Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Arquitectura de software

Obligatorio

Integrantes

- Juan Pablo Sobral - 192247
- Federico Carbonell - 224359

Docentes

- Gerardo Quintana
- Juan Bautista Possamay
- Jorge Marcelo Caiafa

Repositorio

- https://github.com/ORTArqSoft/192247_228990_224359

Introducción

Con motivo del próximo mundial, pero pensando a futuro, la empresa qatarSoftware decidió desarrollar un sistema de ventas de paquetes turísticos enfocados a eventos llamado qEvents.

A alto nivel, la idea de este sistema es que sea utilizado por Administradores, Proveedores y Clientes para poder realizar distintas operaciones asociadas a la compraventa y análisis de los datos generados por estas operaciones.

Requerimientos funcionales

Creación de eventos

POST al endpoint **/events** de la api de eventos con los siguientes campos:

header:
- authorization: string

body:
- name: string
- description: string
- startDate: date
- endDate: date
- country: string
- city: string

Edición de eventos

PUT al endpoint **/events/:eventId** de la api de eventos con los siguientes campos.

header:
- authorization: string

body:
- name: string
- description: string
- startDate: string
- endDate: string
- country: string
- city: string

En el caso de que el evento en cuestión haya sido aprobado, sólo se pueden modificar los campos startDate y endDate.

Aprobación de eventos

PATCH al endpoint **/events/:eventId** de la api de eventos con el siguiente campos:

header:
- authorization: string

body:
- enabled: bool

Este endpoint es exclusivamente para aprobación de eventos, las actualizaciones deberán realizarse mediante PUT y en caso de intentar de hacerlas usando PATCH no van a tener efecto.
El usuario solicitador de la aprobación debe ser distinto al usuario que dio de alta el evento.

Alta de proveedor

POST al endpoint **/supplier** de la api de proveedores con los siguientes campos:

header:
- authorization: string

body:
- name: string
- email: string
- phone: string
- address: Object
    - country: string
    - city: string
- integrationURL: string

Definición de protocolo para obtención de productos

Para el correcto funcionamiento del sistema los proveedores deben implementar dos endpoints, la tecnología utilizada no importa mientras se cumpla OpenApi 3.0.

Los endpoints a implementar son:
- GET /products
- PUT /products/:supplierId/products/:productId

De manera que podamos obtener una lista de todos los productos y otro para poder actualizar el stock al momento de que se realice una venta.

La definición de los protocolos se encuentra correctamente documentada en el repositorio entregado, dentro de la carpeta **suppliers-product-mock/protocol-documentation.** Dentro de esta se encontrará un html de la documentación de swagger y a su vez el esquema que deberá cumplir la api implementada.

Consumo de bitácora de aprobaciones de eventos

GET al endpoint **/events/approvals** de la api de logs con los siguientes campos:

header:
- authorization: string

query:
- since: date (optional)
- until: date (optional)

Consumo de bitácora de actualizaciones de eventos

GET al endpoint **/events/updates** de la api de logs con los siguientes campos:

header:
- authorization: string

query:
- since: date (optional)
- until: date (optional)

Consumo de bitácora de ventas de productos para eventos

GET al endpoint **/sales/products/:productId** de la api de logs con los siguientes campos:

header:
- authorization: string

query:
- since: date (optional)
- until: date (optional)

Se validará que el usuario autenticado sea el dueño del producto.

Lista de eventos

GET al endpoint **/events/** de la api de eventos.

Inicio de venta

POST al endpoint **/transaction** de la api de transacciones con los siguientes campos:

body:
- name: string
- birthdate: date
- country: string

La respuesta a esta consulta es un token que va a ser utilizado a lo largo de la transacción.

Consulta de venta

POST al endpoint **/transaction** de la api de transacciones con los siguientes campos:

body:
- transactionId: string

La respuesta a esta consulta es el estado de la transacción.

Listado de productos por evento

GET al endpoint **/eventsProducts/:eventId** de la api de transacciones.

Compra de producto

POST al endpoint **/purchase** de la api de transacciones con los siguientes campos:

header:
- sessionToken: string

body:
- email: string
- ci: date
- product: object
    - productId: string
    - supplierEmail: string
    - eventId: string
    - quantity: int

La respuesta a esta consulta es una url en la cual realizar el pago para concretar la compra.

Pago de transacción

POST al endpoint **/payment** de la api de transacciones con los siguientes campos:

header:
- sessionToken: string

body:
- fullName: string
- cardNumber: string
- birthDate: date
- billingAddress: string

La respuesta a esta consulta es una url en la cual realizar el pago para concretar la compra.

Consulta de ventas por evento

GET al endpoint **/sales/events/** de la api de logs con los siguiente campos:

header:

- authorization: string

Este endpoint está protegido, por lo cual hay que incluir un header de autorización con un token de **usuario administrador**.

Consulta detallada de evento

GET al endpoint **/sales/events/:eventId** de la api de logs con los siguientes campos:

header:

- authorization: string

Este endpoint está protegido, por lo cual hay que incluir un header de autorización con un token de **usuario administrador**.

Análisis y justificación de la arquitectura

A alto nivel, la arquitectura que decidimos implementar fue una orientada a servicios dado que se nos pedían varias funcionalidades standalone y algunas puntuales que dependerían de otras. Esto nos indicó que lo correcto sería implementar servicios autónomos para ciertas partes de la aplicación que luego sean consumidos por otras, logrando un alto grado de desacoplamiento, buena mantenibilidad y la capacidad de escalar de manera flexible.

En un escenario de deployment en producción, habría ciertas mejoras que se podrían hacer para mejorar la escalabilidad y robustez de la solución.

Primero que nada, colocar un load balancer delante de los servicios de cara a usuarios y potencialmente también en los servicios de administración si hubiera suficiente carga como para justificarlo.

Luego, Redis es utilizado como cache y cómo message queue lo cual no es ideal dado que está cumpliendo dos funcionalidades independientes. En este caso sería razonable manejar dos instancias de Redis, una con cada responsabilidad.

Finalmente, la instancia única de mongo podría ser dividida en varias instancias, dado que las colecciones no están relacionadas entre sí. Esto podría mejorar la performance de los servicios que dependan de estas bases de datos.

