FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1

Grupo

[75.73/TB034] Arquitectura del Software
Segundo cuatrimestre de 2025

Alumno	Padrón	Email
CASTRO MARTINEZ, Jose Ignacio	106957	jcastrom@fi.uba.ar
DEALBERA, Pablo Andres	106858	pdealbera@fi.uba.ar
FIGUEROA RODRIGUEZ, Andrea	110450	afigueroa@fi.uba.ar
RICALDI REBATA, Brayan Alexander	103344	bricaldi@fi.uba.ar

Índice

1. Introducción **3**
1.1. Contexto (startup arVault). 3
1.2. Objetivos del TP. 3
1.3. Alcance del análisis. 3

2. Atributos de calidad (QA) identificados **3**
2.1. Disponibilidad 3
2.2. Escalabilidad (Elasticidad) 3
2.3. Performance 3
2.4. Visibilidad 4
2.5. Seguridad 4

3. Arquitectura base **4**
3.1. Análisis de la influencia de decisiones de diseño en los QA's 4
3.1.1. Disponibilidad 4
3.1.2. Escalabilidad (Elasticidad) 6
3.1.3. Performance 6
3.1.4. Visibilidad: 6
3.1.5. Seguridad 7
3.1.6. Testabilidad 7
3.1.7. Portabilidad 7
3.1.8. Interoperabilidad 7
3.1.9. Usabilidad 7
3.1.10. Manejabilidad 7
3.1.11. Confiabilidad 7
3.1.12. Simplicidad 7
3.1.13. Modificabilidad 7
3.2. Diagrama C&C inicial. 7
3.3. Crítica a arquitectura base. 8

4. Metodología de pruebas **8**
4.1. Herramientas usadas (Artillery, Grafana, etc.). 8
4.2. Escenarios de carga diseñados. 8
4.3. Métricas recolectadas. 8

5. Resultados – Caso base **8**
5.1. Screenshots de dashboards. 8
5.2. Interpretación de resultados. 8

6. Propuestas de mejora **8**
6.1. Tácticas aplicadas. 8
6.1.1. Implementación de Valkey para persistencia 8
6.1.2. Configuración en Docker Compose 8
6.1.3. Beneficios para escalabilidad horizontal 8
6.2. Justificación de por qué se eligieron. 9
6.3. Diagramas C&C modificados. 9

7. Resultados – Casos mejorados **10**
7.1. Screenshots comparativos. 10
7.2. Análisis de impacto sobre QA. 10
7.3. Trade-offs detectados. 10

8. Conclusiones **10**
8.1. Resumen de hallazgos. 10
8.2. Beneficios y limitaciones de las mejoras. 10
8.3. Futuro (ej: pasar a DB externa en TP2). 10

9. Anexos **10**
9.1. Configuraciones de escenarios de Artillery. 10
9.2. Configs modificadas de Nginx, Docker Compose, etc. 10

1. Introducción

1.1. Contexto (startup arVault).

1.2. Objetivos del TP.

1.3. Alcance del análisis.

2. Atributos de calidad (QA) identificados

2.1. Disponibilidad

Al ser un servicio de exchange de monedas, asumimos que es un servicio que se utiliza durante todos los dias habiles de la semana en horario cambiario. Por lo tanto, es importante que el servicio se encuentre disponible durante esos horarios para no perder clientes.

Además, dado el contexto en el que queremos recuperar la confianza de los usuarios y remontar la reputación, el sistema debe ser altamente accesible para los usuarios y permitir realizar correctamente sus operaciones respetando tiempos razonables de respuesta.

2.2. Escalabilidad (Elasticidad)

La escalabilidad, y en particular la elasticidad, constituyen un atributo de calidad crítico para el servicio de intercambios de arVault. Esto se debe a que la infraestructura del sistema debe ser capaz de adaptarse dinámicamente a variaciones en la demanda de usuarios.

