Arquitectura de Software 75.73

Informe del Trabajo Práctico N°1

Integrantes

Albornoz, Rodolfo - 107975 - ralbornozt@fi.uba.ar
Almendro, Gonzalo - 80698 - galmendro@fi.uba.ar
Lamanna, Tobias - 104126 - tlamanna@fi.uba.ar
Ramirez Scarfiello, Nicolas - 102090 - nramirez@fi.uba.ar

Análisis de la arquitectura original

Componentes presentes en el sistema

El sistema en su versión base se trata de una solución monolítica, contenerizada que expone una API REST mediante un servidor Node.js al que se accede a través de un reverse proxy implementado con Nginx. El sistema está instrumentado con herramientas básicas de monitoreo para escenarios de pruebas de carga utilizando Artillery, y utiliza almacenamiento en archivos .json.

Se identifican entonces los siguientes componentes, todos en containers de docker:

- **api**: servicio principal desarrollado utilizando Express. Expone endpoints en el puerto 3000 para operaciones sobre cuentas, tasas de cambio y la ejecución de cambios de monedas. Utiliza archivos .json para persistir cuentas, tasas y logs.

- **nginx**: actúa como reverse proxy, redirigiendo tráfico HTTP desde el puerto 5555 hacia el backend en el puerto 3000.

Otras herramientas utilizadas dentro del proyecto como Docker, Artillery y demás, no forman parte de los componentes ya que no son parte de la funcionalidad principal de la aplicación. Sin embargo, podrían incluirse dentro de las capas de la aplicación:

- **graphite**: encargado de recibir métricas desde statsD (por UDP puerto 8125) y exponerlas por HTTP.

- **grafana**: visualizador de métricas. Se puede acceder a través de un navegador web a su interfaz para observar las métricas. Se conecta por HTTP a graphite utilizándolo como Data Source.

- **cadvisor**: herramienta de monitoreo de contenedores que recolecta métricas de uso de recursos (CPU, RAM) y las envía por UDP al puerto 8125 correspondiente a graphite.

Capas

Dentro del proyecto, las partes están bien separadas, y cada parte tiene su propia responsabilidad y no está cargada de alguna responsabilidad que no deba tener, realizando una tarea en específico. Estas partes pueden separarse en 5:

- Capa de presentación: Recibe las solicitudes HTTP. Su componente es el Nginx que es el load balancer
- Capa de aplicación: Procesa las solicitudes y aplica la lógica del sistema. Sus componentes son Node.js y Express
- Capa de datos: Guardar los datos, permite recuperarlos y también los persiste. Su componente son los archivos JSON de los cuales se obtiene información y hacia los cuales se guarda nueva información.
- Capa de monitoreo: Mide el rendimiento, permite ver y graficar métricas. Sus componentes son Artillery (Pruebas de carga), Grafana (Visualizar métricas) y StatsD, cAdvisor y Graphite (Obtener métricas)

- Capa de infraestructura: Orquesta todos los servicios y conecta todas las partes de la aplicación. Sus componentes son Docker (Contenerizar servicios) y Docker compose (Orquestar servicios)

