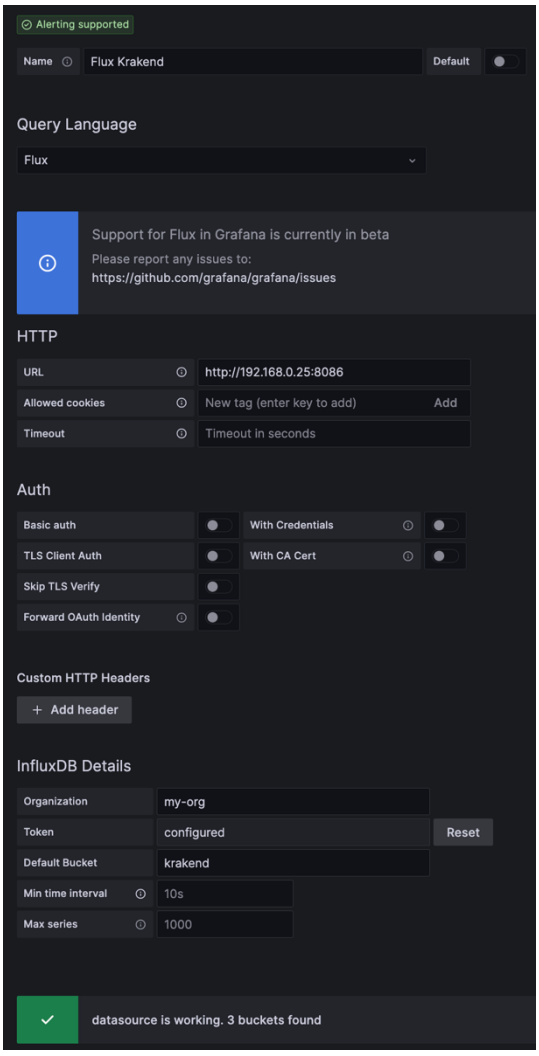
Uso de memoria y detalles
Endpoints y códigos de estado
Mapas de calor
Conexiones abiertas
Rendimiento
Distribuciones, temporizadores, recogida de basuras y un largo etcétera

Por lo que voy a explicar cómo he montado Grafana. He creado un contenedor para Grafana. [43]

```
grafana:
  image: grafana/grafana-oss
  container_name: grafana
  ports:
    - "192.168.0.25:3001:3000"
```

Una vez que tengo mi contenedor corriendo accedo a la web mediante un login en el que uso admin tanto para usuario como para contraseña.

Primero debemos de añadir Data Source que será nuestra conexión con InfluxDB.



En el name podemos añadirle cualquier nombre que nos parezca apropiado, en mi caso Flux Krakend.

En el query language -> Flux

En la url añado la url al contenedor que tengo sirviéndome InfluxDB.

En la organización -> my-org.

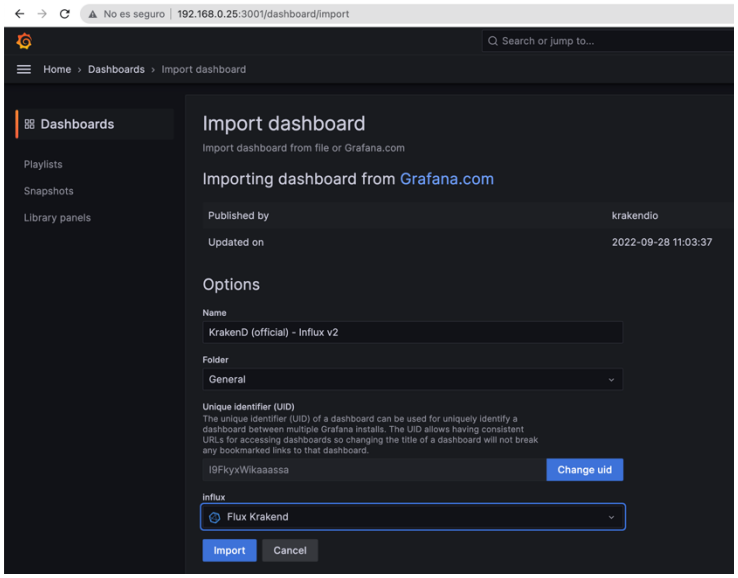
En el token -> my-super-secret-auth-token.