En el contexto del negocio, es esperable la aparición de picos significativos de demanda en momentos específicos (por ejemplo, en la apertura y cierre del horario cambiario), así como también períodos de baja o nula actividad. A ello se suma que, dado que el servicio busca captar rápidamente un gran volumen de nuevos usuarios, especialmente tras campañas de promoción destinadas a revertir percepciones negativas de experiencias pasadas, existe el riesgo de enfrentar aumentos inesperados de tráfico.

Si el sistema careciera de elasticidad, estos picos de operaciones de cambio de moneda podrían derivar en saturación de recursos, lo que a su vez ocasionaría demoras, rechazos de transacciones o caídas del servicio. Dichos incidentes afectarían de manera directa la reputación de la empresa, un aspecto considerado prioritario en función de los objetivos actuales y de las expectativas de los stakeholders.

2.3. Performance

El atributo de calidad Performance, y en particular el User-Perceived Performance, adquiere relevancia crítica en el servicio de intercambio de monedas de arVault, para sustentar esta afirmación nos basamos en el siguiente análisis del contexto y antecedentes brindados:

Tras el lanzamiento de la funcionalidad, se registraron reclamos de usuarios relacionados con demoras y fallas en la ejecución de operaciones de cambio, lo que ha derivado en reseñas negativas y pérdida de confianza en la plataforma. En un contexto donde la empresa necesita con urgencia atraer nuevas rondas de inversión, estas deficiencias de rendimiento representan un riesgo directo, ya que los potenciales inversores han condicionado su apoyo a la realización de mejoras en la calidad del servicio.

En una aplicación financiera, la percepción de agilidad y confiabilidad en la respuesta del sistema es esencial: tiempos de espera excesivos o transacciones fallidas afectan la experiencia de los usuarios y minan la credibilidad de la plataforma. Aunque el diferencial de arVault reside en ofrecer tasas de cambio más convenientes que la competencia, dicho valor se ve neutralizado si el servicio de intercambio no responde con la rapidez y estabilidad que los clientes esperan.

Por ello, la mejora del User-Perceived Performance se presenta como un paso imprescindible no solo para recuperar la confianza de los usuarios actuales, sino también para restaurar la reputación

de la empresa ante el mercado y viabilizar la captación de nuevos inversores, garantizando así la continuidad y evolución del negocio.

2.4. Visibilidad

El valor de este atributo de calidad es más indirecto pero estratégico pues permite entender el comportamiento real del sistema, identificar cuellos de botella de performance, localizar errores en operaciones de cambio y detectar patrones de saturación que anticipen problemas de disponibilidad o escalabilidad. Es decir, la visibilidad no impacta de forma inmediata en la experiencia del usuario, pero habilita a los arquitectos y al equipo técnico a diagnosticar, mejorar y sostener los otros atributos de calidad prioritarios.

2.5. Seguridad

Es fundamental que el sistema sea seguro para evitar posibles ataques que puedan comprometer la integridad del sistema, la privacidad de los datos de los clientes o perdida/robo de dinero. Incluso pensando que debemos tambien tener en cuenta marcos regulatorios sobre el manejo de datos personales y financieros. Tambien entendemos del enunciado que es importante la reputacion del sistema, y esto podria verse muy dañado en caso de que haya una brecha de seguridad.

3. Arquitectura base

3.1. Análisis de la influencia de decisiones de diseño en los QA's

3.1.1. Disponibilidad

1. **Puntos únicos de falla (únicas instancias de servicios) y su impacto**

   Analizando las decisiones de diseño tomadas por el desarollador, particularmente con un análisis de la infraestructura y diseño del despliegue del sistema, nos percatamos que los puntos que se mencionan a continuación impactan negativamente en la Disponibilidad del sistema pues modelan una arquitectura con alta dependencia de componentes individuales, sin mecanismos de redundancia, con múltiples puntos únicos de falla y carente de mecanismos de recuperación automática, lo cual implica que la falla de un solo servicio (API, Nginx, almacenamiento local) ocasionaría la indisponibilidad total del sistema.

   a) **Backend (API)**

   Se despliega una única instancia del servicio de API (según la configuración en docker-compose). De esta forma, la ausencia de réplicas y de un mecanismo efectivo de balanceo de carga (porque no hay múltiples nodos entre los cuales se balancee la carga) la caída de dicha instancia del backend causaría que el sistema completo deje de responder a solicitudes.

   b) **Nginx (reverse proxy)**

   Para este servicio también existe una sola instancia configurada como punto de entrada y aunque se define un bloque 'upstream', este solo redirige a una única API backend.