Servicios

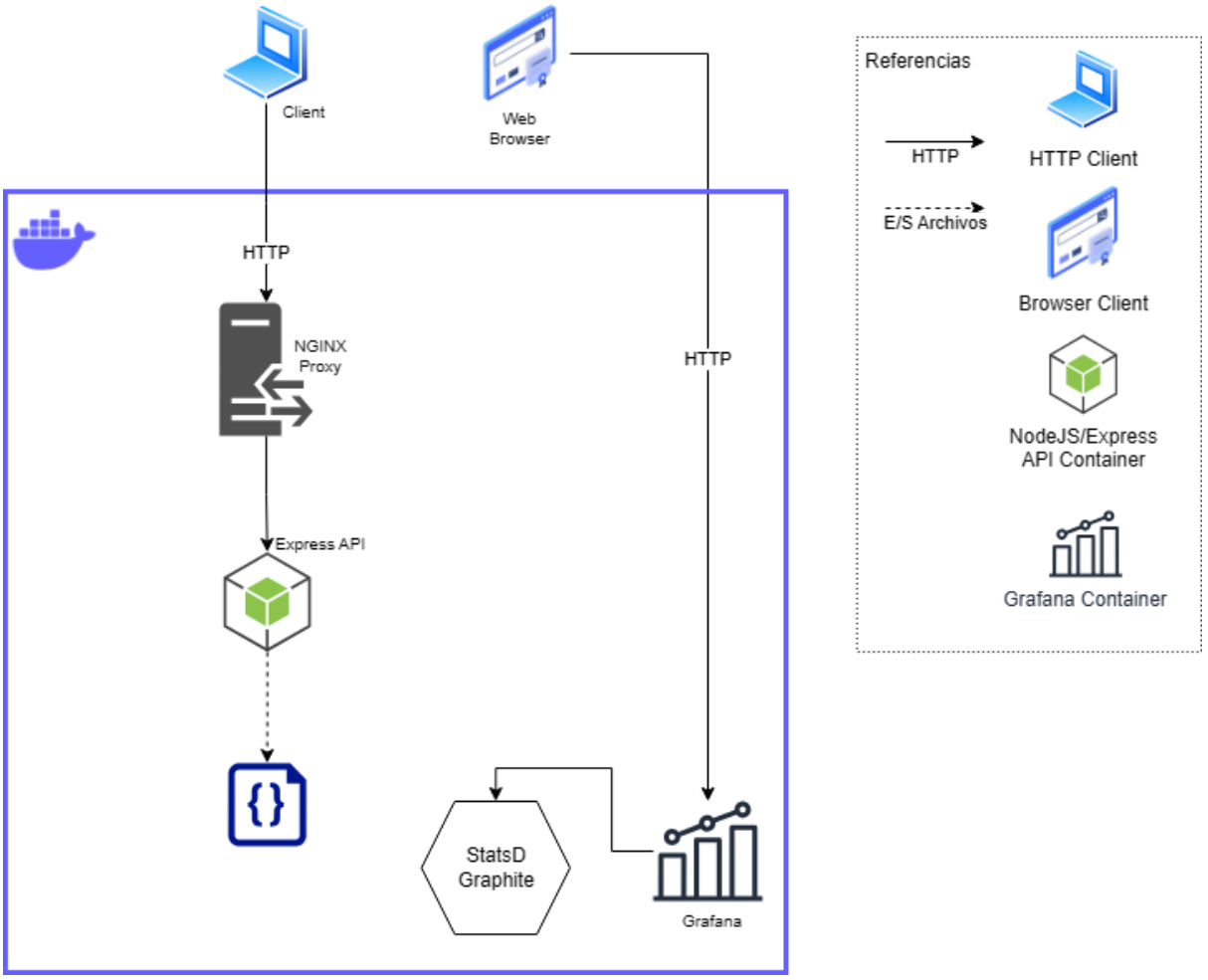
Tasas de cambio (Rates): Tiene la responsabilidad de obtener y modificar las tasas de cambio. Cuando se altera una tasa de cambio también calcula la recíproca. Por cómo se desarrolló este endpoint, no hay una ganancia para arVault por la conversión (O sea no se cobra comisión por la conversión) También se sugiere que se pueden crear niveles de usuarios, y darle una mejor tasa a un usuario de mayor nivel. Esto abre la idea de escalabilidad para el sistema.

Cuentas (Accounts): Obtiene las cuentas y modifica sus saldos. Por cómo está diseñado el método PUT, el endpoint de cuentas tiene más responsabilidad de la que debería, no debería poder modificar su saldo. Para eso, se puede crear otro servicio que lo haga.

Cambio (Exchange): Registra las operaciones de cambios. Las validaciones de saldo suficiente están hechas para la cuenta propia pero no para la cuenta del cliente.

Logs: Devuelve un log de operaciones.

Diagrama Components & Connectors



Atributos de Calidad claves para el servicio

El servicio de exchange tiene requisitos funcionales simples pero críticos, lo cual pone en evidencia ciertos atributos de calidad prioritarios:

- **Fiabilidad**: Al ser una aplicación de manejo e intercambio de dinero, un error en alguna operación puede significar pérdida de dinero, lo que implica demandas de usuarios, problemas legales, etc. Es importante que en el sistema, las operaciones sean correctas, que no haya margen de error, que no se realicen operaciones inválidas, que si una operación no puede realizarse, no deje inconsistencia en los datos de la aplicación, etc.
- **Escalabilidad**: Si bien en lo inmediato no es un problema, si la aplicación busca incrementar su base de usuarios orgánicamente se dará un incremento en la cantidad de operaciones, por eso es importante que el sistema sea capaz de escalar para procesar correctamente el aumento de tráfico.
- **Disponibilidad**: Dada la naturaleza de volatilidad del mercado financiero es de interés que el servicio se encuentre siempre operativo y sin interrupciones no programadas; dado que la falta de disponibilidad del servicio en un momento clave puede representar pérdidas de dinero para los usuarios.