En el bucket -> krakend

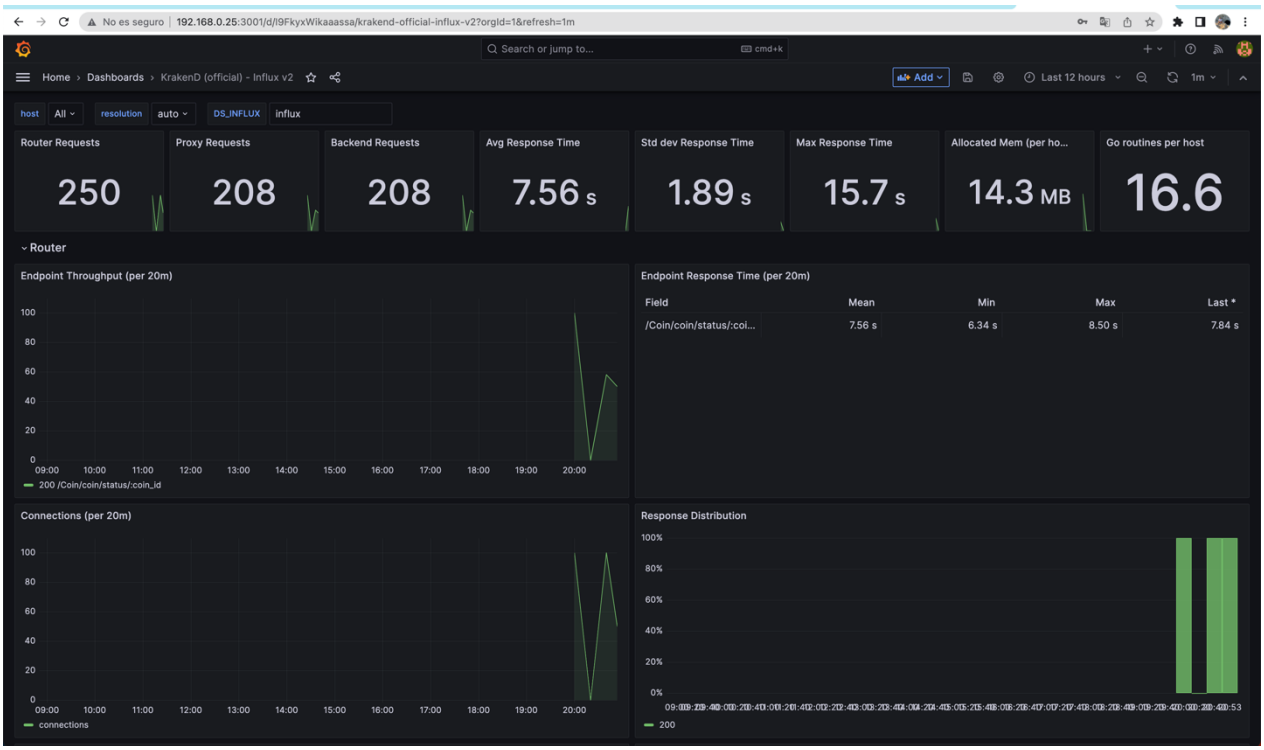
Estos campos los he sacado de las variables que he utilizado para lanzar mi contenedor de InfluxDB

Si nos fijamos en la parte inferior de la imagen encontramos un mensaje en el que nos da feedback sobre si nuestra datasource funciona. Además de los buckets que encuentra, estos son los que tengo creados en InfluxDB.