   Por esto la arquitectura termina teniendo dos puntos críticos: tanto el proxy (nginx) como el backend, la indisponibilidad de cualquiera de ellos impacta directamente en la experiencia del usuario final.

   c) **Persistencia de datos**

   Actualmente la aplicación utiliza archivos JSON locales para la persistencia, este enfoque presenta múltiples limitaciones: falta de replicación, ausencia de mecanismos de recuperación ante fallas, y dependencia del almacenamiento local del contenedor/host. Una pérdida de datos o la caída del servicio implican tiempos de recuperación prolongados, degradando así directamente la disponibilidad.

2. **Arquitectura monolítica y su impacto**

   Al analizar la estructura lógica del sistema, se observa que este responde a un patrón **monolítico**, en el cual toda la lógica de negocio, el manejo de estado y la persistencia de datos se concentran en un solo bloque sin separación clara de responsabilidades ni interfaces desacopladas. Este diseño acarrea consecuencias directas sobre la **Disponibilidad**, entre las que se destacan:

   a) **Arquitectura unificada** Toda la lógica de negocio (gestión de cuentas, tasas, transacciones) se encuentra contenida en un único módulo. La caída de cualquier componente interno afecta al sistema en su totalidad, ya que no existen mecanismos de aislamiento de fallos ni tolerancia a errores.

   b) **Alto acoplamiento entre módulos** Los componentes del sistema tienen dependencias directas y requieren inicializaciones en un orden específico. Esto implica que la indisponibilidad de un módulo interno impide el correcto funcionamiento del resto, amplificando los riesgos de interrupción total.

   c) **Escalabilidad y resiliencia limitadas** Al no existir modularidad ni servicios independientes, no es posible escalar ni recuperar selectivamente partes del sistema. Cualquier estrategia de replicación debe aplicarse al monolito completo, lo cual incrementa la complejidad operativa y reduce la capacidad de respuesta frente a fallos.

   En síntesis, la naturaleza monolítica del sistema no solo **explica** la existencia de múltiples puntos únicos de falla en la infraestructura actual, sino que también **agrava su impacto**: ante un error en un módulo o en la persistencia de datos, la indisponibilidad afecta a toda la aplicación. Esto limita severamente la capacidad de mantener una operación continua y dificulta la incorporación de mecanismos de alta disponibilidad o recuperación automática.

3. **Carencia de uso de transacciones y su impacto**

   Otro aspecto crítico identificado es la ausencia de un sistema de **transacciones confiables** para el manejo de operaciones financieras (por ejemplo, conversiones entre diferentes monedas). Actualmente, la persistencia de datos se basa en archivos JSON locales, sin soporte nativo para propiedades ACID.

   Esta limitación introduce riesgos importantes que afectan directamente el atributo de calidad **Disponibilidad**, principalmente se tiene un gran riesgo de **inconsistencias de datos e incremento del tiempo de recuperación**, pues, al no existir mecanismos transaccionales, fallas en medio de una operación (ej. caída del proceso, error de escritura en disco) pueden dejar el sistema en un estado inconsistente. Esto obliga a tareas manuales de verificación y corrección, aumentando el tiempo que el sistema permanece fuera de servicio o con datos inválidos. En ausencia de transacciones, las operaciones incompletas no pueden deshacerse ni repetirse de forma segura. Frente a fallos, el sistema requiere procesos de recuperación manual o la restauración de copias de seguridad, en consecuencia, se disminuye la disponibilidad percibida.

   En conclusión, la carencia de un sistema de transacciones robusto aumenta significativamente la probabilidad de inconsistencias críticas y prolonga los tiempos de recuperación ante fallas. Dado el carácter financiero de las operaciones que maneja el sistema, esta limitación constituye un factor determinante que degrada la **Disponibilidad**, al no poder garantizar continuidad operativa ni datos válidos tras un incidente.