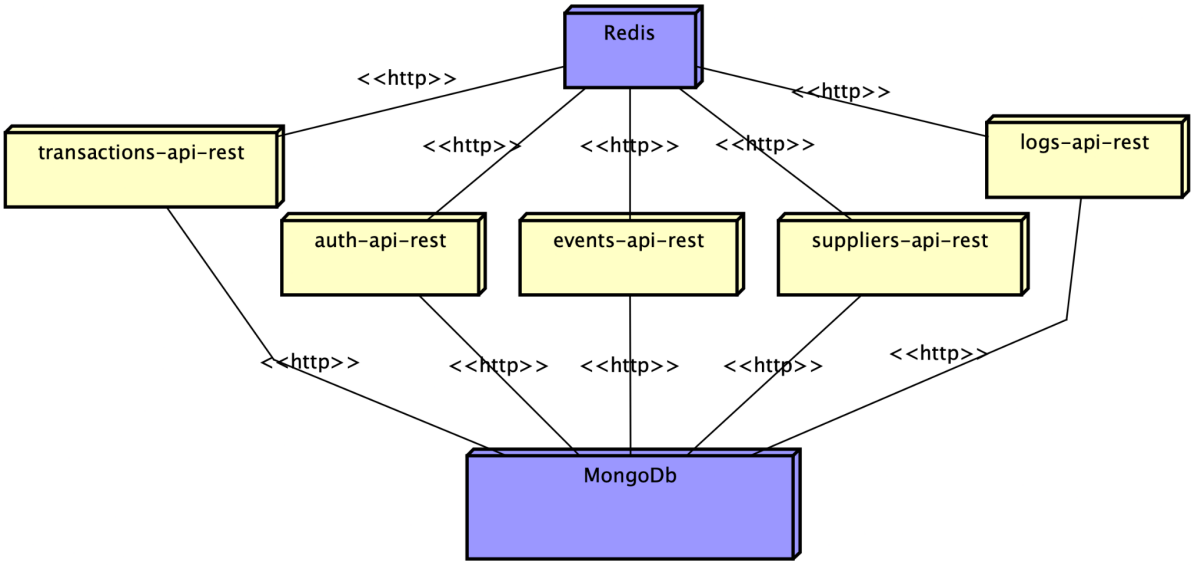
Estas mejoras posibles a la arquitectura, serían simples de implementar dado que utilizamos la táctica de modificabilidad de retrasar el binding. Al levantar de archivos .env la configuración para las bases de datos y redis, con actualizar estos archivos y reiniciar los servicios ya estaría preparada esta solución para ejecutar en una arquitectura más robusta.

En cuanto a decisiones arquitectónicas interesantes, las principales dos que identificamos son las de utilizar un caché para el listado de productos por evento y la de utilizar el patrón publish and subscribe para el manejo de logueo.

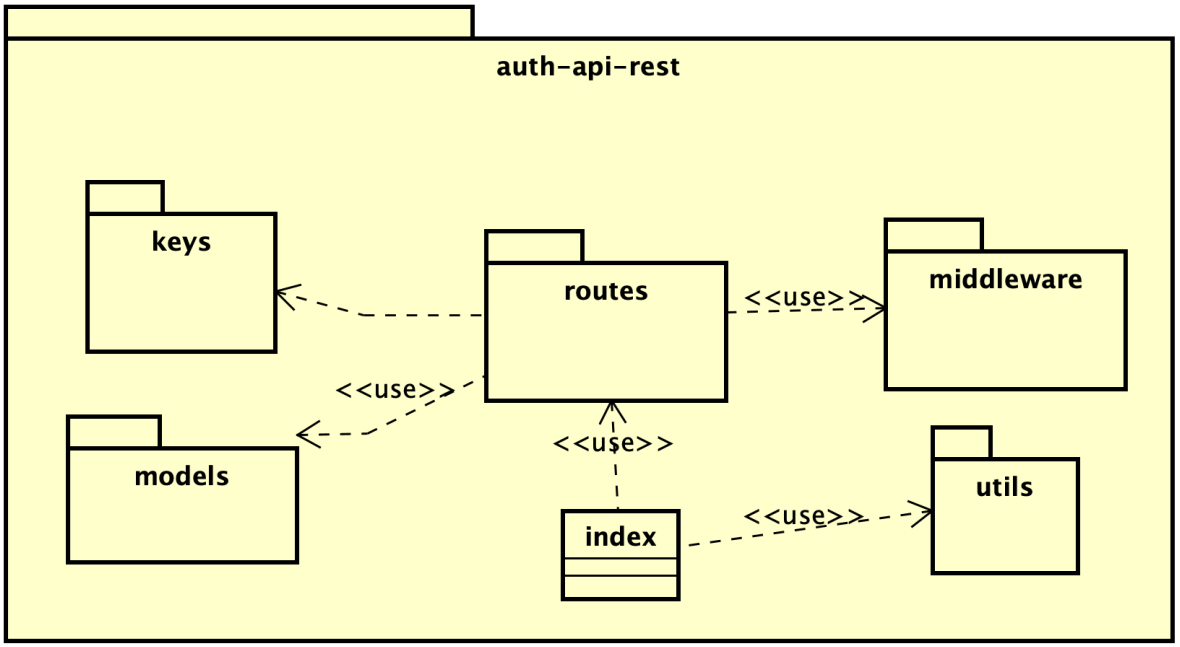
El caché de listado de productos por evento nos permite responder con facilidad a una consulta intensiva, reduciendo o directamente eliminando el gasto de ir a buscar información a la base de datos por cada consulta entrante.

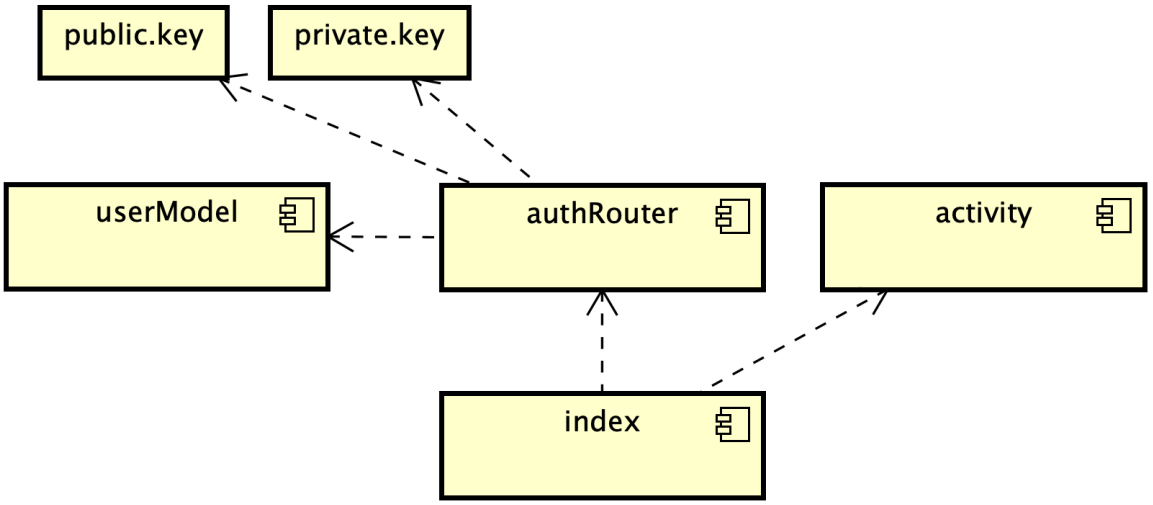
El utilizar publish and subscribe para logueo nos permitió delegar el costo de bajar la información logueada a la base de datos al suscriptor, permitiéndonos no afectar el tiempo de respuesta del servicio con funcionalidades que no le reportan una utilidad al cliente.

