Análisis y Diseño de Aplicaciones II

UT4_TFU

Joaquín Ballara

Franco Filardi

Stefano Francolino

Mateo Hernandez

Mauro Machado

Parte 1

Se seleccionaron 7 patrones arquitectónicos que creemos que modelan de mejor manera el problema de los microservicios de la unidad anterior. Para este trabajo elegimos siete patrones que nos ayudan a que el Gestor de Proyectos sea más sólido y fácil de usar en la vida real.

- En **disponibilidad**, vamos con **Bulkhead** para que un problema en una parte no arrastre a todo el sistema, y **Retry** para reintentar de forma inteligente cuando algo falla por un rato.
- Para **rendimiento**, usamos **CQRS** (separamos lecturas de escrituras para que cada una vaya rápido) y **Materialized View** (dejamos listo el dato “sumado” para los reportes y no recalcular todo cada vez).
- En **seguridad**, optamos por **Federated Identity** para iniciar sesión con proveedores confiables (tipo Google/Microsoft o Facebook) y **Gateway Offloading** para que el gateway se encargue de TLS, validar tokens y limitar abusos, dejando la app enfocada en su lógica.
- En **facilidad de modificación y despliegue**, optamos por **External Configuration Store** para lograr facilitar la modificación de configuraciones -como la cantidad de reintentos o credenciales de la base de datos- sin tener que volver a desplegar los contenedores.

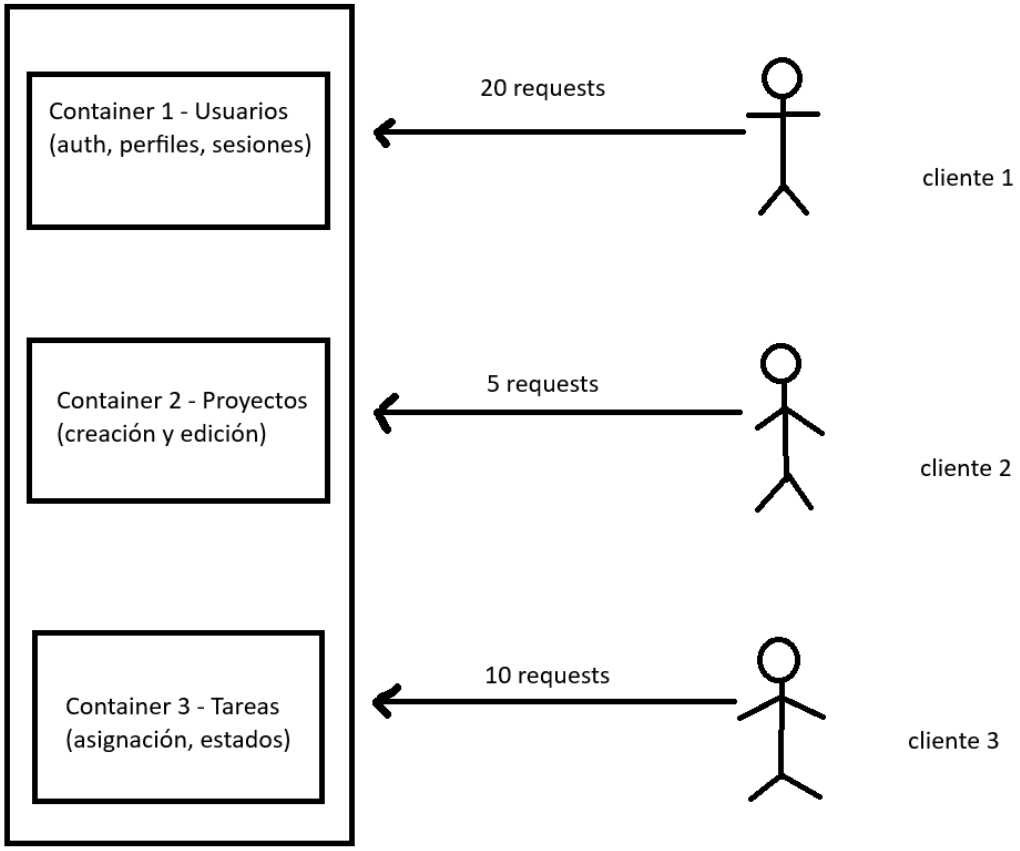
1) Patrones de Disponibilidad

Bulkhead:

El patrón Bulkhead mejora el atributo de disponibilidad al aislar recursos y cargas por compartimentos (por ejemplo, límites de concurrencia por módulo o endpoint). Si un componente sufre saturación o falla, el impacto queda contenido en ese compartimento y el resto del sistema se mantiene operativo, evitando caídas globales y preservando tiempos de respuesta en las funcionalidades no afectadas.