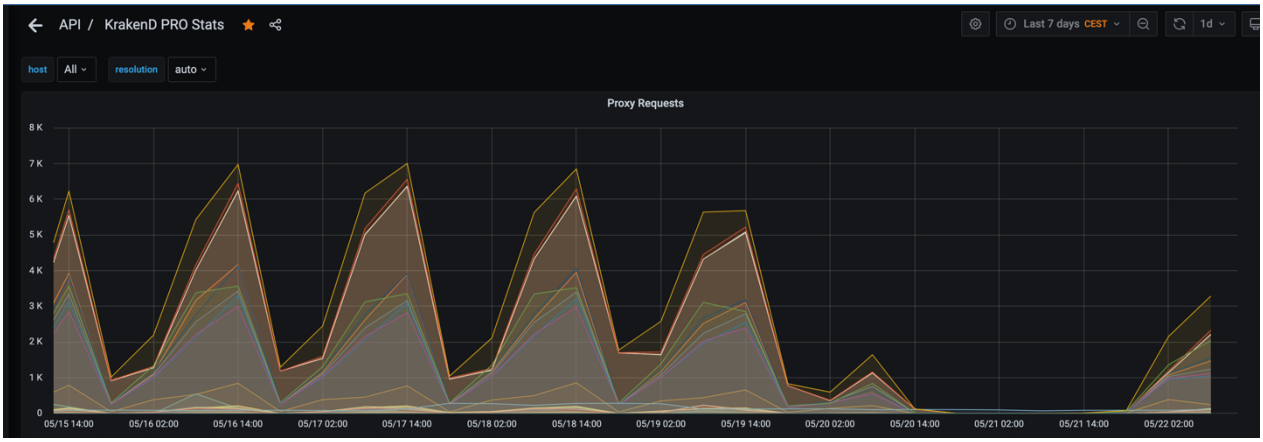
Para el dashboard donde vamos a plasmar nuestros datos para no tener que crearlo yo usaré una plantilla con id `17074.`



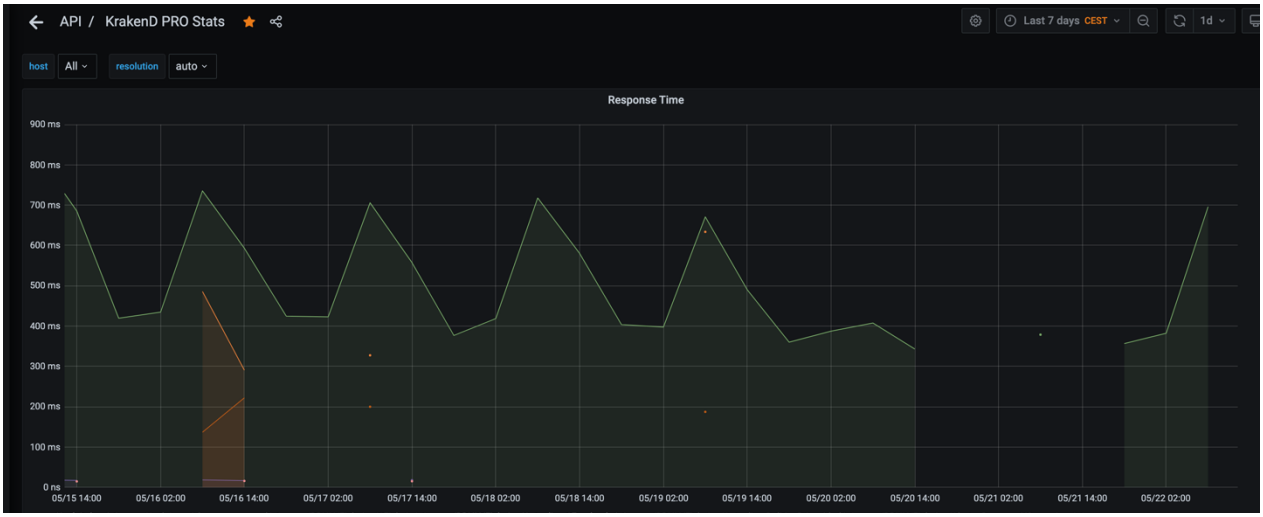
Si aceptamos la importación de este dashboard encontramos nuestro panel con los datos que hayamos generado desde que se lanzan los contenedores.