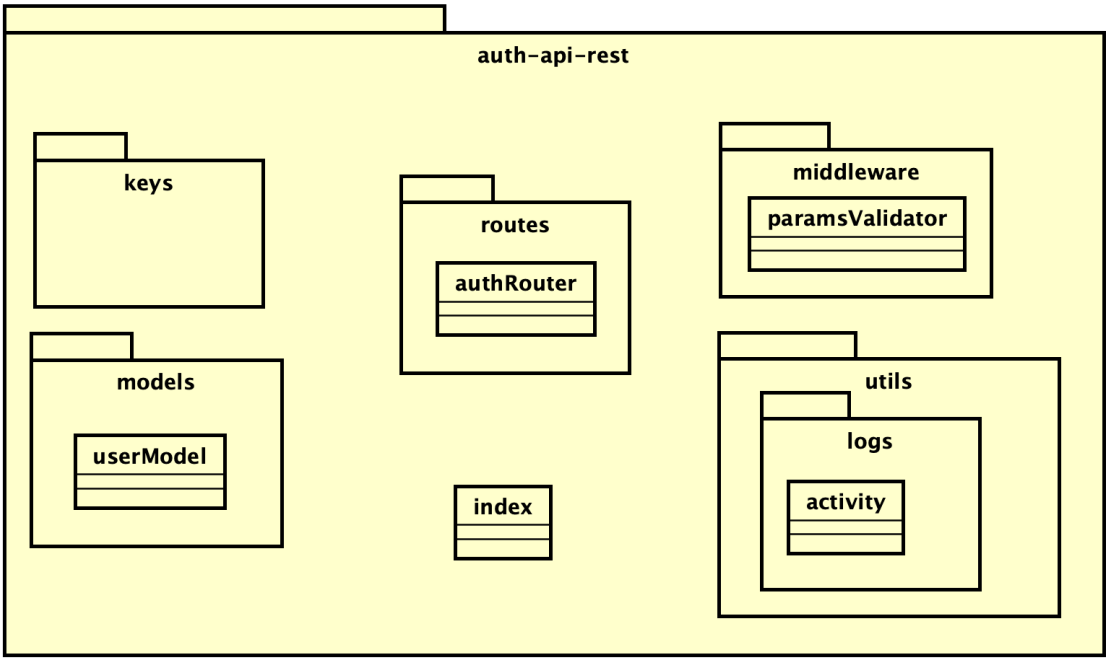
Diagramas de la arquitectura



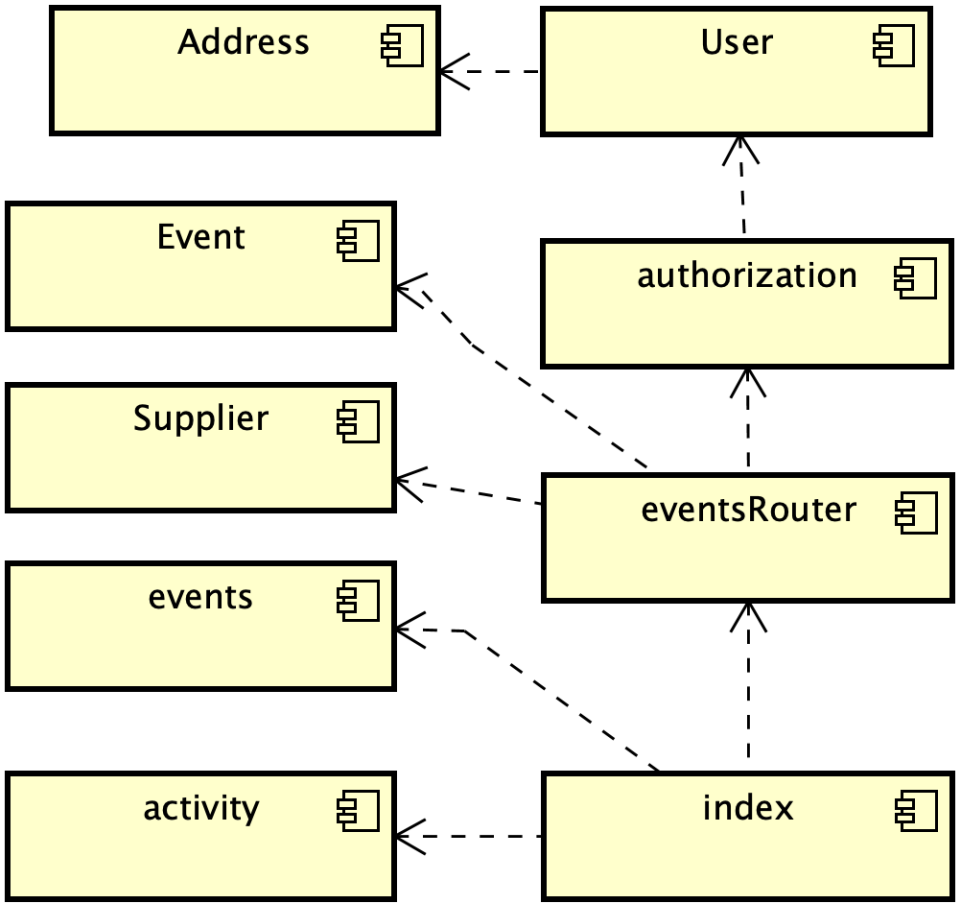
Servicio de autenticación/autorización



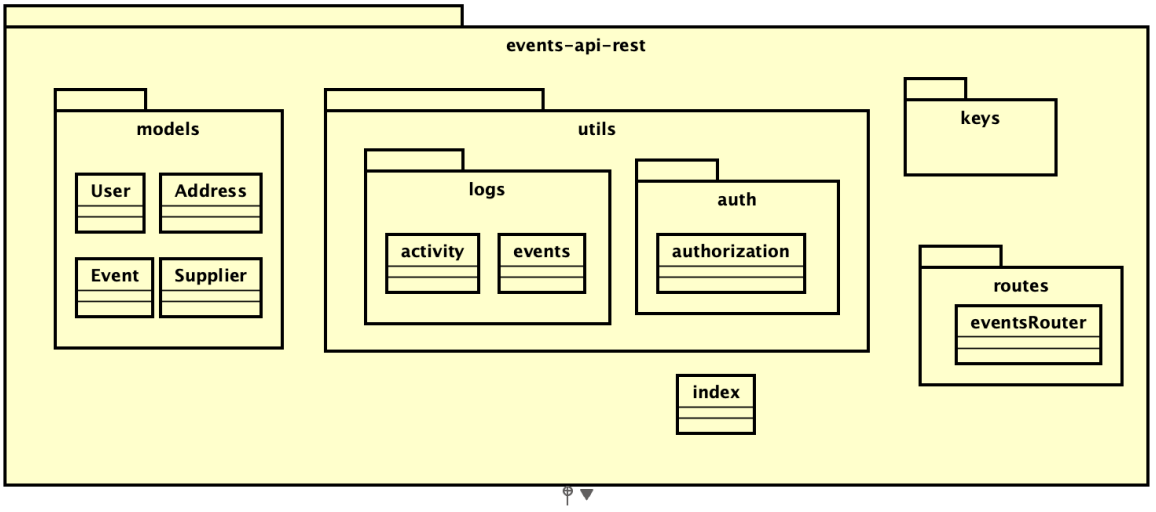

public.key
private.key
userModel
authRouter
activity
index

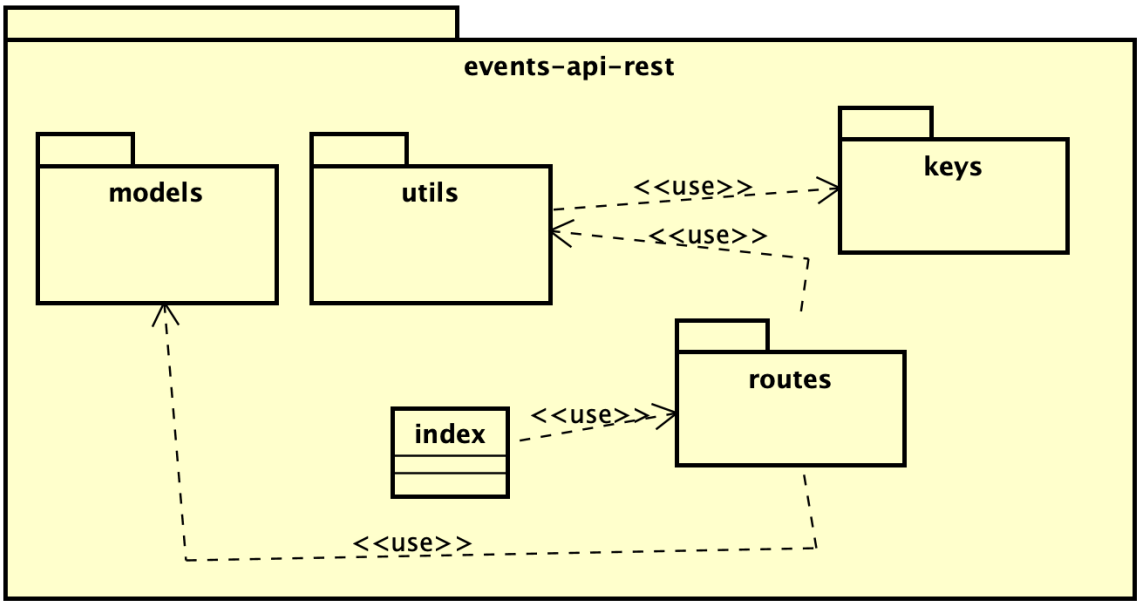

auth-api-rest
keys
routes
authRouter
middleware
paramsValidator
models
userModel
index
utils
logs
activity

Servicio de eventos

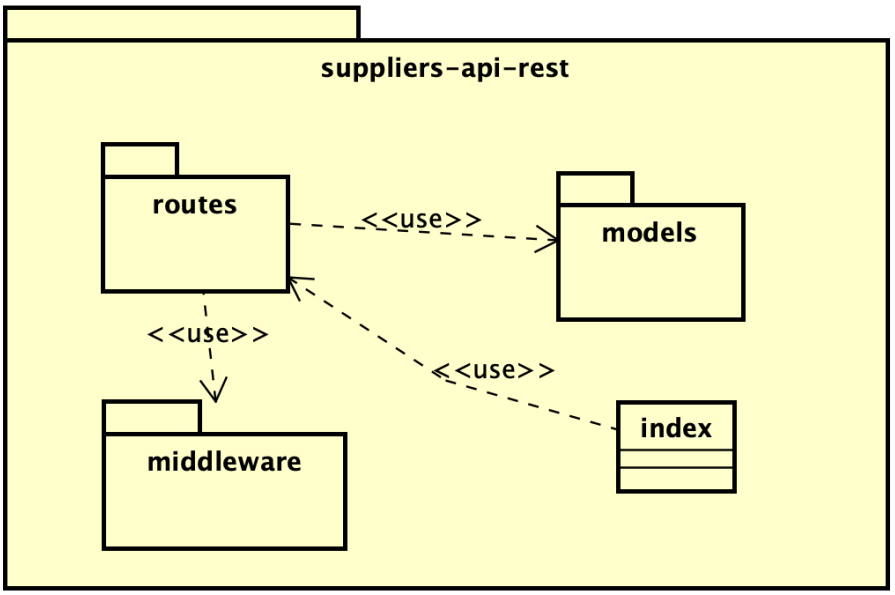


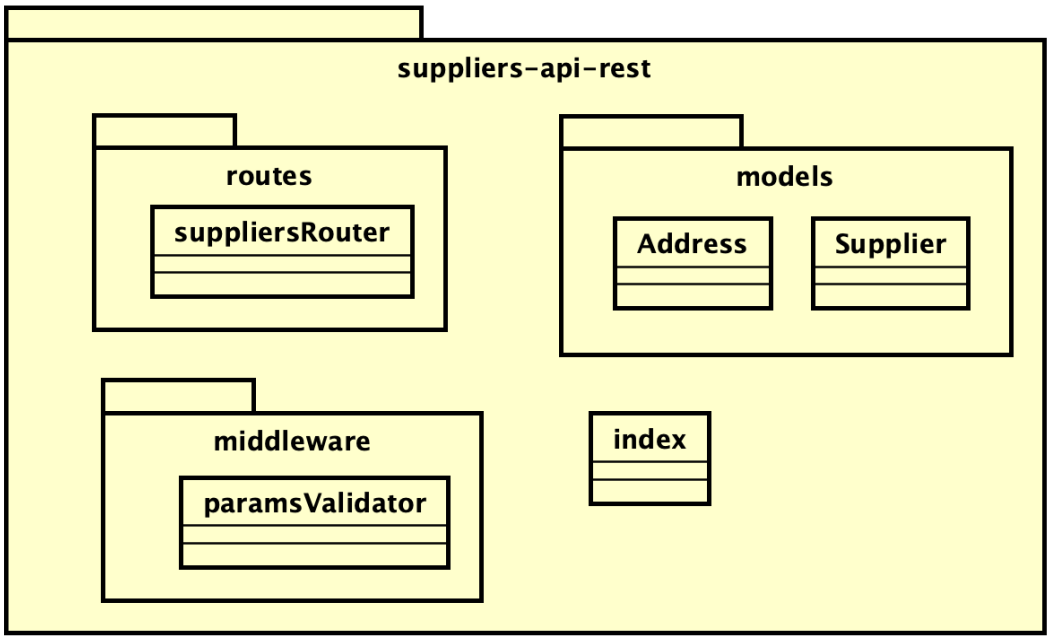
s

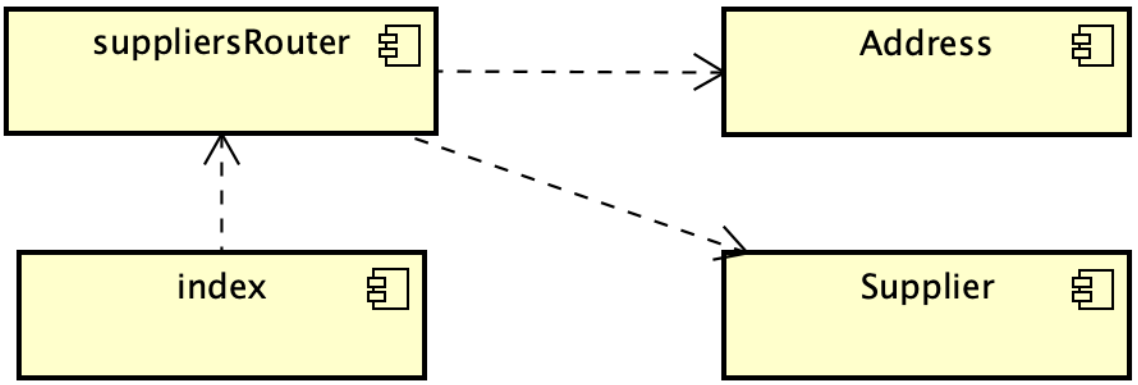



events-api-rest
models
utils
keys
<<use>>
<<use>>
routes
index
<<use>>
<<use>>

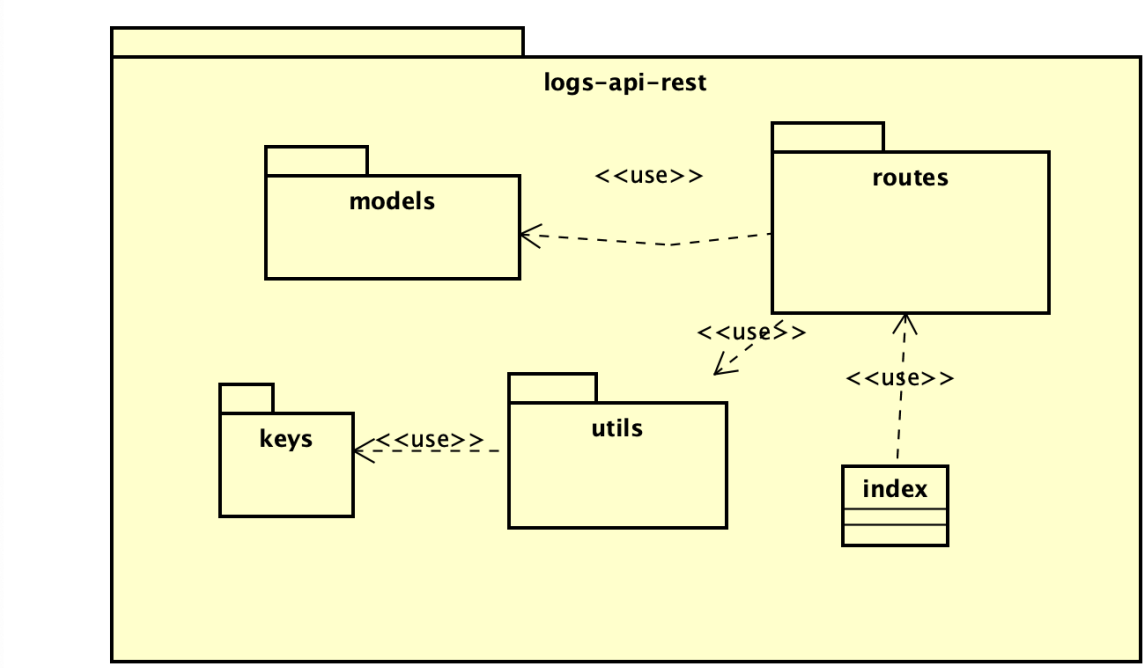
Servicio de proveedores



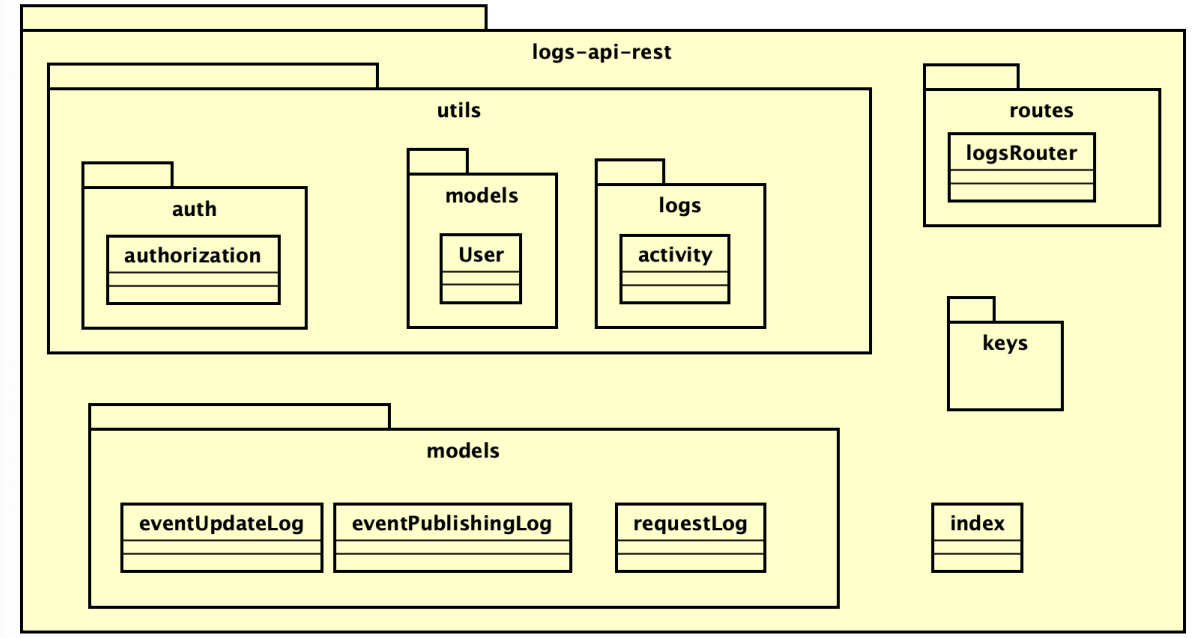


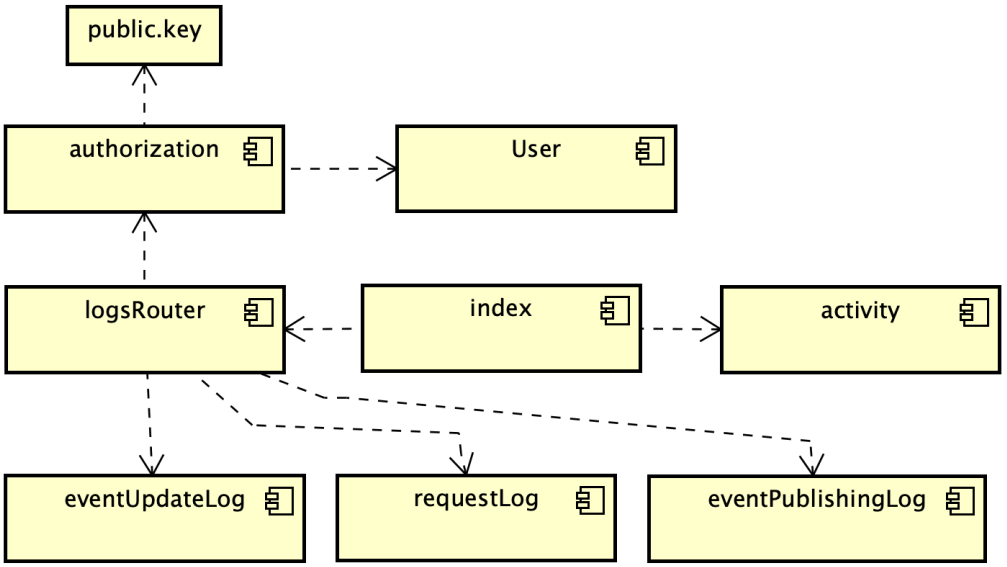

suppliersRouter
Address
index
Supplier

Servicio de log

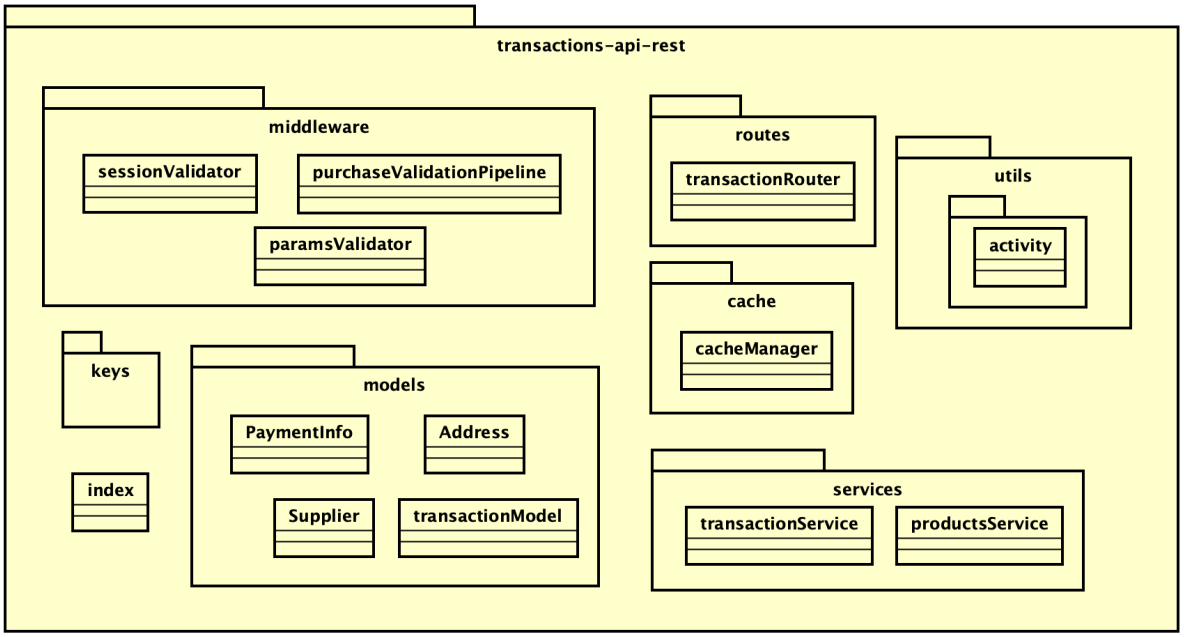


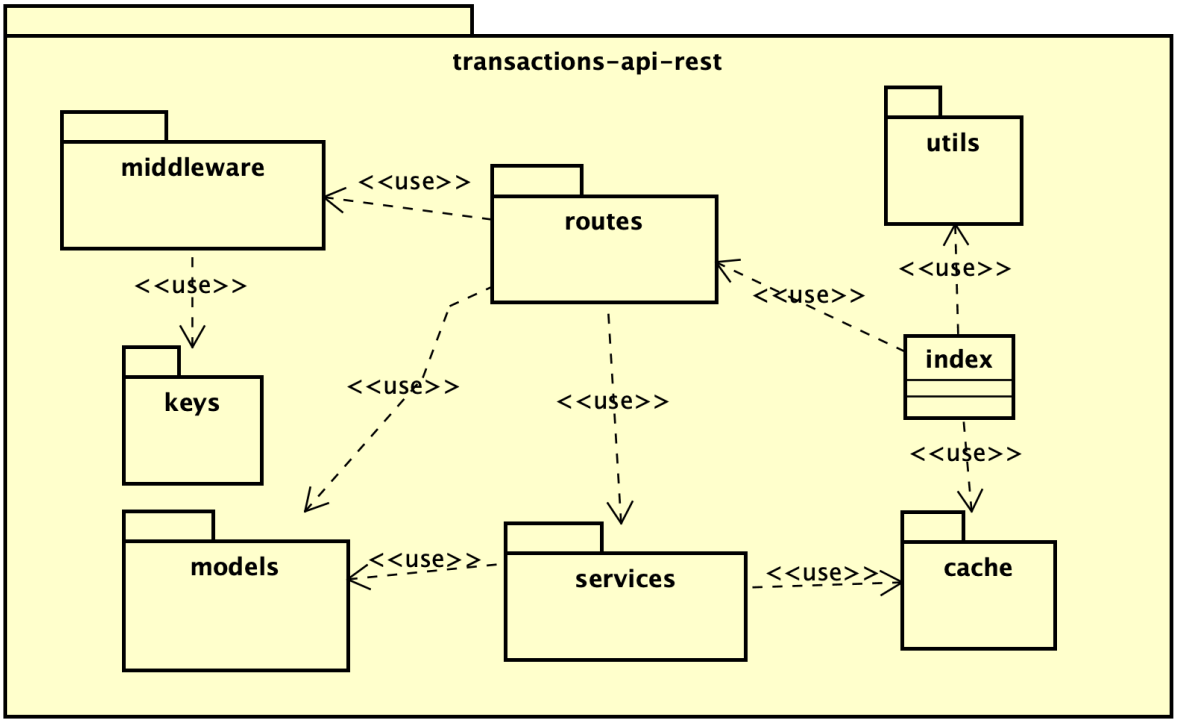
s

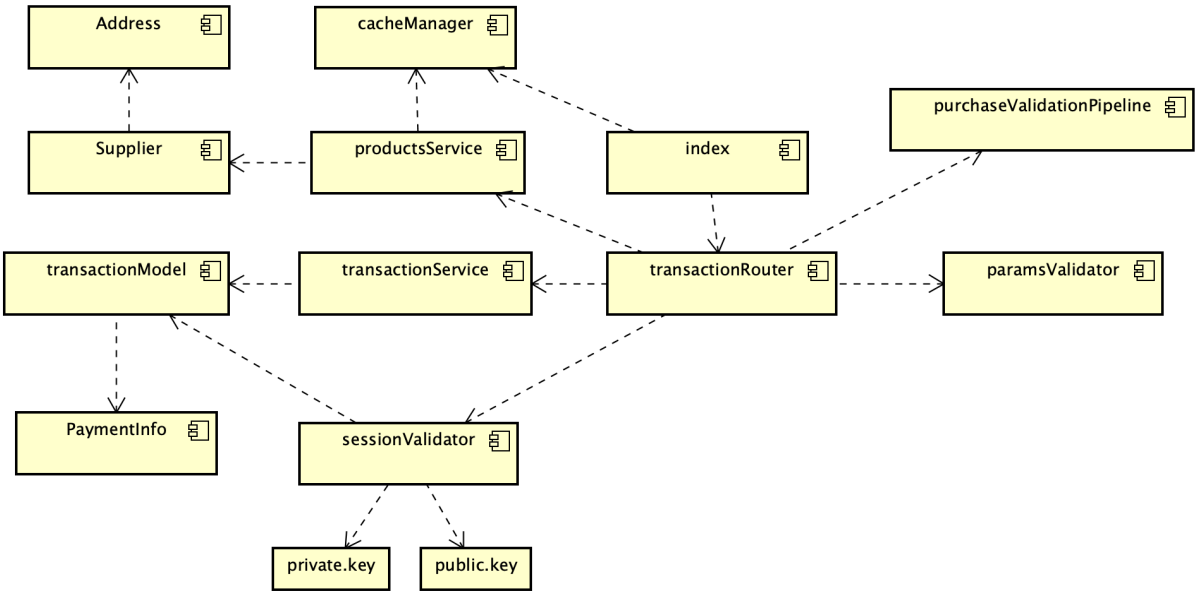



public.key
authorization
User
logsRouter
index
activity
eventUpdateLog
requestLog
eventPublishingLog

Servicio de transacciones






Address
cacheManager
purchaseValidationPipeline
Supplier
productsService
index
transactionModel
transactionService
transactionRouter
paramsValidator
PaymentInfo
sessionValidator
private.key
public.key

Tácticas utilizadas

Tácticas de disponibilidad

Timestamp

Los logs tienen timestamp de entrada y salida, además de timetaken. Esto nos permite por un lado verificar el correcto funcionamiento y potencialmente identificar el degradamiento de ciertos componentes de la solución.

Exception handling