- **Performance**: Es clave que las operaciones puedan realizarse en un tiempo razonable, y que por ejemplo, un intercambio de monedas no tarde un día. Siguiendo la línea de la escalabilidad, también es clave que no se deteriore la performance con un gran volumen de operaciones.

Críticas de las decisiones de diseño originales

Uso de Nginx como Reverse Proxy: La decisión de colocar Nginx como un reverse proxy es una práctica común en arquitecturas web modernas. El impacto positivo principal es que aísla y protege el servidor (app) de la exposición directa a internet, pudiendo manejar tareas como potencialmente el balanceo de carga en futuras iteraciones. Esto mejora la Seguridad y la Modificabilidad del sistema. Sin embargo, introduce una capa adicional de red, lo que genera una latencia mínima adicional en cada solicitud. Aunque este problema identificado existe, el impacto en el rendimiento suele ser insignificante comparado con los beneficios de seguridad y flexibilidad que aporta, considerándose generalmente un trade off aceptable.

Almacenamiento de Datos en Archivos .json: La elección de archivos .json para persistir datos críticos (tasas de cambio, saldos de cuentas o límites) representa una decisión particular que compromete algunos de los QA claves del servicio. El impacto positivo es la facilidad y rapidez de implementación inicial, especialmente en un entorno Node.js donde el manejo de JSON es nativo; no obstante, los problemas identificados son severos y numerosos. Este enfoque no es escalable; el acceso secuencial a archivos se convierte rápidamente en un cuello de botella bajo carga concurrente empeorando la performance. Es poco flexible para realizar consultas complejas o modificar el esquema de datos. Más críticamente, es altamente propenso a la corrupción de datos y fallos de procesamiento debido a condiciones de carrera.. Esto compromete gravemente la fiabilidad, la escalabilidad, y la performance siendo inadecuado para una aplicación financiera.

Ausencia de Persistencia Robusta y Consistente: Relacionado directamente con el uso de archivos .json (punto anterior), el sistema carece de un mecanismo de persistencia que garantice la atomicidad, consistencia, aislamiento y durabilidad (ACID) de las operaciones, algo que es de especial interés en operaciones financieras. Aunque reduce la complejidad inicial al evitar la configuración y gestión de una base de datos transaccional, el problema identificado es crítico: impide asegurar la consistencia y disponibilidad de los datos ante fallos del sistema o acceso concurrente, tampoco permite realizar rollbacks automáticos de operaciones de escritura si se desean realizar sobre múltiples entidades y alguna de ellas falla. Una caída durante una escritura puede dejar los datos corruptos, y operaciones simultáneas pueden llevar a estados inconsistentes.. Esta falta de fiabilidad y garantías de transaccionalidad es inaceptable para una Fintech.

Métricas de cAdvisor vía UDP a StatsD: La inclusión de cAdvisor para monitorear métricas a nivel de contenedor (CPU, memoria, red) es positiva para la Observabilidad de la infraestructura. Permite identificar si los contenedores cuentan con la cantidad adecuada de recursos. El problema identificado radica en la forma de integración: cAdvisor puede generar un volumen muy alto de métricas y enviarlas vía UDP a la misma instancia StatsD que procesa las métricas personalizadas de la aplicación, lo que puede saturar la conexión. Dado que UDP no garantiza la entrega, si StatsD está ocupado procesando la avalancha de métricas de cAdvisor, podría descartar paquetes UDP que contienen métricas de negocio más críticas. Esto impacta negativamente la fiabilidad de la recolección de métricas clave y puede generar carga innecesaria en la red, afectando el rendimiento del sistema de

monitoreo.

Ausencia Inicial de Métricas de Negocio: Originalmente, la aplicación no emite métricas específicas que reflejaran su uso real desde una perspectiva de negocio (ej. volumen operado por moneda). El único impacto positivo es el ahorro de tiempo de desarrollo inicial, pero su contraparte es la incapacidad de medir el rendimiento del negocio, comprender el comportamiento del usuario (¿qué monedas son más populares?), gestionar riesgos, o tomar decisiones estratégicas informadas. Sin estas métricas, la empresa opera "a ciegas" respecto a su actividad principal. La falta de Observabilidad a nivel de negocio dificulta la gestión operativa y la capacidad de demostrar valor a los inversores.

En conjunto, se observa un cuadro de un sistema construido priorizando la velocidad de desarrollo inicial sobre los atributos de calidad (tanto claves como aquellos que no lo son) acumulando una deuda técnica significativa que ahora se manifiesta en problemas operativos.