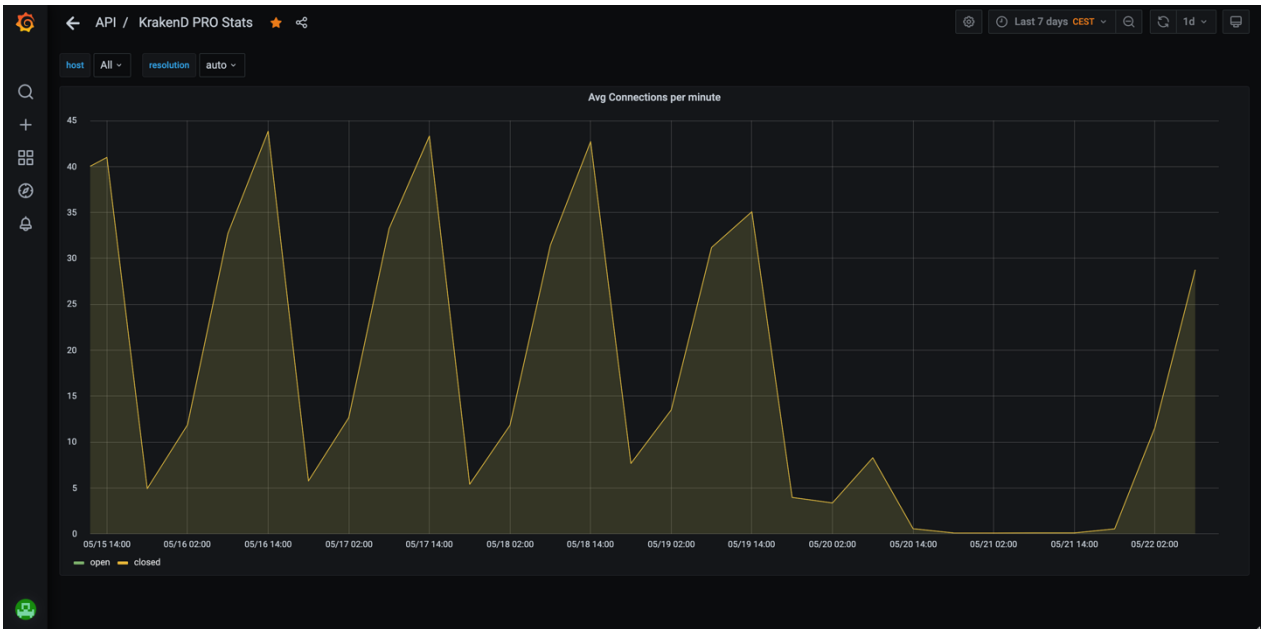
Estos datos son unos generados por mí haciendo uso de la herramienta *hey*, sin embargo, he conseguido datos de flujo de trafico de una empresa grande que vamos a llamar *SHS*.



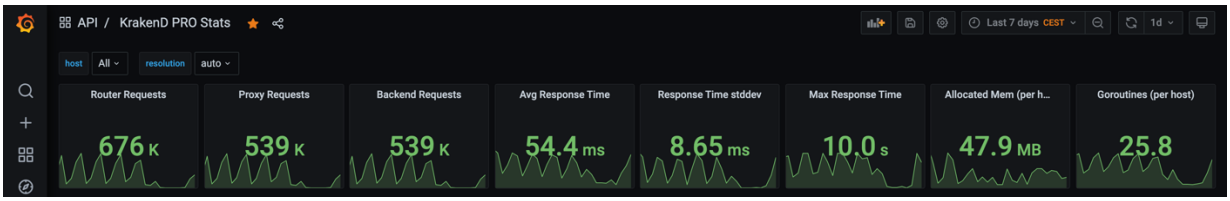
En esta grafica podemos observar las peticiones que se realizan al Proxy de la empresa *SHS*



En esta grafica podemos observar los tiempos de respuesta de un endpoint.



En esta grafica podemos observar la media de conexiones al KrakenD.



En esta imagen podemos observar los datos más relevantes obtenidos de la conexión de KrakenD

CONCLUSIÓN

En resumen, el informe presente ha evaluado el uso de KrakenD como API Gateway y demuestra porque es una elección sólida y ventajosa frente a sus competidores.