4. **Otras decisiones de diseño con impacto indirecto en la disponibilidad**

   Existen además otras decisiones de diseño que, si bien no afectan a la **Disponibilidad** de manera directa, sí lo hacen de forma indirecta al influir en atributos de calidad relacionados:

   - **Monitoreo y métricas**: la toma de métricas y la incorporación de herramientas de observabilidad impactan directamente en el atributo de calidad **Visibilidad**. A su vez,

una mayor visibilidad facilita la detección temprana de fallas y acelera los procesos de recuperación, contribuyendo indirectamente a la disponibilidad del sistema.

- **Escalabilidad**: las limitaciones en la capacidad del sistema para crecer o adaptarse a aumentos de carga afectan principalmente al atributo de calidad **Escalabilidad**. Sin embargo, la incapacidad de manejar picos de demanda también puede llevar a interrupciones o caídas, degradando en consecuencia la disponibilidad.
- **Mantenibilidad y evolución**: un diseño con alto acoplamiento o con dificultades para introducir cambios de manera segura impacta directamente en la **Mantenibilidad**. De forma indirecta, esto puede derivar en mayor riesgo de errores durante despliegues o en tiempos prolongados de indisponibilidad ante actualizaciones.

Estas decisiones se abordarán en mayor detalle en las secciones correspondientes a cada atributo de calidad. Aquí basta con señalar que, aunque su impacto sobre la **Disponibilidad** no sea inmediato, sí la condicionan en tanto facilitan (o dificultan) la prevención, mitigación y recuperación frente a fallos.

3.1.2. Escalabilidad (Elasticidad)

Actualmente hay un Nginx que actua como reverse proxy y potencialmente balanceador de carga, pero en este momento solo tiene configurado una sola instancia de la app de Node.js. De todas formas, notamos varios problemas con esto. En principio, la app es stateful porque guarda el estado en memoria y guarda cada tantos segundos el estado de la memoria en distintos archivos json en la carpeta `state/`. Esto hace que no se pueda escalar horizontalmente la app sin perder el estado, ya que cada instancia tendria su propio estado en memoria y no habria forma de sincronizarlos.

3.1.3. Performance

3.1.4. Visibilidad:

Actualmente hay un contenedor de Graphite y otro de Grafana para monitorear el sistema, y tienen algunas metricas en un dashboard creado por la catedra que permite visualizar algunas metricas como Scenarios launched, Request state, Response time y Resources. Faltarian metricas mas especificas del negocio como por ejemplo, volumen de transacciones por moneda, cantidad de clientes activos, etc.

