🎯

Challenge

🧭 Enunciado:

Los requerimientos son los siguientes:

1. Debes desarrollar una API REST en Spring Boot utilizando java 11 o superior, con las siguientes funcionalidades:
    a. Debe contener un servicio llamado por api-rest que reciba 2 números, los sume, y le aplique una suba de un porcentaje que debe ser adquirido de un servicio externo (por ejemplo, si el servicio recibe 5 y 5 como valores, y el porcentaje devuelto por el servicio externo es 10, entonces (5 + 5) + 10% = 11). Se deben tener en cuenta las siguientes consideraciones:
        i. El servicio externo puede ser un mock, tiene que devolver el % sumado.
        ii. Dado que ese % varía poco, podemos considerar que el valor que devuelve ese servicio no va cambiar por 30 minutos.
        iii. Si el servicio externo falla, se debe devolver el último valor retornado. Si no hay valor, debe retornar un error la api.
        iv. Si el servicio falla, se puede reintentar hasta 3 veces.
    b. Historial de todos los llamados a todos los endpoint junto con la respuesta en caso de haber sido exitoso. Responder en Json, con data paginada. El guardado del historial de llamadas no debe sumar tiempo al servicio invocado, y en caso de falla, no debe impactar el llamado al servicio principal.
    c. La api soporta recibir como máximo 3 rpm (request / minuto), en caso de superar ese umbral, debe retornar un error con el código http y mensaje adecuado.
    d. El historial se debe almacenar en una database PostgreSQL.
    e. Incluir errores http. Mensajes y descripciones para la serie 4XX.

2. Se deben incluir tests unitarios.
3. Esta API debe ser desplegada en un docker container. Este docker puede estar en un dockerhub público. La base de datos también debe correr en un contenedor docker. Recomendación usar docker compose
4. Debes agregar un Postman Collection o Swagger para que probemos tu API
5. Tu código debe estar disponible en un repositorio público, junto con las instrucciones de cómo desplegar el servicio y cómo utilizarlo.
6. Tener en cuenta que la aplicación funcionará de la forma de un sistema distribuido donde puede existir más de una réplica del servicio funcionando en paralelo.

🖌️ Asunciones:

- Para la suma de números no especifica el tipo: integer, float, money, etc. Así que voy a usar **BigDecimal** porque es el más abarcativo.
- Para el historial:
  - El requerimiento dice “todos los llamados a todos los endpoint junto a la respuesta”, asumo que es solo el body de la misma.
- Como el % cambia cada 30 minutos asumo que el servicio externo me va a devolver un fecha hora de la ultima vez que se modificó, además del porcentage.
- Se asume 30 minutos como tiempo de expiración de un %

💡 Decisiones de diseño:

Api:

> Para el diseño de la api me basé en las buenas prácticas de api rest Full.

- **Endpoint para sumar y aplicar un %:** El requerimiento es que reciba 2 números, los sume, y le aplique una suba de un porcentaje que debe ser adquirido de un servicio externo.

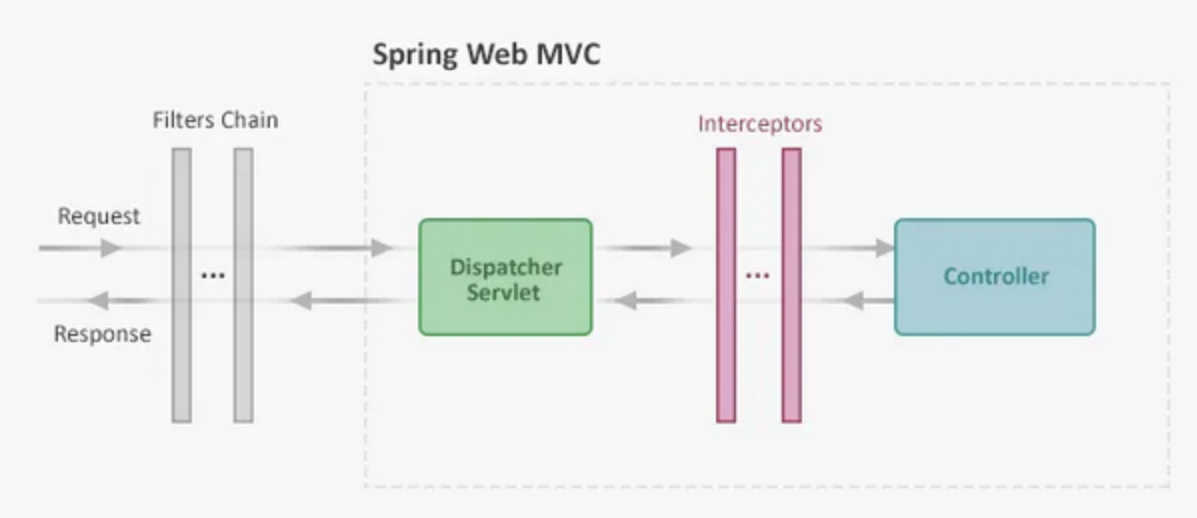
- El método a usar debe ser un **POST**, porque es una acción que realiza el servicio, operando con los parámetros que se le pasan. No es un *PUT* ni un *PATCH* porque no modifica nada existente, tampoco un *GET* porque no es una consulta de un recurso.
- Los parametros pueden ser: *query params, path params o body params*.
  Los path params se usan para designar los recursos a los q se quiere acceder, por ejemplo si se tratan de productos, podemos usar la categoria y un hash para designar al producto al q se quieres acceder, supongamos que queremos consultar una determinada gaseosa, usaríamos así los path params (categoría: gaseosa, hash id: 23123jjh322) {baseUrl}/gaseosa/23123jjh322;
  Entonces nos quedan por analizar si usamos query params o body params, los primeros se usan para acotar o condicionar el request, un buen ejemplo de esto son los parametros de paginación: ?page=1&limt=10, así que el más adecuado es el **body param**;
  Si bien sólo se recibirán dos números, decidí recibirlos en un array [..,…] y no por separado {operando1, operando2}, porque el enunciado es muy genérico, no se qué representan esos valores y además para que pueda escalar sin necesidad de modificar el código ni el contrato, si mañana el requerimiento cambia a 3 o 4 o N números, se puede seguir usando el mismo servicio sin modificarle nada.
  Asimismo, como tampoco se menciona que significa esa operación el nombre que lo voy a asignar es calc.
- Entonces el endpoint quedará así:
  - Path: {baseUrl}/api/api-rest/calc
  - Método: POST
  - Body param: {[1,3]}