El uso de un API Gateway, proporciona numerosos beneficios. En primer lugar, actúa como punto de entada único para todas las solicitudes de API. Esto simplifica la gestión y control de acceso. Esto permite implementar políticas de seguridad centralizadas como autenticación y autorización, mejorando la protección de los recursos ya que garantiza que sólo los usuarios autorizados puedan acceder a las APIs.

Además, un API Gateway como KrakenD ofrece funcionalidades avanzadas de monitorización y análisis. Esto facilita la detección de cuellos de botella e identificación de problemas de rendimiento, lo que permite una toma de decisiones informada para mejorar la eficiencia y la calidad del servicio.

Otro de los beneficios del uso de un API Gateway es la simplificación de la gestión de versiones y la adaptación de diferentes APIs subyacentes. Al proporcionar una capa de abstracción entre los clientes y los servicios permite realizar cambios y actualizaciones en la API sin afectar directamente a los clientes. Esto aumenta la flexibilidad y facilita la evolución de las APIs a medida que los requisitos empresariales cambien.

En comparación con otras soluciones un API Gateway ofrece una serie de ventajas como proporcionar un punto centralizado de control y gestión lo que simplifica la administración y el seguimiento de las APIs. Además, permite una mayor agilidad y escalabilidad al facilitar la orquestación de los servicios.

En comparación con otros competidores del mercado, KrakenD destaca por su arquitectura ligera y de alto rendimiento que nos garantiza un procesamiento eficiente de las solicitudes de API, lo que desemboca en tiempos de respuesta más rápidos y una mayor capacidad de carga. Esta eficiencia es especialmente valiosa en entornos empresariales donde se gestionan volúmenes de solicitudes grandes.

En conclusión, el uso de KrakenD como API Gateway aporta importantes ventajas como una mayor seguridad, un mejor rendimiento y una gestión eficiente de las APIs. En definitiva, KrakenD es una herramienta valiosa y una elección recomendada para implementar un API Gateway en entonos empresariales que busquen gestionar y proteger sus APIs de manera efectiva.

LINEAS FUTURAS

En un futuro me gustaría realizar alguna funcionalidad y hacer un aporte a la comunidad de personas que usamos KrakenD. KrakenD es un API Gateway de código abierto que realiza su programación en Go.

Me gustaría empezar a utilizar algún otro lenguaje de programación que no sea PHP, y con la idea de realizar una funcionalidad para KrakenD en otro lenguaje lo conseguiría.

En primer lugar, debería aprender Go, pero la funcionalidad que me gustaría realizar es…

En un proyecto pequeño como el explicado en este informe no se ha podido apreciar, pero una de las cosas que me gustaría mejorar en KrakenD es la posibilidad de realizar subdirectorios en la configuración flexible.

Para aclarar esto cuando utilizo el comando para acceder a un template:

{{ template “coin_validator.tmpl” . }}

Funcionará correctamente ya que ese template lo tengo en la raíz del directorio. Ahora imaginemos que tenemos un proyecto mayor en el que tenemos, por poner un ejemplo, 50 endpoints. Por muy buen nombre que le definamos al template a medida que nuestro proyecto escale esto llegara a ser muy engorroso por la cantidad de endpoints apelmazados en un mismo directorio ya que aún no está implementada la opción del uso de subdirectorios.

BIBLIOGRAFÍA

1. **DesarrolloWeb.com**. Crear un proyecto Laravel con Composer. [Online]
https://desarrolloweb.com/articulos/crear-proyecto-laravel-composer.html

2. **DesarrolloWeb.com**. Crear un proyecto con Laravel Sail. [Online]
https://desarrolloweb.com/articulos/laravel-sail

3. **Aws.amazon.com**. ¿Qué es un API? [Online]
https://aws.amazon.com/es/what-is/api/

4. **HubSpot**. ¿Qué es un API Gateway? [Online]
https://blog.hubspot.es/website/que-es-api-gateway

5. **Laravel**. ¿Qué es Laravel? [Online]
https://laravel.com/docs/10.x

6. **Laravel**. The PHP Framework for Web Artisans. [Online]
https://laravel.com/

7. **Laravel**. Laravel Passport. [Online]
https://laravel.com/docs/10.x/passport