3.1.5. Seguridad

3.1.6. Testabilidad

3.1.7. Portabilidad

3.1.8. Interoperabilidad

3.1.9. Usabilidad

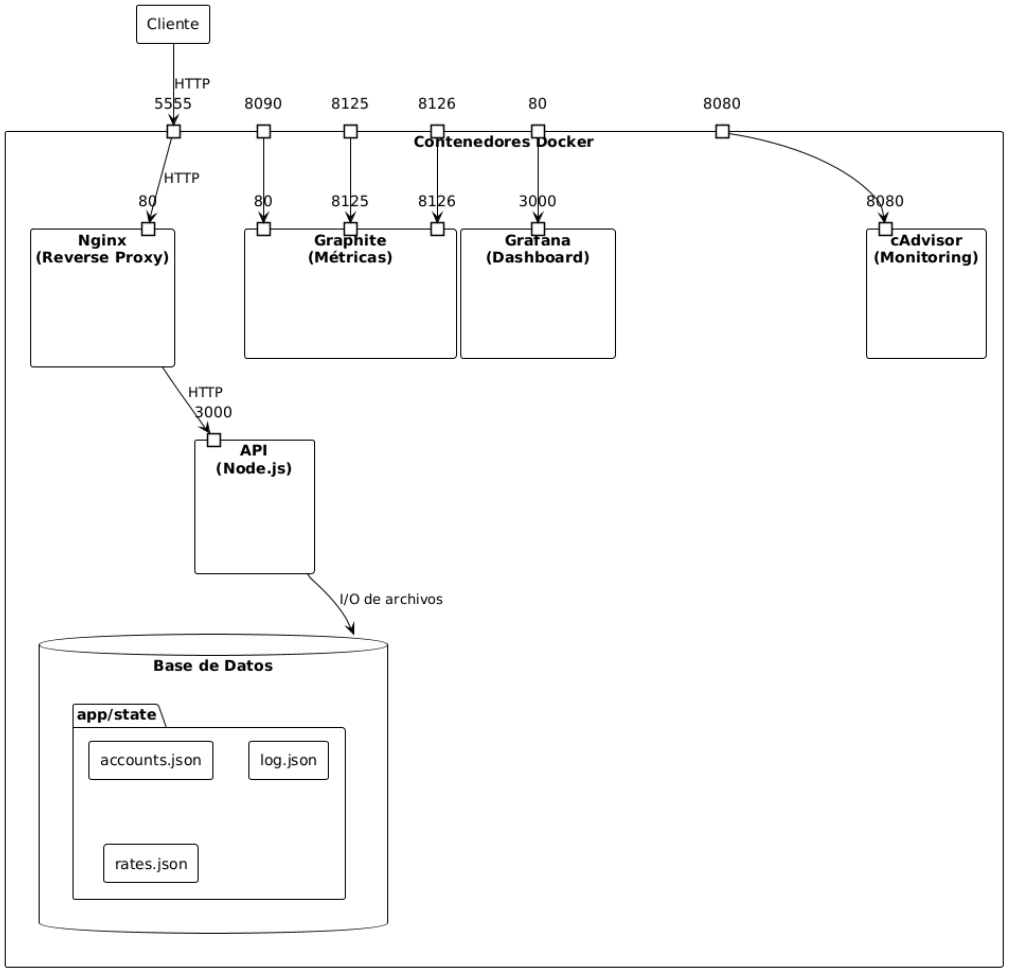
3.1.10. Manejabilidad

3.1.11. Confiabilidad

3.1.12. Simplicidad

3.1.13. Modificabilidad

3.2. Diagrama C&C inicial.



3.3. Crítica a arquitectura base.

4. Metodología de pruebas

4.1. Herramientas usadas (Artillery, Grafana, etc.).

4.2. Escenarios de carga diseñados.

4.3. Métricas recolectadas.

5. Resultados – Caso base

5.1. Screenshots de dashboards.

5.2. Interpretación de resultados.

6. Propuestas de mejora

6.1. Tácticas aplicadas.

Intentamos agregar una base de datos tipo Redis (específicamente Valkey, un fork de Redis) para tener persistencia real y poder escalar horizontalmente los nodos. Esta solución aborda los problemas de la persistencia basada en archivos JSON locales, que no permiten compartir estado entre instancias y carecen de atomicidad en las operaciones.

6.1.1. Implementación de Valkey para persistencia

La implementación utiliza Valkey como almacén de datos centralizado, reemplazando la persistencia en archivos JSON. Los datos se almacenan como claves en Redis:

- `accounts`: Almacena la lista de cuentas de usuario en formato JSON.
- `rates`: Contiene las tasas de cambio entre monedas.
- `log`: Registra el historial de transacciones realizadas.

El módulo `valkey.js` proporciona funciones asíncronas para inicializar la conexión (`init()`), obtener datos (`getAccounts()`, `getRates()`, `getLog()`) y actualizarlos (`setAccounts()`, `setRates()`, `setLog()`). Estas funciones serializan/deserializan los datos a JSON para almacenarlos como strings en Redis.

En `exchange.js`, se importa y utiliza este módulo para todas las operaciones de persistencia, reemplazando las lecturas/escrituras directas a archivos. La inicialización se realiza al inicio de la aplicación con `await valkeyInit()`.