Todos los componentes tienen manejo integrado de excepciones, y rara vez una excepcion produce la caída del componente, siendo la excepción al caso la falla al conectarse con la base de datos, en cuyo caso luego de cierto número de retries se recicla el servicio. Además, el sistema de logueo permite un grado altísimo de trazabilidad sobre las fallas. En caso de que al usuario le llegue un 404, nosotros podemos saber a que recurso quiso acceder. Si le llega un 400 o 500 generico tenemos acceso al endpoint y los datos enviados, de forma de poder reconstruir desde la lógica lo ocurrido.

Tácticas de modificabilidad

Pipes and filters

El uso de pipes and filters no es en sí mismo una táctica de modificabilidad, pero decidimos incluirlo en esta sección dado que es una táctica de arquitectura que beneficia este atributo de calidad.

Redistribute responsibilities - Abstract common services

A medida que avanzabamos en el desarrollo nos fuimos dando cuenta de que tendíamos a repetir ciertas funciones en distintos componentes, por lo cual se decidió agruparlas en un módulo al que todos los componentes tienen acceso. El tipo de funciones que se pueden llegar a encontrar ahí, es por ejemplo middleware para logueo.

Use an Intermediary

Para el sistema de logueo, por preocupaciones de acoplamiento, decidimos separar la responsabilidad de loguear de cada uno de los componentes, y simplemente hacer middleware que estos usen, de forma de publicar en una cola de mensajes los logs, y tener un suscriptor del otro lado que se encargue de consumir y bajar a la base de datos estos logs.

Resource files

Nos apoyamos en la librería dotenv para poder hacer el bindeo de valores en tiempo de ejecución en lugar de tiempo de compilado o build.

Tácticas de performance

Limit event response

Apoyandonos en la librería x encolamos las consultas que llegan a los endpoint de venta, de forma de no generar problemas de concurrencia y consistencia de datos.

Increase resource efficiency

En todos los casos en que una consulta requiere bajar a la base de datos, nos tomamos el trabajo de hacer una consulta lo más compleja posible en cuanto a obtener los datos ya filtrados y potencialmente agrupados, delegando la optimización de la misma a la base de datos, y asegurándonos de no tener que volver a realizar una operación tan costosa.

Maintain multiple copies of data

Para asegurar la performance en la operación de obtener los productos para un evento dado, se hace un cacheo de los datos usando Redis, agilizando el acceso a los datos que se consultan en este tipo de operaciones, de forma de asegurar el cumplimiento del RNF mencionado previamente.