Comprobación empírica: puede verificarse creando un escenario de estrés dirigido por ejemplo un pico de carga sobre /task degrada también /project cuando todo comparte el mismo pool. Al activar Bulkhead (pools y concurrencia separados), los P95/P99 de /project se mantienen prácticamente igual y los errores casi desaparecen, señal de que el problema queda contenido y la disponibilidad mejora.

Diagrama UML:

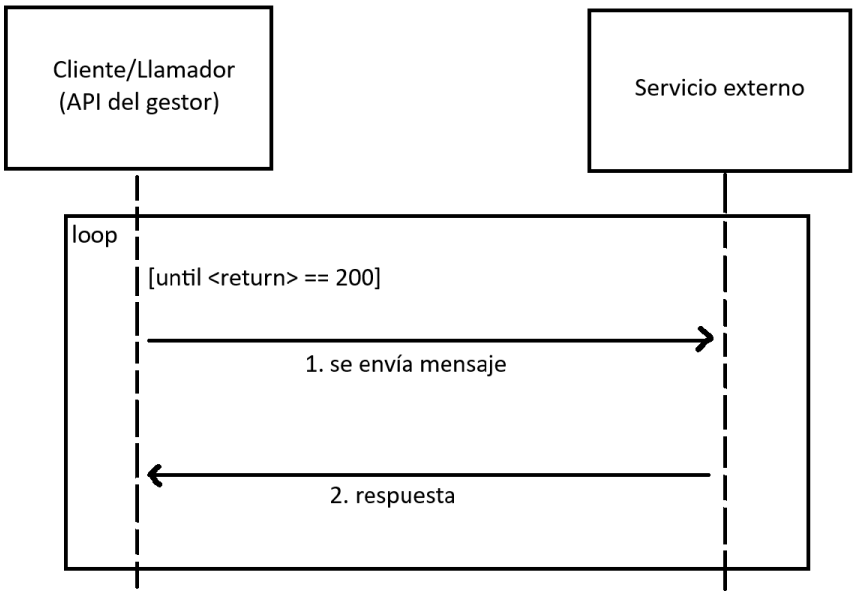


Retry:

El patrón Retry mejora el atributo de disponibilidad al reintentar automáticamente operaciones que fallan por errores temporales como timeouts o cortes de red. Esto permite que el sistema siga respondiendo sin requerir intervención manual, evitando fallas visibles para el usuario y manteniendo la continuidad del servicio.

Comprobación empírica: se puede verificar mediante pruebas que simulen fallos transitorios. Al comparar dos escenarios con y sin Retry, se observa una reducción significativa en la tasa de errores y un aumento del porcentaje de solicitudes exitosas. Además, pruebas de carga y monitoreo de uptime muestran una mayor resiliencia y estabilidad del sistema ante interrupciones momentáneas.

Diagrama UML (Secuencia):