Algunas críticas que pueden realizarse adicionales al sistema respecto a otros atributos de calidad puntuales son:

- **Testeabilidad**: El sistema se encuentra relativamente modularizado, lo que permite el desarrollo de suites de pruebas, pero no se encuentran actualmente en el código lo que indica que no se desarrolló teniendo esto en mente.
- **Observabilidad**: El sistema cuenta con algunas métricas básicas de infraestructura proveídas por cAdvisor, lo que no es suficiente pero la inclusión de graphite y statsD permite que la adición de métricas no sea un proceso difícil.
- **Portabilidad**: La decisión de containerizar todos los componentes del sistema mejora su portabilidad al poder ser desplegado en múltiples sistemas operativos.
- **Interoperabilidad**: De la mano con la portabilidad el uso de docker y la exposición de puertos a través de NGINX le permite operar fácilmente con clientes externos.
- **Manejabilidad**: El sistema no parece contar con una interfaz back office, la configuración de tasas de cambio por ejemplo debe realizarse manualmente mediante la edición de archivos JSON lo que no es conveniente desde el punto de vista de la manejabilidad.
- **Modificabilidad**: La falta de pruebas unitarias dificulta la refactorización del código, que si bien no es muy extenso en caso de crecer orgánicamente como lo hizo hasta ahora se puede convertir en un problema a futuro.

Instrumentación de métricas, escenarios de carga y evaluación

Setup del stack de monitoreo

Para instrumentar el sistema y obtener métricas relevantes, se configuró un stack de observabilidad basado en:

- **StatsD (vía graphite-statsd):** Receptor de métricas tipo contador.
- **Graphite:** Motor de almacenamiento y visualización de métricas.
- **Grafana:** Dashboard de métricas unificado, conectado vía HTTP a Graphite.
- **Artillery:** Generador de carga configurado para enviar métricas directamente a StatsD

.

Descripción de los escenarios simulados con Artillery

Para la realización de las pruebas se diseñaron múltiples escenarios a ser ejecutados utilizando Artillery. Estos escenarios tienen como objetivo generar carga para los endpoints:

GET accounts, PUT accounts, GET Log, GET Rates, POST Exchange.

En el documento anexado "*Resultados grafana"* pueden observarse las mediciones obtenidos para estos escenarios para la versión V1 correspondiente al sistema base recibido (y el que se analiza a continuación), y también a las versiones V1.1 y V2 correspondientes a modificaciones posteriores que se realizaron al sistema que se detallan en la sección *Tácticas aplicadas y diseño actualizado*

Análisis de resultados

Para cada endpoint:

- GET Accounts
  - Todas las solicitudes se procesan correctamente.
  - Se observa que a medida que aumenta el tráfico también lo hace el response time alcanzando un p95 medio de 226 ms..
  - El uso de recursos del sistema alcanza el 30% del CPU y menos del 1% de la memoria.
- GET Rates
  - Todas las solicitudes se procesan correctamente.
  - El aumento del tráfico no representa un aumento en los tiempos de respuesta, hay algunos picos de hasta 600 ms pero la media del p95 es de 57 ms.
  - El uso del CPU se mantiene entre un 2% y un 8%, y el uso de memoria se mantiene por debajo del 1%.
- GET log
  - Se observan errores que superan el 30% de error rate en el tramo final de la prueba al aumentar el tráfico.
  - El response time se mantiene planchado hasta el aumento de tráfico que lo eleva hasta alcanzar valores de hasta 10 segundos.
  - El uso de CPU alcanza el 50% en el pico final de la prueba.
- PUT Rates
  - Todas las solicitudes se procesan correctamente.
  - La media del p95 es de 25 ms a lo largo de la prueba.
  - El uso del CPU no supera el 6%
- POST Exchange
  - La tasa de errores empieza a aumentar en la última etapa de la prueba, llegando a un punto crítico de 100% de errores. A lo largo de toda la prueba la tasa de error es alta encontrándose por encima del 50%.
  - Los tiempos de respuesta son altos, la media del p95 es mayor a 500 ms, la media del p99 alcanza los 976 ms. Todas las mediciones del tiempo de respuesta aumentan en el último tramo de la prueba casi triplicando su valor.

- El uso de CPU presenta una media del 25% alcanzando picos superiores al 60%.

Se observa mediante las pruebas que la escalabilidad, fiabilidad, y performance se encuentran comprometidas en el sistema. Los incrementos de response time y error rate al incrementar el tráfico indican problemas relacionados a las decisiones de diseño previamente discutidas.

Tácticas aplicadas y diseño actualizado de la arquitectura

Tras el análisis inicial de la arquitectura y la identificación de problemas críticos que afectan atributos de calidad clave como la Fiabilidad, Seguridad, Escalabilidad y Performance, se propusieron e implementaron una serie de tácticas arquitectónicas para mitigar dichos problemas y mejorar la robustez general del servicio. Esta sección detalla las tácticas aplicadas, describe la arquitectura resultante, y analiza su impacto en los atributos de calidad.

Tácticas Aplicadas

1. Mejora de la Seguridad y Fiabilidad mediante Validación de Entradas

El sistema no tiene validaciones, lo que afecta a la seguridad del sistema pudiendo enviarse datos erróneos, realizar operaciones inválidas o enviar muchas peticiones erróneas que tiren abajo el sistema. Potencialmente también se podría ver afectada la disponibilidad ya que si no hay control de los datos que se envían en las requests, y se envían muchas para atacar al sistema, el sistema caerá debido a un ataque DoS (Denial-of-Service).

Se agregaron validaciones a los campos enviados a los endpoints de PUT Rate, POST Exchange y PUT Balance. Se validó que se enviaran al request los campos obligatorios, que tuvieran los valores, tipos y/o formatos correctos. Además, se muestra el mensaje de error correspondiente que señala el campo que está incorrecto al intentar enviar el request. Los mensajes de éxito o de error en la request también se mejoraron y se hicieron más descriptivos.

Todos estos aspectos mejoran la fiabilidad del sistema, haciendo que sea más sólido al tener un sistema de validación de peticiones y no permitiendo cualquier petición. Sobre todo mejora la seguridad ya que evita que se pasen datos incorrectos o erróneos, y también permite defenderse de ataques maliciosos de envío de muchas peticiones rechazadas si estas no tienen los valores correctos en los campos.

Nuestra hipótesis fue que también disminuiría la performance ya que más validaciones requieren más tiempo de procesamiento, pero en la versión final modificada (v1.1) la performance no se vio afectada negativamente.

2. Optimización del Rendimiento en Búsquedas Internas (Paso Intermedio)

Al tener que realizar búsquedas de cuentas por su id (Por ejemplo en el endpoint POST Transfer), se hacía la búsqueda en un arreglo, lo que hacía que su complejidad sea O(n). Ahora, al hacer la

búsqueda, se hace en el arreglo de cuentas convertido en una estructura Map, y entonces la complejidad se convierte en O(1).
Nuestra hipótesis era que esto ayuda a la performance del sistema. Aunque la mejora en Performance no fue drástica con pocas cuentas, esta optimización sentó una base para mejorar la Escalabilidad. Sin embargo, esta táctica fue superada por la adopción de Redis posteriormente.

3. Mejora de la Persistencia de Datos con Redis (Táctica Principal - V2)

El uso de archivos JSON para almacenar estado crítico presentaba graves deficiencias de Performance, Escalabilidad, Seguridad y Fiabilidad.
Por lo que se adoptó Redis como almacén de datos externo en memoria (Key-Value Store). Toda la lógica de acceso y modificación de datos fue refactorizada para interactuar con Redis mediante un cliente específico (ver exchange-v2-final.js y state-v2.js), reemplazando completamente el acceso a archivos JSON para la versión V2 de la API.
Esta táctica genera mejoras sustanciales:

- Performance/Latencia: Acceso a datos en memoria mucho más rápido que I/O de disco.
- Escalabilidad: Redis maneja concurrencia de manera más eficiente que los bloqueos de archivos.
- Fiabilidad: Se reduce el riesgo de corrupción por escrituras concurrentes o fallos. Redis ofrece mecanismos de persistencia opcionales (no explorados aquí) y operaciones atómicas.
- Seguridad: Los datos ya no residen en archivos planos fácilmente accesibles.
- Mantenibilidad: Centraliza la gestión de datos.

4. Mejora de la Observabilidad con Métricas de Negocio

En el apartado de observabilidad y fiabilidad se observa la carencia de visibilidad sobre el uso real del servicio.

Se instrumentó la aplicación para generar y enviar métricas específicas de negocio (volumen total y neto por moneda) vía UDP a una instancia dedicada de StatsD/Graphite (graphite-business); también se adicionó la métrica de response time medida en el servicio de exchange.

La separación del endpoint de métricas fue necesaria debido al volumen de datos enviado por cAdvisor a la instancia de graphite existente. Al ser UDP un protocolo que no garantiza entrega se encontró un comportamiento no deseado, las métricas de negocio compiten contra las enviadas por cAdvisor lo que generaba una gran pérdida de paquetes. Para solventar este problema se optó por la duplicación de la instancia de graphite desdoblandola en graphite-infra y graphite-business, cada una recibiendo paquetes por su propio puerto.