8. **GitHub**. League OAuth2. [Online]
https://github.com/thephpleague/oauth2-server

9. **PHP.net**. PHP. [Online]
https://www.php.net/manual/es/intro-whatis.php

10. **Wikipedia**. PHPUnit. [Online]
https://es.wikipedia.org/wiki/PHPUnit

11. **GitHub**. Mockery. [Online]
https://github.com/mockery/mockery

12. **Composer**. Composer. [Online]
https://getcomposer.org/

13. **GitHub**. GrumPHP. [Online]
https://github.com/phpro/grumphp

14. **Aws.amazon.com**. ¿Qué es Docker? [Online]
https://aws.amazon.com/es/docker/

15. **GitHub**. Karate. [Online]
https://github.com/karatelabs/karate

16. **GitLab**. Introduction to Git Workflows. [Online]
https://docs.gitlab.com/ee/topics/gitlab_flow.html

17. **Fidao, Chris**. Hexagonal Architecture. [Online]
https://fideloper.com/hexagonal-architecture

18. **CodelyTV**. Introducción Arquitectura Hexagonal – DDD. [Online]
https://www.youtube.com/watch?v=GZ9ic9QSO5U

19. **Laravel**. Eloquent Laravel. [Online]
https://laravel.com/docs/4.2/eloquent

20. **Vulcand**. Vulcand. [Online]
https://vulcand.github.io/

21. **Kong**. Kong. [Online]
https://konghq.com/products/kong-gateway

22. **Tyk**. Tyk. [Online]
https://tyk.io/

23. **KrakenD**. Running KrakenD server. [Online]
https://www.krakend.io/docs/overview/run/

24. **KrakenD**. Understanding the configuration file. [Online]
https://www.krakend.io/docs/configuration/structure/

25. **KrakenD**. Flexible configuration. [Online]
https://www.krakend.io/docs/configuration/flexible-config/

26. **KrakenD**. Hot reloading the configuration. [Online]
https://www.krakend.io/docs/developer/hot-reload/

27. **KrakenD**. Validating the configuration with check. [Online]
https://www.krakend.io/docs/configuration/check/

28. **KrakenD**. Creating API endpoints. [Online]
https://www.krakend.io/docs/endpoints/

29. **KrakenD**. Output encoding. [Online]
https://www.krakend.io/docs/endpoints/content-types/

30. **KrakenD**. Validating the requests with JSON Schema. [Online]
https://www.krakend.io/docs/endpoints/json-schema/

31. **KrakenD**. Concurrent Requests. [Online]
https://www.krakend.io/docs/endpoints/concurrent-requests/

32. **KrakenD**. Sequential Proxy. [Online]
https://www.krakend.io/docs/endpoints/sequential-proxy/

33. **KrakenD**. Declaring and connecting to backends. [Online]
https://www.krakend.io/docs/backends/

34. **KrakenD**. Returning the backend headers and errors. [Online]
https://www.krakend.io/docs/backends/detailed-errors/

35. **KrakenD**. Data manipulation. [Online]
https://www.krakend.io/docs/backends/data-manipulation/

36. **KrakenD**. The Circuit Breaker. [Online]
https://www.krakend.io/docs/backends/circuit-breaker/

37. **KrakenD**. Modify requests and responses with Martian. [Online]
https://www.krakend.io/docs/backends/martian/

38. **KrakenD**. JSON Web Token Validation. [Online]
https://www.krakend.io/docs/authorization/jwt-validation/

39. **KrakenD**. OAuth 2.0 Client Credentials (2-legged flow). [Online]
https://www.krakend.io/docs/authorization/client-credentials/

40. **KrakenD**. Load balancing [Online]
https://www.krakend.io/docs/throttling/load-balancing/

41. **KrakenD**. Timeouts. [Online]
https://www.krakend.io/docs/throttling/timeouts/

42. **KrakenD**. Exporting metrics and events to InfluxDB. [Online]
https://www.krakend.io/docs/telemetry/influxdb/

43. **KrakenD**. Preconfigured Grafana dashboard. [Online]
https://www.krakend.io/docs/telemetry/grafana/