- **Endpoint con el histórico de los request:**
  - Por lo expuesto en el endpoint anterior, este es definitivamente un GET.
  - Los parámetros de paginación se tienen que pasar por query param.
    Nota: nada me impide que en un GET pueda mandar body params, pero si queremos usar buenas prácticas de api rest, los GETs nunca llevand body.
  - En resumen:

    - Path: {baseUrl}/api/api-rest/historical
    - Método: GET
    - Query param:
        - page: numero de página comenzando con 0
        - limit: cantidad máxima por página
- Usé el prefijo api para todos los endpoints porque es un standard de la industria y luego “api-rest” porque es el que mencionaba el enunciado.

Guardar historico sin sumar tiempo a las request

- Podemos resolver esto abriendo un hilo de ejecución al final de cada endpoint, de esta manera, el endpoint response y no se queda esperando a que finalice el insert en la db.
- Otra solución es enviar los datos necesarios a una cola de mensajes (como Kafka) y que un consumer tome el mensaje y persista en la db, pero esto es una solución más compleja y dados los requerimientos no vale la pena.
- Esas dos formas de resolverlo tienen la gran desventaja que habría que llamar a un método, que podemos denominar trackRequest pasándole como parámetros el path del endpoint y su respuesta, entonces el metodo quedaría así: void trackRequest(String path, String response), pero es engorroso tener que hacer ese llamado en cada endpoint que agreguemos, sin mencionar lo fácil que sería omitirlo al crear un endpoint nuevo.
- Para resolver este punto lo más simple es usar un componente que nos provee spring: Filter. Entonces vamos a extender una clase de OncePerRequestFilter para que se ejecute una sola vez. Necesitamos guardar en una cache el payload del request y response porque esto pueden ser leidos solo una vez: para esto vamos a crear las clases Cache… que estan en el mismo paquete que el filtro. Por último, para no ocupar tiempo de la request y que una exception persistiendo en el historial no afecte el resultado abriremos un hilo de ejecución.



DB:

- Tabla:

```
CREATE TABLE endpoints_request_historical(
id SERIAL PRIMARY KEY,
endpoint VARCHAR(40) NOT NULL,
params VARCHAR(60) NOT NULL,
response VARCHAR(60) NOT NULL,
date_time TIMESTAMP NOT NULL DEFAULT NOW());
```

- Para la creación de la tabla y eventuales migraciones usé flyway

Cliente del servicio externo

- Para esto creé una interface *PercentageClient* con el método *getPercentage*() y una implementación *MockPercentageClient.*
  *P*ara desarrollar una implementación real, alcanza con implementar la interface y agregar el código usando RestTemplate.
- El Service principal recibe un PercentageClient, esto nos permite cambiar la implementación facilmente y por ambiente (usando spring profile). Por ej en development usar el mock, en otros ambientes otro cuya url se pase por variable de entorno.

- El mock genera un porcentage al azar y si el porcentage es mayor a un umbral (0.70 en este caso) tira una exception, esto nos permite testear diferentes escenarios. También generar un fecha/hora que equivale a la última vez que se actualizó el %, la genera tomando la fecha/hora actual menos un número al azar entre 0 y 30 minutos (tiempo que se tomó como expiración del valor)

Reintentar 3 veces consultar el servicio externo:

- Para esto se puede usar @Retry de Spring, pero implica agregar dos dependencias al pom (retry y aspects), para este caso con un simple loop lo podemos resolver.