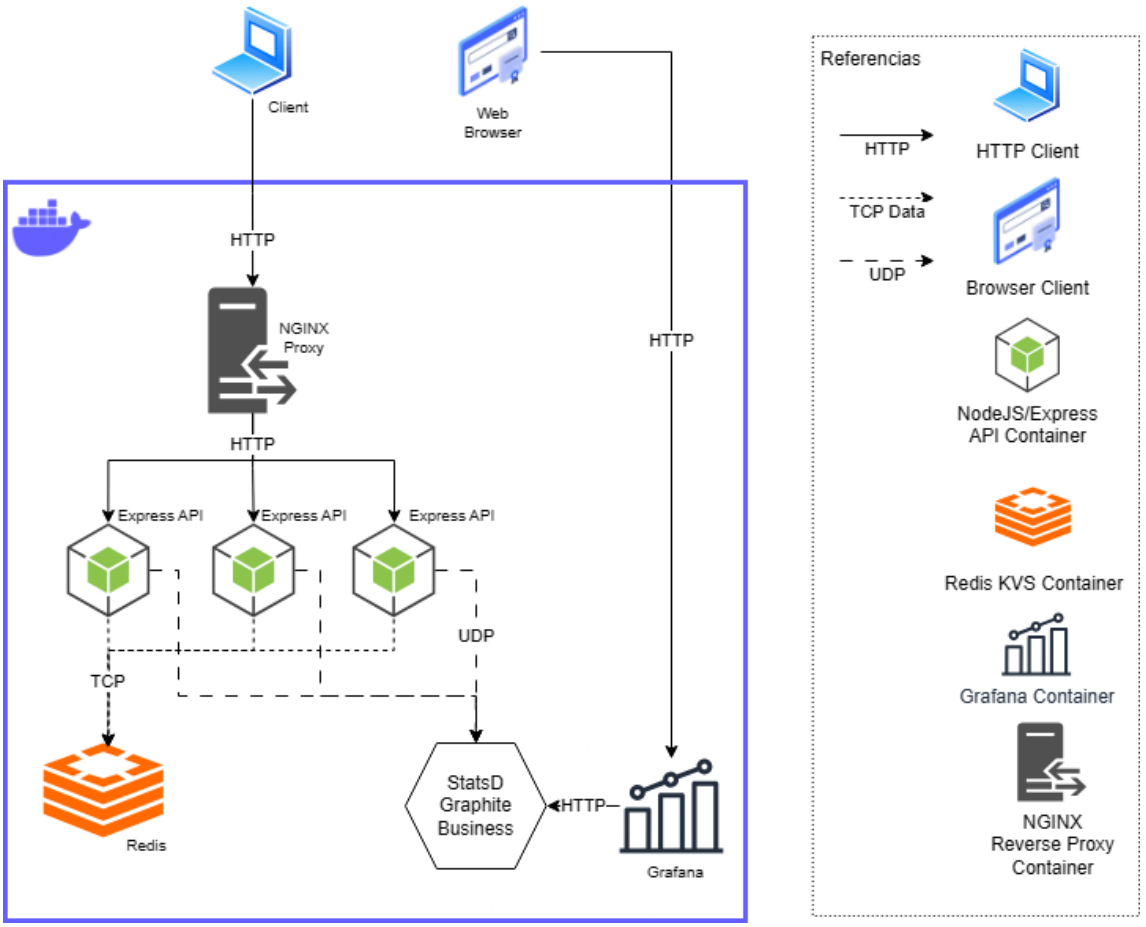
Este cambio mejoró significativamente la Observabilidad del negocio, permitiendo análisis de comportamiento y la implementación de una nueva instancia de graphite dedicada para métricas del negocio mejoró la fiabilidad de las métricas.

5. Mejora de la Escalabilidad y Disponibilidad con Múltiples Instancias (Balanceo de Carga)

Una única instancia de la aplicación limita la capacidad y es un punto único de fallo, por esto se optó en la arquitectura final utilizar Nginx no solo como reverse proxy sino también como balanceador de carga, distribuyendo el tráfico HTTP entrante entre múltiples instancias (tres en el diagrama y en las pruebas) del contenedor de la API Node.js/Express.

Esta táctica de replicación mejora directamente la Escalabilidad Horizontal (se puede aumentar la capacidad añadiendo más instancias) y la Disponibilidad (si una instancia falla, Nginx puede redirigir el tráfico a las instancias restantes).

Diseño Actualizado de la Arquitectura (v2)



La arquitectura resultante, reflejada en el diagrama de Componentes y Conectores final y el código v2, representa una evolución significativa:

- Cliente Externo: Interactúa con el sistema vía HTTP.
- Nginx (Reverse Proxy y Load Balancer): Punto de entrada único. Gestiona conexiones, y distribuye la carga vía HTTP a las instancias de la API.
- Instancias de API Node.js/Express (Replicadas): Contenedores idénticos (v2) que ejecutan la lógica de negocio interactuando con Redis. El número de instancias puede ajustarse para escalar.

- Redis: Almacén de datos en memoria centralizado. Las instancias de la API se conectan a Redis (vía TCP) para leer/escribir datos (cuentas, tasas, tiers, logs).
- Stack de Monitoreo:
    - StatsD/Graphite (Business): Instancia dedicada (graphite-business:8126) que recibe métricas de negocio (volumen, neto, response time del lado del servidor) vía UDP desde las instancias de la API.
    - Grafana: Consulta a Graphite (vía HTTP API) para visualizar los dashboards.

Este diseño actualizado aborda las principales deficiencias del sistema original. La introducción de Redis y el balanceo de carga con Nginx son cambios estructurales clave que mejoran drásticamente la Performance, Escalabilidad, Fiabilidad y Disponibilidad. Las validaciones, refactorización de endpoints y métricas de negocio refuerzan la Seguridad, Fiabilidad y Observabilidad, resultando en una arquitectura significativamente más robusta.

Evidencia de mejoras

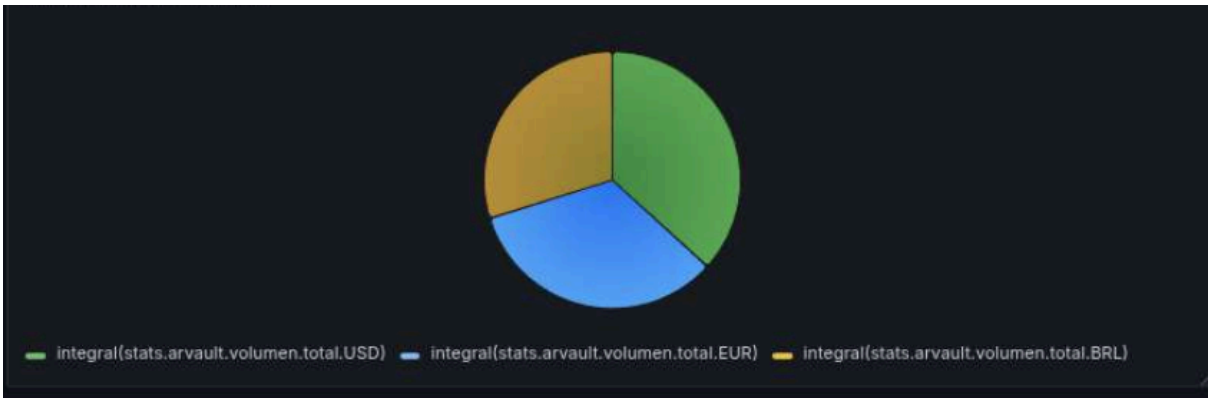
Los gráficos obtenidos a partir de las pruebas realizadas sobre las versiones V1.1 (con validaciones y mejora de algoritmo de búsqueda de cuentas) y V2 (con Redis y múltiples instancias de la API) pueden visualizarse en el archivo “Resultados grafana” anexado.