Tácticas de seguridad

Identify actors

Para cualquier operación que requiera autenticación, en los logs de actividad se puede ver el identificador del usuario que la lleva a cabo, de forma de tener trazabilidad respecto a la actividad de los usuarios.

Authenticate actors

Uno de los componentes de la aplicación es el servicio de autenticación, que permite el registro y autenticación de los usuarios retornando un JWT firmado.

Authorize actors

Dos de las funciones disponibles en el módulo de utilidades, son middleware que se utiliza para validar los permisos del usuario autenticado, es decir, el rol del mismo.

Requerimientos no funcionales y tácticas utilizadas

Listado de productos para eventos

Un requerimiento planteado fue que las consultas de listado de productos para un evento dado no tome más de un segundo en responder.

Para cumplir con este atributo de calidad, decidimos decantarnos por una táctica de la categoría **Gestionar recursos**, puntualmente **Mantener múltiples copias de los datos**. Nos apoyamos en un cache de Redis, el cual se popula al levantar la aplicación y se actualiza cuando se responde una consulta que modifique los recursos cacheados además de periódicamente. En nuestro benchmarking, que fue realizado en el contexto de un ambiente de desarrollo local usando docker-compose con acceso a 2 vCPUs y 4gb de RAM, el aplicar la táctica nos reportó mejoras de aproximadamente 50% en los tiempos de respuesta, que sin cache ya eran menores a un segundo pero aplicando la táctica resultaron aún mejores.

Compra de producto

Otro requerimiento planteado fue que las validaciones sobre la información pedida para realizar la operación de compra puedan cambiarse con facilidad.

Para cumplir con este RNF nos inclinamos por implementar el patrón arquitectónico **Pipes and Filters**. Lo implementamos apoyándonos en la librería *pipes-and-filters* y nuestra implementación fue muy sencilla dada que tenía por objetivo aportar **Modificabilidad** a nuestra solución. El pipeline corre en el mismo servidor que lo invoca, y los filtros pueden ser extendidos, modificados o reordenados con facilidad.

Consultas de transacciones

El cliente planteo tambien que las consultas sobre las transacciones no pasen de cinco segundos en tiempo de respuesta.

A la hora de alcanzar este objetivo, utilizamos una táctica de la categoría **Controlar la demanda de recursos**, puntualmente **Aumentar eficacia de uso de recursos**. El recurso en cuestión en este caso es la base de datos. Las consultas sobre transacciones nos piden los datos bastante refinados, además de información que implica comparar datos sobre los mismos.

Lo primero que hicimos fue apoyarnos lo más posible en el motor de base de datos usado (Mongo) para aprovechar la optimización que realiza sobre las consultas. Luego, aprovechando la agrupación de datos brindada y el descarte de datos innecesarios ya por la

consulta a la BD, se hace el procesamiento de datos en memoria para lograr los resultados requeridos. Esto lo decidimos para minimizar la cantidad de consultas a la base de datos necesarias para responder a los clientes, dado que entendemos que es más barato en cuanto a consumo de recursos el procesar a nivel memoria.

Manejo de carga en picos de ventas

Otro pedido fue lograr la mayor capacidad de procesamiento posible sin perdida de datos, logrando la mejor latencia posible. Además se pide no perder disponibilidad del servicio en este tipo de escenarios.

Dado lo amplio que es este pedido, tuvimos que aplicar varias tácticas para cumplirlo. Para no perder disponibilidad del servicio ni perder datos por inconsistencias se utilizó la táctica de performance **Limitar la respuesta a eventos**. Utilizamos la librería *express-queue* para lograr esto, en cierta forma resolviendo dos problemas a la vez porque también soluciona posibles problemas de concurrencia debido a las condiciones de carrera. A su vez el límite se encuentra en un archivo de configuración que puede ser modificado fácilmente sin tener que realizar cambios en la implementación del mismo.

Por otro lado, para lograr una mejora en la capacidad de procesamiento, hicimos uso de un cache en Redis, puntualmente en el caso del listado de productos para eventos, que es la operación más intensiva de las que forman parte de un flujo de venta.

Gestión de errores y fallas

En la letra se pide que el sistema sea capaz de proveer la informacion necesaria para poder diagnosticar las causas de cualquier tipo de falla o error.

Para cumplir con este requerimiento fuimos por dos lados. Primero, todos los servicios tienen un try catch en su método main que conecta con la base de datos y comienza a escuchar por consultas, derivando en que cualquier error que se dé dentro del marco de la respuesta a una consulta sea logueado a consola. Además, uno de nuestros middleware para el pipeline de consultas de express se encarga de loguear una gran cantidad de información respecto a todos los pedidos que ocurren en el sistema. Esto lleva a que si se da un error, podemos diagnosticar con relativa facilidad ya sea mirando la salida de consola del servidor, o mirando los logs a nivel base de datos. La táctica aplicada acá sería **Manejo de excepciones.**

Además se pide que las herramientas o librerías utilizadas para producir la información sobre estos errores o fallas pueda ser intercambiada sin demasiada complejidad. Esto se alcanzó mediante el uso de dos tácticas de modificabilidad: **Abstraer servicios comunes y Usar un intermediario**. Modificando el middleware que se usa para logueo, se impacta sobre todos los componentes que lo utilizan, y habría que simplemente modificar el suscriptor de la cola para que soporte el nuevo formato de los mensajes que va a levantar de la misma.

Conclusiones y mejoras a futuro

- No se está utilizando HTTPS, lo cual sería clave a la hora de deployar la aplicación en producción para lograr la comunicación segura que se pide.
- Ante una caída de la base de datos, los servicios no siguen operativos.
- Ante una caída del redis, los servicios no siguen operativos
- Los mensajes de error no tienen la misma calidad para todos los servicios, nos concentramos principalmente en la API de cara a los clientes. A futuro estaría bueno mejorar todos.
- No utilizamos typescript, pero si se decide convertir el MVP en un producto real, sería importante para alcanzar mejor mantenibilidad.
- A la hora de determinar si devolver el stock ante un pago fallido o no, decidimos ir por el no. La justificación, es que si el pago falla, puede ser por una gran variedad de motivos, y no necesariamente implica que no se vaya a finalizar la transacción, por ejemplo el usuario podría usar una tarjeta que si funcione, y en ese caso solo queda pendiente el pago, no teniendo que realizar todo el flujo de transacción nuevamente (empobrece la UX).

Instructivo de instalación/ejecución

Una vez clonado el repositorio, asumiendo que el usuario cumple con los requisitos previos de tener node y docker instalados y funcionales, simplemente va a tener que pararse en la raíz del proyecto y ejecutar run.sh o run.ps1, que se va a encargar de instalar las dependencias y levantar el compose.