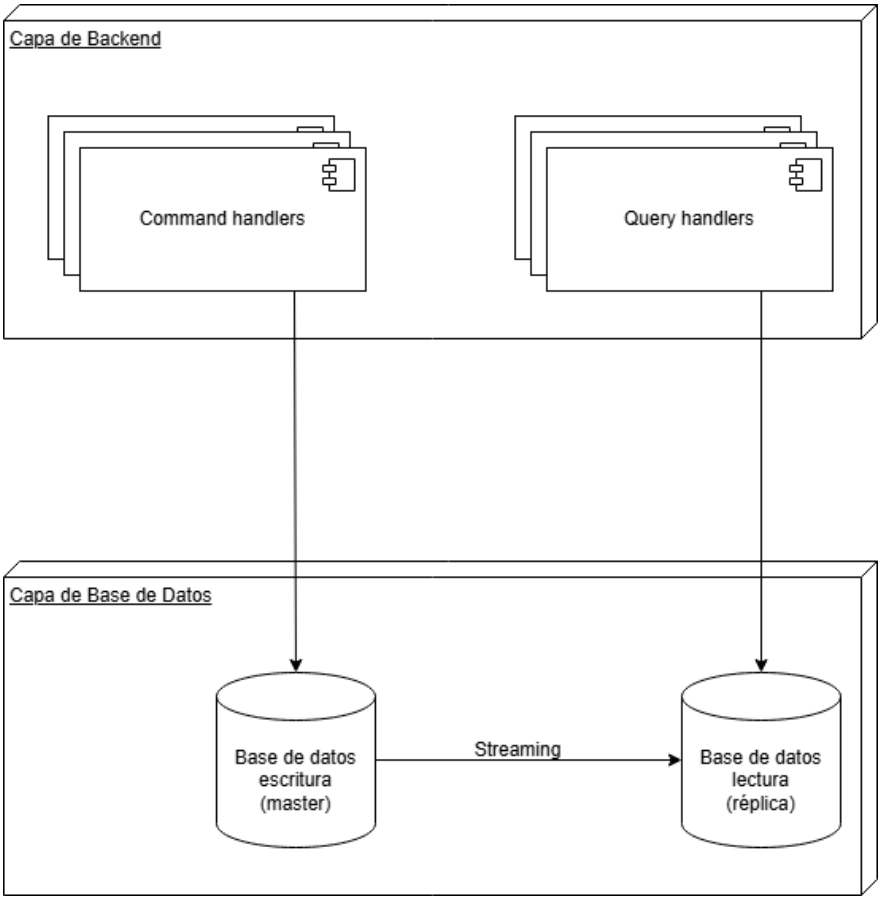
2) Patrones de Rendimiento

CQRS:

Este patrón mejora el atributo de rendimiento al separar las operaciones de lectura y escritura en modelos y rutas diferentes. Esto permite optimizar cada una según su naturaleza: las escrituras se enfocan en mantener la consistencia de los datos, mientras que las lecturas pueden usar estructuras más simples y rápidas para responder consultas. Además, reduce la competencia por los mismos recursos y facilita escalar de forma independiente cada tipo de operación.

Comprobación empírica: se puede medir el impacto realizando pruebas de carga simultáneas de lectura y escritura. En la versión tradicional (sin CQRS), las lecturas suelen presentar mayor latencia cuando aumentan las escrituras. En cambio, con CQRS, las lecturas mantienen tiempos de respuesta más bajos y estables, demostrando una mejor eficiencia bajo carga mixta.

Diagrama UML:



Materialized View:

Este patrón mejora el atributo de rendimiento al almacenar resultados precalculados de consultas costosas (por ejemplo, reportes o métricas agregadas). En lugar de recalcular información cada vez, la aplicación accede a una vista ya actualizada periódicamente, reduciendo drásticamente la carga sobre la base de datos y los tiempos de respuesta.

Comprobación empírica: puede verificarse mediante pruebas de rendimiento que comparen el tiempo de generación de reportes con y sin vista materializada. Al usar este patrón, las consultas complejas muestran una reducción notable en la latencia promedio y en el uso de CPU de la base de datos, evidenciando una mejora significativa en la capacidad de respuesta del sistema.

Diagrama UML:

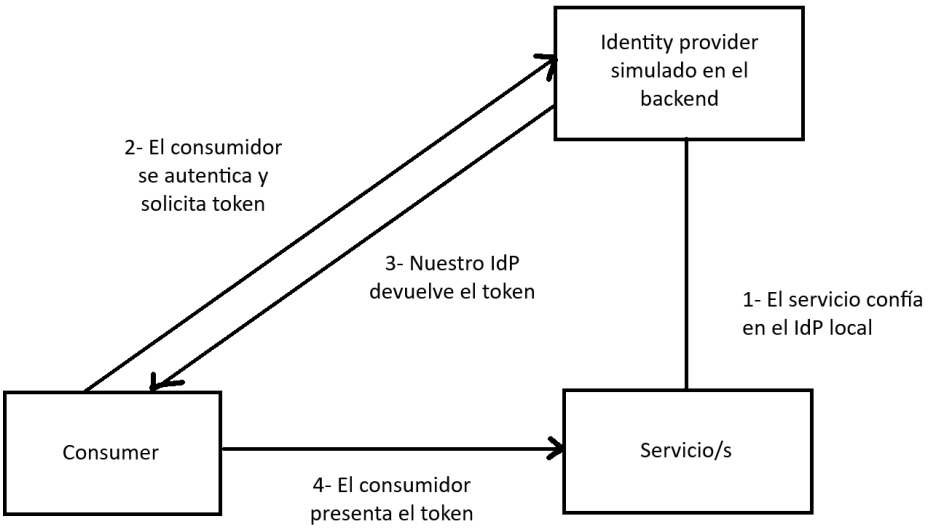
3) Patrones de Seguridad

Federated Identity:

Este patrón mejora el atributo de seguridad al delegar la autenticación de los usuarios a un proveedor externo confiable (por ejemplo, Google o Microsoft). De esta forma, la aplicación no gestiona directamente contraseñas ni datos sensibles, reduciendo el riesgo de exposición o manejo inseguro de credenciales. Además, centraliza el control de identidad, facilitando la revocación y auditoría de accesos.

Comprobación empírica: puede verificarse integrando un flujo de inicio de sesión federado y midiendo la reducción de intentos fallidos o de contraseñas almacenadas localmente. También se pueden realizar pruebas de autenticación simultáneas para confirmar que el sistema solo permite el acceso con tokens válidos emitidos por el proveedor externo, garantizando un control de acceso más robusto y confiable.