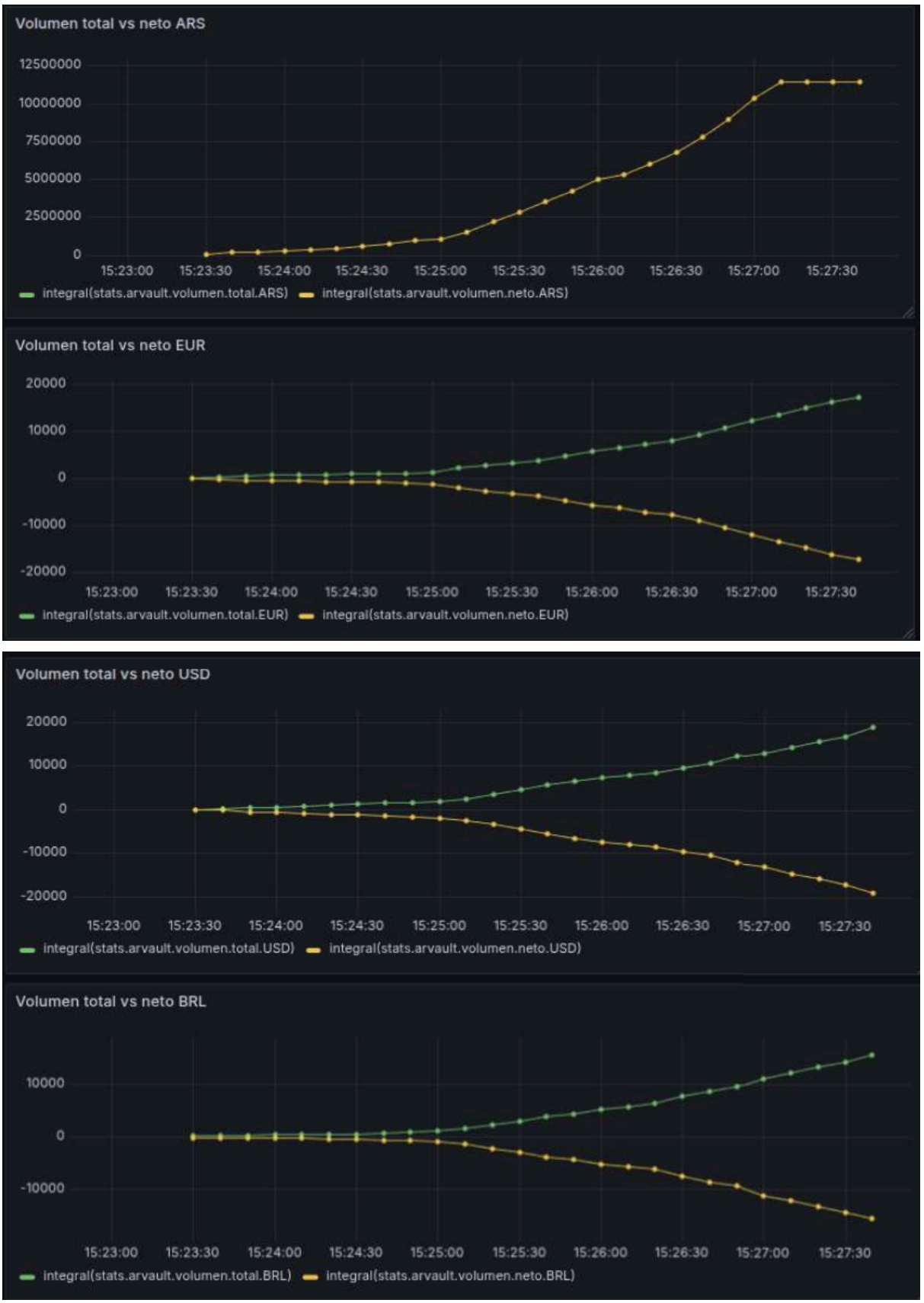
En términos generales se observa que la versión V2 presenta una mejora en el error rate, reduciéndose a 0 en las pruebas sobre el endpoint POST Exchange que previamente presentaba un error rate mayor al 50%. También los tiempos de respuesta se vieron mejorados, y el consumo de recursos no representó un incremento sustancial.

Consideraciones sobre métricas solicitadas por el fundador

Se incorporaron las métricas de negocio correspondientes a volumen total y neto operado por moneda

Volumen total USD vs EUR vs BRL




Volumen total vs neto ARS
12500000
10000000
7500000
5000000
2500000
0
15:23:00
15:23:30
15:24:00
15:24:30
15:25:00
15:25:30
15:26:00
15:26:30
15:27:00
15:27:30
integral(stats.arvault.volumen.total.ARS)
integral(stats.arvault.volumen.neto.ARS)
Volumen total vs neto EUR
20000
10000
0
-10000
-20000
15:23:00
15:23:30
15:24:00
15:24:30
15:25:00
15:25:30
15:26:00
15:26:30
15:27:00
15:27:30
integral(stats.arvault.volumen.total.EUR)
integral(stats.arvault.volumen.neto.EUR)
Volumen total vs neto USD
20000
10000
0
-10000
-20000
15:23:00
15:23:30
15:24:00
15:24:30
15:25:00
15:25:30
15:26:00
15:26:30
15:27:00
15:27:30
integral(stats.arvault.volumen.total.USD)
integral(stats.arvault.volumen.neto.USD)
Volumen total vs neto BRL
10000
0
-10000
15:23:00
15:23:30
15:24:00
15:24:30
15:25:00
15:25:30
15:26:00
15:26:30
15:27:00
15:27:30
integral(stats.arvault.volumen.total.BRL)
integral(stats.arvault.volumen.neto.BRL)

Anexos

Resultados grafana