Limitar el rpm

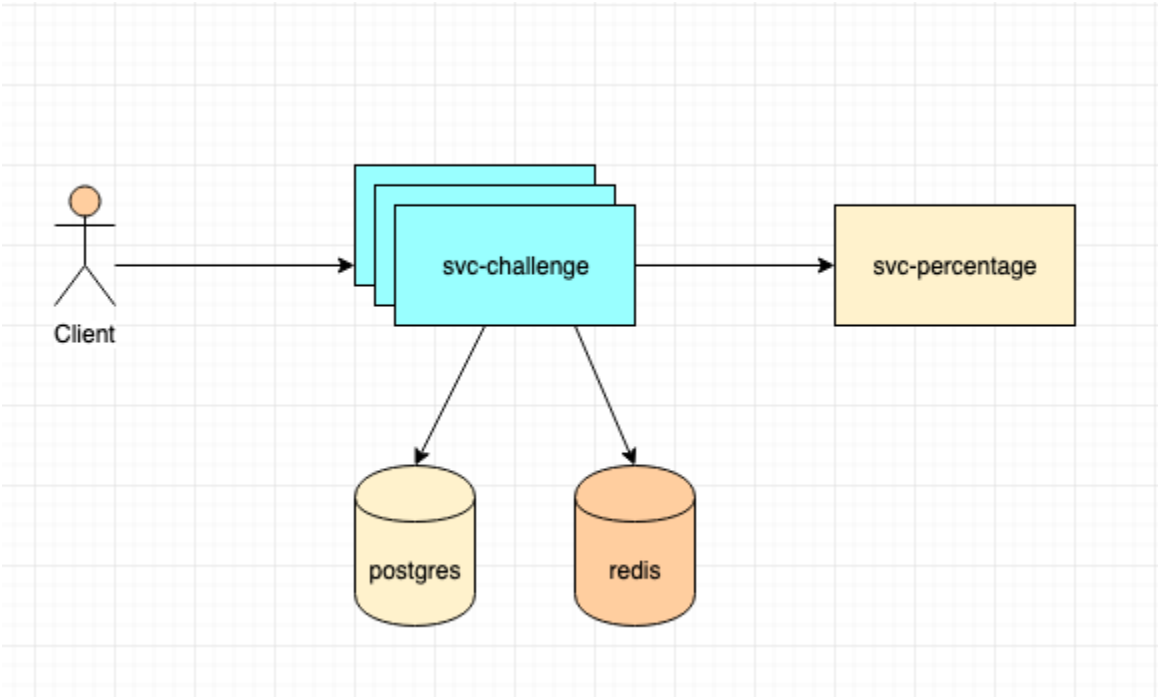
- Para esto voy a usar la lib **Bucket4j** y cuando supere el máximo de 3 el servicio arrojará la exception RateLimitException, con código de error 429, Too Many Requests.
- Para que aplique a todos los endpoints usaré un Interceptor, ubicaré esa validación en el método preHandle que se ejecuta antes del controller.
- Si queremos agregar más seguridad se puede configurar una regla en el firewall, esto se lo podemos pedir al equipo de infra.

Devolución del último valor de % y optimización:

- Para devolver el último porcentage que se usó tenemos dos opciones:
  - Usar una cache
  - Usar la db de postgres: una columna en una tabla nueva
- Como el % varía poco, cada 30 minutos, se necesita recuperar el último valor en caso de que el servicio externo no esta disponible y por cuestión de optimización usaré una cache, en este caso redis, usando jedis.
- El servicio usará una interface CacheService y voy a crear la implementación para Jedis, esto nos permite cambiar fácilmente de implementación.

- Voy a guardar dos valores en la cache, uno tendrá un TTL haciendo este cálculo: tomo el last updated del % le sumo 30 minutos y este sería el tiempo de vencimiento, cuando consulto ese valor y lo voy a guardar en la cache tomo la fecha actual y me fijo cuanto tiempo le queda para que se venza, ese tiempo será el TTL.
  Adicionalmente, lo guardo con otra key con un TTL de 30 minutos, ese será el valor que se use como fallback: si no esta en la cache y si el servicio externo no responde.
- Entonces el algoritmo sería así:
  - Busco el % en la cache
  - Si no está se lo pido al servicio externo, puedo reintentar 3 veces
  - Si no lo pudo obtener voy a la cache a buscar el valor de fallback
  - Si no lo pudo obtener lanzo la exception `PercentageNotFoundException`

Arquitectura:



Testing:

- Usé junit 4 y mockito
- Se los apliqué sólo a las clases que consideré más importantes

---

Documentación:

- Este pdf 😃
- Swagger
- Postman collection
- No usé java doc porque considero que el código es bastante simple y fácil de seguir, ya que lo hice siguiendo principios de clean code. Además esta la documentación mencionada arriba.