6.1.2. Configuración en Docker Compose

Se agregó un servicio `valkey` en el `docker-compose.yml` utilizando la imagen `valkey/valkey:8.1.4-alpine`, expuesto en el puerto 6379. La aplicación se conecta mediante la variable de entorno `VALKEY_URL=redis://valkey:6379`

6.1.3. Beneficios para escalabilidad horizontal

Al centralizar el estado en Valkey, múltiples instancias de la API pueden compartir el mismo almacén de datos. Esto elimina la dependencia de estado local en memoria o archivos, permitiendo:

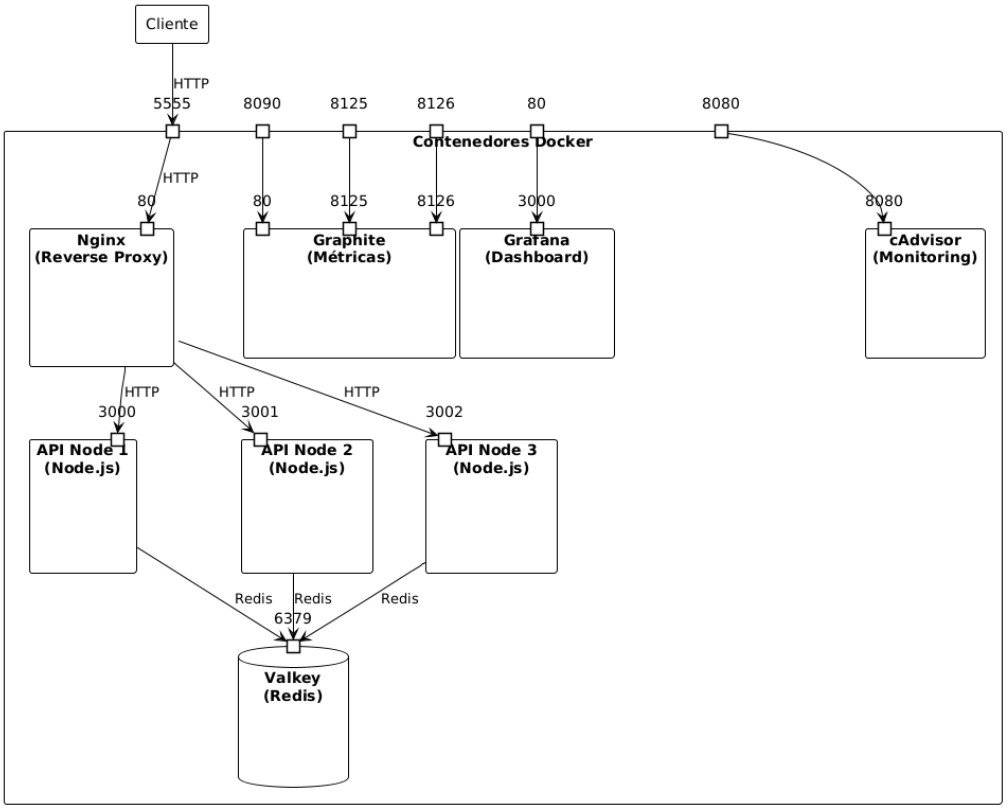
- Escalado horizontal sin pérdida de consistencia.
- Persistencia real de los datos, sobreviviente a reinicios de contenedores.

- Operaciones atómicas en Redis para transacciones financieras.

Esta táctica mejora significativamente la Disponibilidad y Escalabilidad, mitigando los puntos únicos de falla relacionados con la persistencia local.

6.2. Justificación de por qué se eligieron.

6.3. Diagramas C&C modificados.



7. Resultados – Casos mejorados

7.1. Screenshots comparativos.

7.2. Análisis de impacto sobre QA.

7.3. Trade-offs detectados.

8. Conclusiones

8.1. Resumen de hallazgos.

8.2. Beneficios y limitaciones de las mejoras.

8.3. Futuro (ej: pasar a DB externa en TP2).

9. Anexos

9.1. Configuraciones de escenarios de Artillery.

9.2. Configs modificadas de Nginx, Docker Compose, etc.