Diagrama UML:



Gatekeeper:

El Gatekeeper centraliza la autenticación y autorización en el punto de entrada de la propia aplicación (middlewares/filters en tu backend Express), integrándose con un Proveedor de Identidad (Federated Identity).

Válida JWT (firma, expiración, scopes/roles), aplica políticas de acceso y normaliza el tráfico (CORS, schema validation, logging) antes de ejecutar la lógica de negocio. Con esto, la seguridad mejora porque:

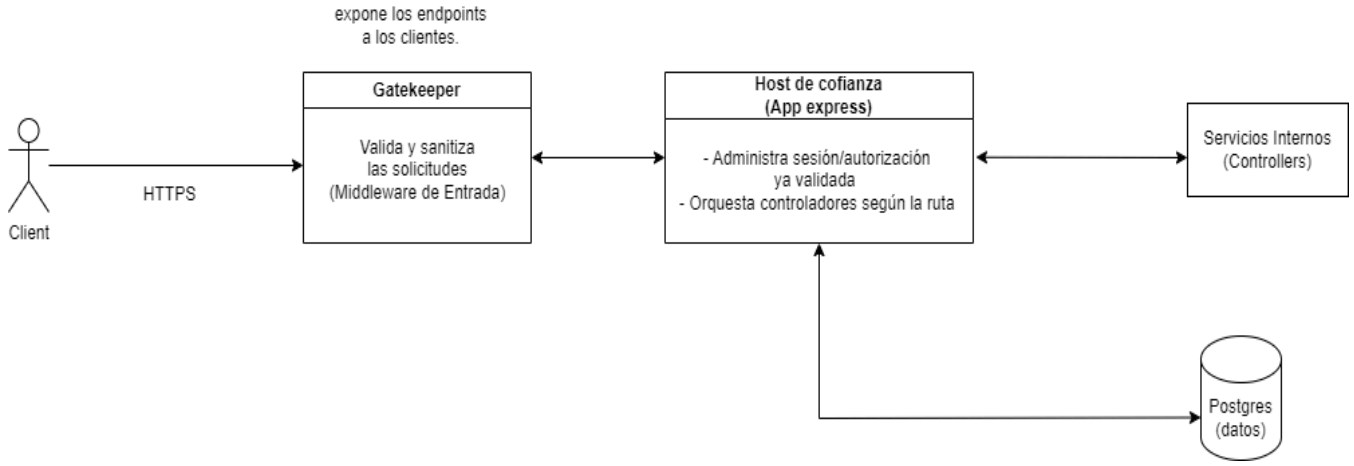
- Todo request **no autenticado o sin permisos** es rechazado **en la puerta** (401/403).
- Se **evita duplicar** lógica de auth en cada controlador/servicio.
- Se **reduce la superficie de ataque**: los endpoints internos solo corren si el contexto de identidad ya fue verificado por el gatekeeper.
- La **federación** permite SSO y gestión centralizada de usuarios/roles, sin manejar credenciales en la app.

Comprobación empírica: Se puede comprobar realizando pruebas con solicitudes no autenticadas o con tokens inválidos, observando que el Gatekeeper las bloquea antes de

que lleguen a los controladores del backend. De igual forma, se pueden enviar peticiones con roles o scopes insuficientes, verificando que el Gatekeeper responde con los códigos adecuados (401 o 403) y que la aplicación interna no ejecuta lógica de negocio.

Además, mediante pruebas de carga con distintos tipos de solicitudes (válidas e inválidas), se puede constatar que el Gatekeeper filtra el tráfico no autorizado y mantiene estables la latencia y el uso de CPU del backend, demostrando su efectividad en la protección y control de acceso centralizado sin degradar el rendimiento general del sistema.

Diagrama UML:



4) Patrón de facilidad de modificación y de despliegue

External Configuration Store:

Este patrón mejora el atributo de facilidad de modificación y despliegue al mantener la configuración del sistema (credenciales, endpoints, claves y parámetros) fuera del código fuente. Esto permite cambiar valores críticos —como URLs de base de datos, claves JWT o rutas de servicios externos— sin necesidad de recompilar ni redeployar la aplicación. Además, reduce el riesgo de exposición de información sensible y facilita la gestión de entornos (desarrollo, prueba y producción) de manera centralizada y segura.

Comprobación empírica: puede verificarse ejecutando la aplicación con variables externas definidas en archivos .env o servicios como Azure Key Vault. Al modificar un parámetro (por ejemplo, la URL de una API externa) y reiniciar el contenedor, el sistema adopta el nuevo valor sin cambios en el código. Esto demuestra una mayor flexibilidad para actualizaciones, mejor seguridad en la gestión de secretos y una administración más ágil entre entornos.

Diagrama UML: